Working Paper Series
ISSN 1170-487X

Proceedings of The First New Zealand Formal Program Development Colloquium

edited by Steve Reeves

Working Paper 94/18
November, 1994

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

The First New Zealand Formal Program Development Colloquium

Department of Computer Science
University of Waikato

30th November and 1st December 1994

Proceedings

Edited by Steve Reeves

Preface

This volume gathers together papers presented at the first in what is planned to be a series of annual meetings which aim to bring together people within New Zealand who have an interest in the use of formal ideas to enhance program development.

Throughout the World work is going on under the headings of "formal methods", "programming foundations", "formal software engineering". All these names are meant to suggest the use of soundly-based, broadly mathematical ideas for improving the current methods used to develop software. There is every reason for New Zealand to be engaged in this sort of research and, of growing importance, its application.

Formal methods have had a large, and growing, influence on the software industry in Europe, and lately in the U.S.A. it is being seen as important. An article in September's "Scientific American" (leading with the Denver Airport débâcle) gives an excellent overview of the way in which these ideas are seen as necessary for the future of the industry. Nearer to home and more immediate are current speculations about problems with the software running New Zealand's telephone system.

The papers in this collection give some idea of the sorts of areas which people are working on in the expectation that other people will be encouraged to start work or continue current work in this area. We also want the fact that this work is going on to be made known to the New Zealand computer science community at large.

The aim of this inaugural meeting was threefold:

> to allow people interested in formal program development to hear and read about some examples of work in this area;
>
> to discuss the formation of a permanent group which could do some or all (or none) of: hold regular meetings; become part of NZCS; be a focus for this sort of work in New Zealand; advertise this sort of work; attract others into this sort of work; provide training courses for industry; develop tools to support this sort of work;
>
> to give a foundation for mutual support for people working in an area which is under-represented in New Zealand but which is already gaining rapidly in importance elsewhere.

These aims are really written above in order of increasing importance. This volume represents the outcome of the first. The outcome of the second aim will not be known until we start the Colloquium. The outcome of the third will take a while longer to become clear.

Thanks are due to all those who have contributed papers - their enthusiasm for writing them and for coming to the Colloquium kept the momentum going and we have a good selection of ideas here which show that New Zealand has something to offer in this area.

Thanks are also due to the Department of Computer Science at the University of Waikato for supporting this Colloquium, with both time and funds, and especially (in no particular order) to Geoff Holmes, Marian McPherson, Bronwyn Webster, Pam and Ian Witten, Dorothy and John Cleary, Carolyn Troup, Chris Knowles, Paul Denize, Mike Vallabh, Murray Person and Mark Apperley.

Steve Reeves
Department of Computer Science
University of Waikato

Contents

An Introduction to the Bird-Meertens Formalism

JEREMY GIBBONS
Department of Computer Science
University of Auckland
Private Bag 92019, Auckland, New Zealand.
Email: jeremy@cs.auckland.ac.nz

ABSTRACT. The Bird-Meertens Formalism, or 'Squiggol', is a calculus for the construction of programs from their specifications by a process of equational reasoning. Developments are directed by considerations of *data*, as opposed to *program*, structure.

This paper presents a brief introduction to the philosophy and notation of the calculus, in the guise of the (well-known) derivation of a linear-time solution to the 'maximum segment sum' problem.

KEYWORDS. Bird-Meertens Formalism, Squiggol, program transformation, program derivation, functional programming.

1 Introduction

In a sentence, the Bird-Meertens Formalism (or 'Squiggol') might be described as a *calculus* for the *construction of programs* by a process of *equational reasoning* from their *specifications*. A lot of meaning is packed into this sentence; let us unpack some of these terms.

By 'calculus' we mean a collection of concepts and notations, together with a body of theorems stating relationships between them. The analogy is with, for example, the differential calculus in mathematics. A calculus is concerned with a particular class of problems in a field, and it aims to embody the 'tricks of the trade' used by practitioners in that field.

By 'construction of programs from their specifications' we mean, essentially, the 'transformational programming' approach to program construction. This is described by Darlington (1981) as follows:

> Using the transformational approach to programming, a programmer does not attempt to produce directly a program that is correct, understandable and efficient, rather he initially concentrates on producing a program which is as clear and understandable as possible ignoring any question of

 This draft dated 1st November 1994. Supported by University of Auckland Research Committee grant number A18/XXXXX/62090/3414013.

> efficiency. Having satisfied himself that he has a correct program he successively transforms it to more and more efficient versions using methods guaranteed not to change the meaning of the program.

Indeed, the notion of 'program' can be relaxed to include also non-executable constructs. In this case, the programmer initially concentrates on producing a 'program' ignoring also any question of implementability (that is, a specification). This must then be transformed into more and more implementable, but still equivalent, versions.

By 'equational reasoning' we mean 'a linear, equational proof that the original specification is extensionally equal to the resulting more efficient version' (Malcolm, 1990). This is the distinguishing characteristic of the Bird-Meertens Formalism ('BMF') over other transformational approaches.

Unpacking further, by 'linear, equational proof' we mean a calculation of the form:

```
    original specification
=       { justification for first step }
    intermediate version
=       { justification for next step }
            ⋮
=       { justification for last step }
    more efficient version
```

The name of the BMF game is to perform as much of the development as is practical in this equational style. This is for expository reasons. It clarifies the distinction between the routine parts of the development (those that consist of straightforward calculation, using 'current technology') and the creative parts (those that we do not yet know how to calculate, but which require more inventive steps). In communicating the *essence* of a development, the calculations can be elided to just their first and last lines:

```
    original specification
=       { routine calculation }
    more efficient version
```

Put another way, we are interested in extending what can be calculated precisely *because* we are not interested in the calculations themselves. Developing techniques that allow more of a derivation to be calculated allows attention to be focussed on the creative parts, which cannot be calculated.

Such an equational calculational style naturally entails a powerful collection of theorems to provide the steps in the calculation. Moreover, these theorems should

be expressed as equalities with just a few simple applicability conditions, so that the flow of the calculational development is not interrupted. Inductive proofs are eschewed as far as possible. The body of theorems used in the BMF arises from a theory of data structures. Ideas borrowed from universal algebra and category theory yield many powerful theorems about common patterns of computation over a data structure, given only the data type definition.

The BMF is applicable to many programming styles and application areas.

- Early work was based on a distinctly sequential intuition, but Skillicorn (1990) and others have shown that the BMF makes just as good a parallel programming language as a sequential one. Indeed, it makes perhaps the best parallel programming language currently available.
- The programming style may be 'functional' (in the sense that it uses higher-order operators to encapsulate common patterns of computation), but the result of a development can just as well be implemented in an imperative language. (The main disadvantage of an imperative programming style over a functional one is the relatively impoverished notion of 'type'. Implementing a functional solution in an imperative language will typically involve a non-trivial data refinement step, from the rich supply of abstract types available in the functional language to the poor supply of more concrete ones in the imperative language.)
- Current work at Oxford (Bird and de Moor, 1993a, 1993b, 1994) and Eindhoven (Backhouse et al., 1991, 1992a, 1992b; Backhouse and Hoogendijk, 1993) is focussed on generalizing the BMF to relations, as opposed to total functions. This allows non-determinism, partial functions and inverses to be expressed naturally, and refinement, rather than equality, to be used as the relationship between steps of a calculation.
- Finally, very similar techniques apply to the development of hardware circuits; the BMF shares much with the *Ruby* (Jones and Sheeran, 1990, 1992) and *Rebecca* (Luk et al., 1994) approaches to hardware design.

Backhouse (1989) describes the impact of the BMF as follows:

> From a calculational perspective [Bird and Meertens'] work represents for me the most major advance that I have encountered in my career as a computing scientist. From the traditional, foundational, perspective, however, their work presents no surprises; the basic theorems and the methods they use are well-known and have been so, in most cases, for several decades. The advance that they have made is to show how concise notation designed around functional algebraic properties can substantially increase the effectiveness of the calculational method.

The purpose of this paper is to give a brief introduction to the philosophy and notation of the BMF. This is done by example, the example in question being the

(well-known) derivation of a linear-time solution to the 'maximum segment sum' problem.

The rest of this paper is structured as follows. In Section 2, we describe the maximum segment sum problem informally. In Section 3, we introduce the notation we need, and in Section 4, we use it to specify the problem formally. In Section 5, we explain *Horner's Rule*, which forms an important part of the solution to the problem. Finally, in Section 6, we calculate this solution.

2 The maximum segment sum problem

Given a list of integers, the *maximum segment sum* ('MSS') problem is to compute the maximum of the sums of any of the non-empty contiguous *segments* of that list. For example, when the list consists of the integers

$$31 \quad -41 \quad 59 \quad 26 \quad -53 \quad 58 \quad 97 \quad -93 \quad -23 \quad 84$$

the segment with the greatest sum, 187, is the one that omits the first two and the last three elements of the list.

The specification 'the maximum of the sums of any of the non-empty contiguous segments' is executable, and takes a number of steps cubic in the length of the list. There is a linear algorithm, arising from *Horner's Rule* for simplifying polynomial evaluation. We introduce just enough notation and laws to express the problem and to derive this algorithm.

This development is due to Bird (1988, 1989), but the problem itself is due to Bentley (1986).

3 Notation

In this section, we introduce just enough notation in order to state and solve the MSS problem.

3.1 Types

We write '$\mathsf{a} : \mathsf{A}$' for the type judgement 'element a has type A'.

3.2 Functions

The identity function is written 'id'.

Function application is written with an infix '.', so that $\mathsf{id.a} = \mathsf{a}$. Function application is right-associative, so that $\mathsf{f.g.a}$ parses as $\mathsf{f.(g.a)}$, and is the tightest-binding infix operator.

We write '$\mathsf{A} \to \mathsf{B}$' for the type of functions from A to B; thus, if $\mathsf{a} : \mathsf{A}$ and $\mathsf{f} : \mathsf{A} \to \mathsf{B}$ then $\mathsf{f.a} : \mathsf{B}$.

Function composition is written with an infix '$\circ$', so that $(\mathsf{f} \circ \mathsf{g}).\mathsf{a} = \mathsf{f.g.a}$. It is the weakest-binding infix operator.

3.3 Binary operators

The BMF makes great use of infix binary operators, where more traditional notations would use prefix alphabetic function names. As Bird (1984) says:

> Not only can such operators enhance the succinctness and, used sparingly, the readability of expressions, they also allow many transformations to be expressed as algebraic laws about their distributive and other properties.

Another notational advantage provided by infix binary operators is known as *sectioning*. A binary operator may be given one of its arguments, yielding a function of its other argument. For example, suppose that $\oplus : A \times B \to C$ (that is, that binary operator $\oplus$ takes a pair of arguments, one of type A and the other of type B, and returns a result of type C), and $a : A$ and $b : B$. Then the 'left section' $(a\oplus)$ has type $B \to C$ and, when applied to b, returns $a \oplus b$. Similarly, the 'right section' $(\oplus b)$ has type $A \to C$ and, when applied to a, also returns $a \oplus b$. Thus, the two sections satisfy the property

$$(a\oplus).b \;=\; a \oplus b \;=\; (\oplus b).a$$

3.4 Lists

If $n \geq 1$ and $a_1, \ldots, a_n$ are all elements of type A, then $[a_1, \ldots, a_n]$ has type $\mathsf{list}.A$ (in other words, it is a list of elements, each of which has type A). We write '$\Box$' for the function that takes a to $[a]$, and '$+\!\!+$' for the associative binary operator that concatenates two lists.

3.5 Map

The most important operation related to lists is the postfix operator '$*$', pronounced 'map'. For a function $f : A \to B$, the function $f*$ has type $\mathsf{list}.A \to \mathsf{list}.B$. Informally, $f*$ applies f to every element of its argument. Thus:

$$f*.[a_1, \ldots, a_n] \;=\; [f.a_1, \ldots, f.a_n]$$

Map is completely determined by the equations

$$\begin{aligned} f*.\Box.a &= \Box.f.a \\ f*.(x +\!\!+ y) &= f*.x +\!\!+ f*.y \end{aligned}$$

It distributes backwards through function composition:

$$(f \circ g)* \;=\; f* \circ g*$$

3.6 Catamorphisms

It can be shown that, for a given function $f : A \to B$ and associative binary operator $\oplus$ of type $B \times B \to B$, there is a unique solution $h : \mathsf{list}.A \to B$ of the equations

$$\begin{aligned} h.\Box.a &= f.a \\ h.(x +\!\!+ y) &= h.x \oplus h.y \end{aligned}$$

This fact is known as the *unique extension property*. We write this unique solution '$(\!|\mathsf{f}, \oplus|\!)$' (it is completely determined by the f and $\oplus$), and call it a *list catamorphism*. Stated another way, the list catamorphism $(\!|\mathsf{f}, \oplus|\!)$ satisfies:

$$\begin{array}{rcl}(\!|\mathsf{f}, \oplus|\!).\Box.\mathsf{a} & = & \mathsf{f.a} \\ (\!|\mathsf{f}, \oplus|\!).(\mathsf{x} +\!\!+ \mathsf{y}) & = & (\!|\mathsf{f}, \oplus|\!).\mathsf{x} \oplus (\!|\mathsf{f}, \oplus|\!).\mathsf{y}\end{array}$$

(and in fact is the only solution to these equations).

The list catamorphism $(\!|\mathsf{f}, \oplus|\!)$ can be thought of as a 'relabelling', replacing every occurrence of $\Box$ by f, and every occurrence of $+\!\!+$ by $\oplus$. For example:

$$(\!|\mathsf{f}, \oplus|\!).(\Box.\mathsf{a} +\!\!+ \Box.\mathsf{b} +\!\!+ \Box.\mathsf{c}) \;=\; \mathsf{f.a} \oplus \mathsf{f.b} \oplus \mathsf{f.c}$$

Many useful functions are list catamorphisms. Some examples are:

$$\begin{array}{rcll}\mathsf{id} & = & (\!|\Box, +\!\!+|\!) & \\ \mathsf{f}* & = & (\!|\Box \circ \mathsf{f}, +\!\!+|\!) & \\ \mathsf{last} & = & (\!|\mathsf{id}, \gg|\!) & \text{where} \quad \mathsf{a} \gg \mathsf{b} = \mathsf{b} \\ \mathsf{sum} & = & (\!|\mathsf{id}, +|\!) & \\ \mathsf{prod} & = & (\!|\mathsf{id}, \times|\!) & \\ \mathsf{max} & = & (\!|\mathsf{id}, \uparrow|\!) & \\ \mathsf{flatten} & = & (\!|\mathsf{id}, +\!\!+|\!) & \end{array}$$

Here, last returns the last element of a list, the binary operator $\uparrow$ returns the greater of its two (numeric) arguments, and flatten concatenates a list of lists into a single long list. Note that, in each of these examples, the second component of the list catamorphism is associative.

3.7 Promotion

The most important property of list catamorphisms is the *Promotion Theorem*. If $\oplus$ and $\otimes$ are associative, and

$$\mathsf{h}.(\mathsf{x} \oplus \mathsf{y}) \;=\; \mathsf{h.x} \otimes \mathsf{h.y}$$

for all x and y (we say 'h is $\oplus$ to $\otimes$ promotable'), then

$$\mathsf{h} \circ (\!|\mathsf{f}, \oplus|\!) \;=\; (\!|\mathsf{h} \circ \mathsf{f}, \otimes|\!)$$

The proof of the Promotion Theorem is by the unique extension property. We have

$$\begin{array}{ll} & (\mathsf{h} \circ (\!|\mathsf{f}, \oplus|\!)).\Box.\mathsf{a} \\ = & \quad \{ \text{ composition } \} \\ & \mathsf{h}.(\!|\mathsf{f}, \oplus|\!).\Box.\mathsf{a} \\ = & \quad \{ \text{ catamorphisms } \} \\ & \mathsf{h.f.a} \\ = & \quad \{ \text{ composition } \} \\ & (\mathsf{h} \circ \mathsf{f}).\mathsf{a} \end{array}$$

and

$$
\begin{array}{ll}
 & (\mathsf{h} \circ (\!|\mathsf{f}, \oplus|\!)).(\mathsf{x} +\!\!+ \mathsf{y}) \\
= & \quad \{ \text{ composition } \} \\
 & \mathsf{h}.(\!|\mathsf{f}, \oplus|\!).(\mathsf{x} +\!\!+ \mathsf{y}) \\
= & \quad \{ \text{ catamorphisms } \} \\
 & \mathsf{h}.((\!|\mathsf{f}, \oplus|\!).\mathsf{x} \oplus (\!|\mathsf{f}, \oplus|\!).\mathsf{y}) \\
= & \quad \{ \text{ assumption of promotability } \} \\
 & \mathsf{h}.(\!|\mathsf{f}, \oplus|\!).\mathsf{x} \otimes \mathsf{h}.(\!|\mathsf{f}, \oplus|\!).\mathsf{y} \\
= & \quad \{ \text{ composition, twice } \} \\
 & (\mathsf{h} \circ (\!|\mathsf{f}, \oplus|\!)).\mathsf{x} \otimes (\mathsf{h} \circ (\!|\mathsf{f}, \oplus|\!)).\mathsf{y}
\end{array}
$$

and so, by the unique extension property, the result holds.

As a corollary, we have the *catamorphism promotion* law:

$$(\!|\mathsf{f}, \oplus|\!) \circ \mathsf{flatten} \;=\; (\!|\mathsf{id}, \oplus|\!) \circ (\!|\mathsf{f}, \oplus|\!)*$$

since, by the promotion theorem, both sides are equal to $(\!|(\!|\mathsf{f}, \oplus|\!), \oplus|\!)$:

$$
\begin{array}{ll}
 & (\!|\mathsf{f}, \oplus|\!) \circ \mathsf{flatten} \\
= & \quad \{ \ \mathsf{flatten} \ \} \\
 & (\!|\mathsf{f}, \oplus|\!) \circ (\!|\mathsf{id}, +\!\!+|\!) \\
= & \quad \{ \ (\!|\mathsf{f}, \oplus|\!) \text{ is } +\!\!+ \text{ to } \oplus \text{ promotable } \} \\
 & (\!|(\!|\mathsf{f}, \oplus|\!), \oplus|\!)
\end{array}
$$

and

$$
\begin{array}{ll}
 & (\!|\mathsf{id}, \oplus|\!) \circ (\!|\mathsf{f}, \oplus|\!)* \\
= & \quad \{ \text{ map } \} \\
 & (\!|\mathsf{id}, \oplus|\!) \circ (\!|\Box \circ (\!|\mathsf{f}, \oplus|\!), +\!\!+|\!) \\
= & \quad \{ \ (\!|\mathsf{id}, \oplus|\!) \text{ is } +\!\!+ \text{ to } \oplus \text{ promotable } \} \\
 & (\!|(\!|\mathsf{id}, \oplus|\!) \circ \Box \circ (\!|\mathsf{f}, \oplus|\!), \oplus|\!) \\
= & \quad \{ \text{ catamorphisms: } (\!|\mathsf{id}, \oplus|\!) \circ \Box = \mathsf{id} \ \} \\
 & (\!|(\!|\mathsf{f}, \oplus|\!), \oplus|\!)
\end{array}
$$

A special case of this is called *map promotion*:

$$\mathsf{f}* \circ \mathsf{flatten} \;=\; \mathsf{flatten} \circ \mathsf{f}**$$

3.8 Rightwards fold

Informally, the rightwards fold $(\mathsf{f}, \oplus)/\!\!\!\rightarrow$ satisfies

$$(\mathsf{f}, \oplus)/\!\!\!\rightarrow.[\mathsf{a}_1, \mathsf{a}_2, \ldots, \mathsf{a}_n] \;=\; ((\mathsf{f}.\mathsf{a}_1 \oplus \mathsf{a}_2) \oplus \cdots) \oplus \mathsf{a}_n$$

The operator $\oplus$ need not be associative, as it must for a list catamorphism. Indeed, the most general typing has $\mathsf{f} : \mathsf{A} \to \mathsf{B}$ and $\oplus : \mathsf{B} \times \mathsf{A} \to \mathsf{B}$, and so $\oplus$ cannot be associative. In this most general case, $(\mathsf{f}, \oplus)/\!\!\!\rightarrow$ has type $\mathsf{list.A} \to \mathsf{B}$.

The rightwards fold $(\mathsf{f}, \oplus)/\!\!\!\rightarrow$ is completely determined by the equations

$$\begin{aligned} (\mathsf{f}, \oplus)/\!\!\!\rightarrow.\Box.\mathsf{a} &= \mathsf{f}.\mathsf{a} \\ (\mathsf{f}, \oplus)/\!\!\!\rightarrow.(\mathsf{x} +\!\!+ \Box.\mathsf{a}) &= (\mathsf{f}, \oplus)/\!\!\!\rightarrow.\mathsf{x} \oplus \mathsf{a} \end{aligned}$$

Every list catamorphism is a rightwards fold; this fact is known as *list catamorphism specialization*. In general, when $\oplus$ is associative, $(\!|\mathsf{f}, \oplus|\!) = (\mathsf{f}, \otimes)/\!\!\!\rightarrow$ where $\mathsf{u} \otimes \mathsf{b} = \mathsf{u} \oplus \mathsf{f}.\mathsf{b}$. For example, $\mathsf{sum} = (\mathsf{id}, +)/\!\!\!\rightarrow$. However, there are rightwards folds that are not list catamorphisms.

3.9 Inits and tails

The two functions inits and tails each take a list and return a list of lists. The first returns all initial segments of its argument, in order of increasing length; the second returns all tail segments, in order of decreasing length. Thus:

$$\begin{aligned} \mathsf{inits}.[\mathsf{a}_1, \mathsf{a}_2, \ldots, \mathsf{a}_n] &= [[\mathsf{a}_1], [\mathsf{a}_1, \mathsf{a}_2], \ldots, [\mathsf{a}_1, \mathsf{a}_2, \ldots, \mathsf{a}_n]] \\ \mathsf{tails}.[\mathsf{a}_1, \mathsf{a}_2, \ldots, \mathsf{a}_n] &= [[\mathsf{a}_1, \mathsf{a}_2, \ldots, \mathsf{a}_n], [\mathsf{a}_2, \ldots, \mathsf{a}_n], \ldots, [\mathsf{a}_n]] \end{aligned}$$

Both are rightwards folds:

$$\begin{aligned} \mathsf{inits} &= (\Box \circ \Box, \oplus)/\!\!\!\rightarrow \qquad \text{where} \quad \mathsf{z} \oplus \mathsf{a} = \mathsf{z} +\!\!+ \Box.(\mathsf{last.z} +\!\!+ \Box.\mathsf{a}) \\ \mathsf{tails} &= (\Box \circ \Box, \otimes)/\!\!\!\rightarrow \qquad \text{where} \quad \mathsf{z} \otimes \mathsf{a} = (+\!\!+\Box.\mathsf{a})*.\mathsf{z} +\!\!+ \Box.\Box.\mathsf{a} \end{aligned}$$

(In fact, both are also list catamorphisms.)

3.10 Rightwards scan

The rightwards scan $(\mathsf{f}, \oplus)/\!\!/\!\!\rightarrow$ consists of applying the rightwards fold $(\mathsf{f}, \oplus)/\!\!\!\rightarrow$ to every initial segment of a list:

$$(\mathsf{f}, \oplus)/\!\!/\!\!\rightarrow \;=\; (\mathsf{f}, \oplus)/\!\!\!\rightarrow * \circ \mathsf{inits} \qquad \text{— (1)}$$

Thus:

$$\begin{aligned} (\mathsf{f}, \oplus)/\!\!/\!\!\rightarrow.[\mathsf{a}_1, \ldots, \mathsf{a}_n] \;=\; & [\mathsf{f}.\mathsf{a}_1, \\ & \mathsf{f}.\mathsf{a}_1 \oplus \mathsf{a}_2, \\ & \ldots \\ & ((\mathsf{f}.\mathsf{a}_1 \oplus \mathsf{a}_2) \oplus \cdots) \oplus \mathsf{a}_n] \end{aligned}$$

It satisfies the equations

$$\begin{array}{rcl} (\mathsf{f}, \oplus)/\!\!/.\square.\mathsf{a} & = & \square.\mathsf{f}.\mathsf{a} \\ (\mathsf{f}, \oplus)/\!\!/.(\mathsf{x} +\!\!+ \square.\mathsf{a}) & = & (\mathsf{f}, \oplus)/\!\!/.\mathsf{x} +\!\!+ \square.(\mathsf{f}, \oplus)/\!\!/.(\mathsf{x} +\!\!+ \square.\mathsf{a}) \\ & = & (\mathsf{f}, \oplus)/\!\!/.\mathsf{x} \otimes \mathsf{a} \qquad \text{where} \quad \mathsf{z} \otimes \mathsf{a} = \mathsf{z} +\!\!+ \square.(\mathsf{last}.\mathsf{z} \oplus \mathsf{a}) \end{array}$$

and so

$$(\mathsf{f}, \oplus)/\!\!/ \;=\; (\square \circ \mathsf{f}, \otimes)/\!\!/ \qquad \text{where} \quad \mathsf{z} \otimes \mathsf{a} = \mathsf{z} +\!\!+ \square.(\mathsf{last}.\mathsf{z} \oplus \mathsf{a}) \qquad -- (2)$$

The important fact about this last equation is that the initial characterization (1) of $(\mathsf{f}, \oplus)/\!\!/$, although executable, requires quadratically many applications of $\oplus$. Characterization (2), on the other hand, requires only linearly many applications. For example, the 'running totals' of a list of numbers is defined by $\mathsf{sum}* \circ \mathsf{inits}$, which involves quadratically many additions. Since sum is a rightwards fold, $(\mathsf{id}, +)/\!\!/$, the running totals can be computed as a rightwards scan, $(\mathsf{id}, +)/\!\!/$, using only linearly many additions. This fact is one of the two important steps in our development of a solution for the MSS problem; the other step is Horner's Rule, as described below.

3.11 Segments

As a final piece of notation, the function segs takes a list and returns a list of lists, consisting of all non-empty contiguous segments of its argument. For example:

$$\mathsf{segs}.[1, 2, 3] \;=\; [[1], [1, 2], [2], [1, 2, 3], [2, 3], [3]]$$

Formally:

$$\mathsf{segs} \;=\; \mathsf{flatten} \circ \mathsf{tails}* \circ \mathsf{inits}$$

The ordering of the segments in the result is necessarily somewhat arbitrary; for our purposes, it will not matter what order segs returns.

4 Specifying the problem

We can now express the MSS problem formally:

$$\mathsf{mss} \;=\; \mathsf{max} \circ \mathsf{sum}* \circ \mathsf{segs}$$

The functions segs produces quadratically many segments, and finding the sum of each takes linear time. Hence this executable specification takes cubic time overall.

5 Horner's Rule

Horner's Rule is a well-known technique for reducing the number of arithmetic operations required to evaluate a polynomial. Instantiated to three terms, it states that:

$$(\mathsf{a} \times \mathsf{b} \times \mathsf{c}) + (\mathsf{b} \times \mathsf{c}) + \mathsf{c} \;=\; ((\mathsf{a} + 1) \times \mathsf{b} + 1) \times \mathsf{c}$$

We can express this for an arbitrary number—in fact, a list—of terms:

$$(\!|\mathsf{id}, +|\!) \circ (\!|\mathsf{id}, \times|\!)* \circ \mathsf{tails} \;=\; (\mathsf{id}, \circledast)/\!\!/ \qquad \text{where} \quad \mathsf{a} \circledast \mathsf{b} = (\mathsf{a} + 1) \times \mathsf{b}$$

The left-hand side requires quadratically many multiplications, the right-hand side only linearly many.

Horner's Rule can be generalized further, abstracting from addition and multiplication the actual algebraic properties required. Suppose that

- $\otimes$ distributes backwards through $\oplus$:

$$(a \oplus b) \otimes c \;=\; (a \otimes c) \oplus (b \otimes c)$$

- $\otimes$ has left unit e:

$$e \otimes a \;=\; a$$

Then

$$(\!|\mathsf{id}, \oplus|\!) \circ (\!|\mathsf{id}, \otimes|\!)* \circ \mathsf{tails} \;=\; (\mathsf{id}, \circledast)\not\rightarrow \qquad \text{where} \quad a \circledast b = (a \oplus e) \otimes b$$

(In fact, a more general result involving rightwards folds rather than list catamorphisms holds, but we do not need it here.) The proof of Horner's Rule is a straightforward induction.

This generalized Horner's Rule illustrates a major theme of the BMF: to encapsulate common patterns of computation as higher-order operations, and to identify general-purpose theorems concerning these higher-order operations—often with algebraic properties of their component operations as side conditions. Such general-purpose theorems can lead to efficient solutions to algorithmic problems. The MSS problem is a case in point, as we shall now see.

6 Calculating a solution to the MSS problem

We now have all of the machinery needed to derive the linear solution to the MSS problem. In fact, this derivation is particularly attractive because it proceeds as a purely sequential calculation from the clear and simple specification to a very elegant yet non-obvious algorithm. That is, the notations we have developed, together with Horner's Rule, reduce the MSS problem to 'mere calculation'; no creativity is needed in the derivation.

We have:

$$\begin{array}{ll} & \mathsf{max} \circ \mathsf{sum}* \circ \mathsf{segs} \\ = & \quad \{\ \mathsf{segs}\ \} \\ & \mathsf{max} \circ \mathsf{sum}* \circ \mathsf{flatten} \circ \mathsf{tails}* \circ \mathsf{inits} \\ = & \quad \{\ \text{map promotion}\ \} \\ & \mathsf{max} \circ \mathsf{flatten} \circ \mathsf{sum}** \circ \mathsf{tails}* \circ \mathsf{inits} \\ = & \quad \{\ \text{catamorphism promotion}\ \} \\ & \mathsf{max} \circ \mathsf{max}* \circ \mathsf{sum}** \circ \mathsf{tails}* \circ \mathsf{inits} \end{array}$$

$$
\begin{array}{ll}
= & \{ \text{ map distributes through composition } \} \\
& \mathsf{max} \circ (\mathsf{max} \circ \mathsf{sum}* \circ \mathsf{tails})* \circ \mathsf{inits} \\
= & \{ \text{ Horner's Rule: let } \mathsf{a} \circledast \mathsf{b} = (\mathsf{a} \uparrow 0) + \mathsf{b} \} \\
& \mathsf{max} \circ (\mathsf{id}, \circledast)\not/* \circ \mathsf{inits} \\
= & \{ \text{ scan } \} \\
& \mathsf{max} \circ (\mathsf{id}, \circledast)/\!\!/
\end{array}
$$

Horner's Rule is applicable because $+$ has identity 0 and distributes (backwards) over $\uparrow$.

Acknowledgements

Thanks are due to Sue Gibbons, for improving the presentation of this paper considerably.

References

Roland Backhouse and Paul Hoogendijk (1993). *Elements of a relational theory of datatypes.* In Möller et al. (1993), pages 7–42.

Roland Backhouse, Peter de Bruin, Grant Malcolm, Ed Voermans, and Jaap van der Woude (1991). *Relational catamorphisms.* In B. Möller, editor, *IFIP TC2/WG2.1 Working Conference on Constructing Programs from Specifications*, pages 287–318. North-Holland.

Roland Backhouse, Peter de Bruin, Paul Hoogendijk, Ed Voermans, and Jaap van der Woude (1992a). *Polynomial relators.* In M. Nivat, C. S. Rattray, T. Rus, and G. Scollo, editors, *Proceedings of the 2nd Conference on Algebraic Methodology and Software Technology, AMAST '91.* Springer-Verlag.

Roland Backhouse, Peter de Bruin, Grant Malcolm, Ed Voermans, and Jaap van der Woude (1992b). *A relational theory of datatypes.* In STOP project (1992).

Roland Backhouse (1989). *Making formality work for us.* Bulletin of the European Association for Theoretical Computer Science, (38):219–249. Text of a lecture given at 5th British Colloquium on Theoretical Computer Science, RHBNC, London, March 1989.

Jon Bentley (1986). *Programming Pearls.* Addison-Wesley.

Richard S. Bird and Oege de Moor (1993a). *From dynamic programming to greedy algorithms.* In Möller et al. (1993).

Richard S. Bird and Oege de Moor (1993b). *Solving optimisation problems with catamorphisms.* In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *LNCS 669: Mathematics of Program Construction*, pages 45–66. Springer-Verlag.

Richard Bird and Oege de Moor (1994). *Hybrid dynamic programming.* Programming Research Group, Oxford.

Richard S. Bird (1984). *The promotion and accumulation strategies in transformational programming.* ACM Transactions on Programming Languages and Systems, 6(4):487–504. See also (Bird, 1985).

Richard S. Bird (1985). *Addendum to "The promotion and accumulation strategies in transformational programming".* ACM Transactions on Programming Languages and Systems, 7(3):490–492.

Richard S. Bird (1988). *Lectures on constructive functional programming.* In Manfred Broy, editor, *Constructive Methods in Computer Science.* Springer-Verlag. Also available as Technical Monograph PRG-69, from the Programming Research Group, Oxford University.

Richard S. Bird (1989). *Algebraic identities for program calculation.* Computer Journal, 32(2):122–126.

John Darlington (1981). *The structured description of algorithm derivations.* In J. W. deBakker and H. van Vliet, editors, *Algorithmic Languages*, pages 221–250. Elsevier North-Holland, New York.

Geraint Jones and Mary Sheeran (1990). *Circuit design in Ruby.* In Jørgen Staunstrup, editor, *Formal Methods for VLSI Design.* North-Holland.

Geraint Jones and Mary Sheeran (1992). *Designing arithmetic circuits by refinement in Ruby.* In STOP project (1992).

Wayne Luk, Tim Cheung, Quentin Miller, and Graham Hutton (1994). *Simulating and compiling designs using Rebecca.* Unpublished draft, Programming Research Group, Oxford.

Grant Malcolm (1990). *Data structures and program transformation.* Science of Computer Programming, 14:255–279.

Bernhard Möller, Helmut Partsch, and Steve Schumann, editors (1993). *LNCS 755: IFIP TC2/WG2.1 State-of-the-Art Report on Formal Program Development.* Springer-Verlag.

David B. Skillicorn (1990). *Architecture independent parallel computation.* IEEE Computer, 23(12):38–51.

STOP project (1992). *STOP 1992 Summerschool on Constructive Algorithmics.*

Yet Another Introduction to Constructive Type Theory

Steve Reeves
Department of Computer Science
University of Waikato
stever@waikato.ac.nz

Abstract

In this paper we present, very briefly, the background to CTT and introduce some of the proof rules that formalize it. We then do some small examples to show how the rules are used and look at two more general points: using a proof of the axiom of choice as a specification transformer and showing how working constructively can mean that you use information during program construction that, classically, would not be available.

1. Introduction

There has been much written, and there is much to say, about the ideas behind Martin-Löf's Constructive Type Theory (or Constructive Set Theory) [M-L84]. If we move away from the particular formal system that is CTT to the underlying philosophy and mathematics (intuitionism) upon which it is based there is even more to say, and even more people have said it [Bee85, Dum77].

Rather than rehearse all of the arguments for and against intuitionism and for and against CTT, in this paper we will take the formalisation as it is and see some of what we can do with it. Having said that, however, a few points will be made in the Conclusion.

If you have never seen CTT before, you are going to find even these small examples hard to understand (I guarantee). However, there are many accessible introductions, with [M-L84,Bee85,NPS90,Ree91,Tho91] being some examples.

1.1 Some jargon and definitions

Given that there are many ways of presenting CTT, we need to start with some jargon. The descriptions given here of the machinery of the formal system are meant just to introduce some of the words we will use; they are not meant as definitions.

CTT works in the natural deduction style. The basic aim, given a specification S, is to build a derivation with a judgement of the form e : S at its root. Such a derivation will have the form

$$\frac{\begin{array}{cccccc} J11 & A11 \ldots J1n1 & A1n1 & J21 & A21 \ldots J2n2 & A2n2 \end{array} \quad \ldots\ldots \quad \begin{array}{cc} Jk1 & Ak1 \ldots Jknk \quad Aknk \end{array}}{\begin{array}{cc} J1 & A1 \end{array} \qquad \begin{array}{cc} J2 & A2 \end{array} \qquad \begin{array}{cc} Jk & Ak \end{array}}$$
$$e : S \quad A$$

where each of the Jis is a judgement and each of the Ais (and A) is a list of assumptions, which are themselves judgements. The Ji/Ai pair are called the premises of the rule and e : S/A is called the conclusion of the rule. We say that the judgement e : S depends on (or has) the assumptions in A. e : S follows from J1...Jn because of a rule, which will have the form

$$\frac{\mathcal{J}1 \quad \mathcal{A}1 \ \ldots \ \mathcal{J}k \quad \mathcal{A}k}{e : \mathcal{S} \quad \mathcal{A}}$$

where J1, A1,...,Jk, Ak, e, S and A are instances of $\mathcal{J}1, \mathcal{A}1, \ldots, \mathcal{J}k, \mathcal{A}k, e, \mathcal{S}$ and $\mathcal{A}$ respectively.

J1 then follows from J11 A11...J1n1 A1n1 because of a certain rule and J11 is similarly produced by some rule, and so on up the derivation (which has the form of a tree of rule instances). The leaves of the derivation will be judgements which need no further rules to justify them. Such a rule is

$$\frac{}{\text{N type}}$$

which simply says that N is a type without further justification (as we will see later). Note that 'A type' for certain A is another sort of judgement.

In the judgement e : S, e is an element of S (when S is viewed as a type or set), or an object (or program) meeting the specification S or (viewing S logically) e is a proof of the proposition S. ("Types are Specifications", "Propositions are Types" and "Proofs are Programs" are all (true) slogans of CTT). All these readings are equivalent - they simply represent different ways of interpreting the formal object e.

Sometimes it is convenient to introduce abbreviations for certain complicated terms. We shall write

$$d \equiv f$$

to mean that d is an abbreviation for f.

Finally, since 'λ' has a particular meaning within the formal system, and since we want to use the usual idea of abstraction and application (that the λ-calculus expresses) in the machinery of

CTT, we need some alternative notation (not involving 'λ') to denote abstractions and applications of expressions. To this end we write

(x)e

to denote the expression that, unofficially, we might be tempted to write as λx.e. So we have that (x)e(a) is the same as e with all free occurrences of x in e replaced by a (with renaming if needed to avoid 'variable capture' as usual). We say that (x)e is an abstraction and that e(a) is an application. The expression e(a) can therefore be viewed either as an expression containing some free occurrences of a or as some abstraction e followed by '(', followed by some expression a, followed by ')'. That is to say the expressions (x)x+2(3) and 3+2 are the same. Also, we allow abstractions on tuples of variables, so (x,y)x+y(1,2) and 1+2 are the same expression, and repeated abstraction, so (x)(y)x+y(2)(3) and 2+3 are the same.

2. Some proof rules

We have already seen one proof rule, namely the one that says that N is a type. Another simple and useful rule is

$$\frac{A\ \text{type}}{x : A\ [x : A]}\ \text{assumption}$$

which says that if A has been shown to be a type then we can introduce the variable x as having that type, by assumption. Note how this assumption is recorded in the list after the judgement.

Other rules involving N are

$$\frac{a : N}{\text{succ}(a) : N}\ \text{N-intro-succ} \qquad \frac{}{0 : N}\ \text{N-intro-0}$$

$$\frac{n : N \quad d : C(0) \quad e(x,y) : C(\text{succ}(x))\ [x : N,\ y : C(x)]}{\text{rec}(n,d,e) : C(n)}\ \text{N-elim}$$

Taking the first two, we can see that derivations like the following are easily constructed

$$\frac{\dfrac{}{0 : N}}{\text{succ}(0) : N} \qquad \frac{\dfrac{\dfrac{}{0 : N}}{\text{succ}(0) : N}}{\text{succ}(\text{succ}(0)) : N}$$

Taking the last rule, we can construct the derivation

$$\frac{0 : N \qquad \dfrac{}{0 : N} \qquad \dfrac{\dfrac{N\ \text{type}}{x : N\ [x : N]}}{\text{succ}(x) : N\ [x : N]}}{\text{rec}(0,0,(x,y)\text{succ}(y)) : N}$$

(Here C is the abstraction (a)N)

This last rule, N-elim, formalizes induction over the natural numbers within CTT. The first premise introduces an arbitrary n in N. The second premise says that we must have some proof, d, of some arbitrary proposition C(0), i.e. some proposition possibly involving 0. The third says that we have some proof e(x,y) of the proposition C(succ(x)), i.e. some proposition possibly involving succ(x) in the same way as 0 was involved in the previous premise, where we can assume that x is in N and that y is a proof that C(x). This third premise, then, is what we would informally call the inductive step and the assumption introducing y is the inductive hypothesis. Given all of these we can conclude that C(n), i.e. the proposition introduced in the premises is true of the arbitrary element n, is proved (with a rather complicated looking proof).

So far, we have nothing which looks like computation. A rule involving N that does look computational (and which explains the proof in the N-elim rule) is

$$\frac{d : C(0) \qquad e(x,y) : C(succ(x)) \ [x : N,\ y : C(x)]}{rec(0,d,e) = d : C(0)} \text{ N-comp-0}$$

and another is

$$\frac{n : N \qquad d : C(0) \qquad e(x,y) : C(succ(x)) \ [x : N,\ y : C(x)]}{rec(succ(n),d,e) = e(n,rec(n,d,e)) : C(n)} \text{ N-comp-succ}$$

The reader can check that, using these rules, we can derive

rec(0,0,(x,y)succ(y)) = 0 : N,

rec(succ(0),0,(x,y)succ(y)) = succ(0) : N

and

rec(succ(0),succ(0),(x,y)succ(y)) = succ(succ(0)) : N.

In fact, the rules above go towards making sure that the type N behaves just like the natural numbers, so 'succ' forms successors, 0 acts as zero and 'rec(a,b,(x,y)succ(y))' acts like a + b whenever a and b are elements of N.

3. Deriving a simple numerical algorithm

The example used in this section is taken from [NPS90] and was treated in a similar manner in [Ree94], where a fuller explanation and presentation of the derivation, using the calculator MacPICT (or "PICTCalc" as it is now called) can be found.

First we need some definitions:

plus ≡ (a,b)rec(b,a,(x,y)succ(y))

and

mult ≡ (a,b)rec(a,0,(x,y)plus(b,y))

and

$1 \equiv succ(0)$

and

$2 \equiv succ(1)$

What we are going to do is construct an object in

$$\prod(N,(x1)\sum(N,(x2)(I(N,x1,mult(x2,2)) \vee I(N,x1,succ(mult(x2,2))))))$$

This says that for all x1 in N, there is an x2 in N such that either x1 = x2*2 or x1 = (x2*2)+1.

For convenience we assume that

$$d : \prod(N,(x1)\prod(N,(x2)(I(N,x1,succ(mult(x2,2)))\rightarrow I(N,succ(x1),mult(succ(x2),2)))))$$

which is to say that d is a proof that, for any x1 and x2 in N, if x1 = (x2 * 2) + 1 then x1 + 1 = (x2 + 1) * 2 which is clearly a simple fact of arithmetic.

If we follow through the derivation (which we won't do here, see [Ree94]) then the object we construct is

$$\lambda((x1)rec(x1,(0,inl(eq)),\ (x2,x3)split(x3,(x4,x5)when(x5,(x6)(x4,inr(eq)),\ (x6)(succ(x4),inl(d(x2,x4,x6)))))))$$

This is a function which given an x1 in N returns a pair consisting of the whole number part of x1/2 together with a proof that the answer is correct. For example, if this is applied to 0 then it immediately simplifies to (0,inl(eq)), i.e. the whole number part of 0/2 is 0. If it is applied to succ(0) then it simplifies to

$$split((0,inl(eq)),(x4,x5)when(x5,\ (x6)(x4,inr(eq)),(x6)\ (succ(x4),inl(d(0,x4,x6)))))$$

and then to

$$when(inl(eq),(x6)(0,inr(eq)),(x6)(succ(0),inl(d(0,0,x6))))$$

and finally to (0,inr(eq)), i.e. the whole number part of 1/2 is 0. If it is applied to succ(succ(0)) then it simplifies to

$$split((0,inr(eq)),\ (x4,x5)when(x5,(x6)(x4,inr(eq)),(x6)\ (succ(x4),inl(d(0,x4,x6)))))$$

and then to

when(inr(eq),(x6)(0,inr(eq)),(x6)(succ(0),inl(d(0,0,x6))))

and finally to

(succ(0),inl(d(0,0,eq)))

i.e. the whole number part of 2/2 is 1.

This is not what we would normally think of as the "div2" function since we do not usually expect to have proof information mixed in with the computational information (though in CTT of course these are the same things - it is our interpretation outside of the theory that makes this distinction between proof objects and computational objects). We consider this point further in the next section.

4. Transforming the specification

The form of the specification above was

$$\prod(N,(x1)\sum(N,(x2)e(x1,x2)))$$

so that we got a function which, for any number x1, gives a pair consisting of the number x1/2 and a proof that this was correct. That is, we get something like

0 -----> (0, a proof that 0/2 = 0)
1 -----> (0, a proof that 1/2 = 0)
2 -----> (1, a proof that 2/2 = 1)

and so on.

However, it is more usual to expect the specification to yield a pair consisting of a function, which works for all numbers, and a proof that the function is correct when we ask for a solution to the 'div2' problem. That is, we get

(f, λn.proof that n/2 = f(n))

In order to get a more usual object we need to change the specification into

$$\Sigma(N \rightarrow N,(f)\prod(N,(x1)(I(N,x1,mult(f(x1),2)) \vee I(N,x1,succ(mult(f(x1),2))))))$$

An object in this type is a pair of the form (F,P) where F is a function in $N \rightarrow N$ which returns the whole-number result of its argument divided by 2 and P is a proof that F has that property, i.e. a proof that it meets its specification. This is clearly the more usual way of specifying a function.

Now we turn to a seemingly unrelated (and surprising) problem, that of proving the axiom of choice. It turns out that the axiom of choice is neither independent of nor inconsistent with CTT - it is in fact a theorem of CTT. This has to do with particular properties of the formalisation we are considering, and Martin-Löf himself seems to have been unsure whether this property was desirable or not. We shall not worry about such things here, and simply carry on.

The axiom of choice (in one of its many equivalent forms) says that if given any family of sets we can collect together into a set one element from each member of the family such that some condition holds then there is a function to do it. (This is clearly not interesting if the family is finite - it becomes interesting when the family is infinite.)

We can formalize this statement by using A as the set which indexes the family, B(x) as a member of the family whenever x is from A, and C is the condition which the element from B(x) satisfies.

So, we have

$$\Pi(A,(a)\Sigma(B(a),(b)C(a,b))) \rightarrow \Sigma(\Pi(A,B),(f)\Pi(A,(a)C(a,f(a))))$$

It is not too hard to show that

$$\lambda((x1)\ (\lambda((x2)fst(x1(x2))),\lambda((x2)snd(x1(x2)))))$$

is an element of this type, so we have proved the axiom of choice. Further, and more interestingly for us, consider the antecedent to the axiom of choice and compare it with the specification for the div2 problem. Note that they match, with A as N, B as (x)N and C as $(I(N,x1,mult(x2,2)) \vee I(N,x1,succ(mult(x2,2))))$. So, we would expect that applying the proof of the axiom of choice to the element in the div2 specification would construct for us an element in the corresponding consequent of the axiom of choice, i.e. an element in

$$\Sigma(N \rightarrow N,(f)\Pi(N,(a)(I(N,a,mult(f(a),2)) \vee I(N,a,succ(mult(f(a),2))))$$

(using the abbreviation Π(N,(x)N) is N → N).

We can check that this is so:

applying the proof of the axiom of choice to the 'div2' object gives the pair

(λ((x2)fst(rec(x2,(0,inl(eq)),(x2,x3)split(x3,(x4,x5)when(x5,(x6)(x4,inr(eq)),(x6)
(succ(x4),inl(d(x2,x4,x6))))))),
λ((x2)snd(rec(x2,(0,inl(eq)),(x2,x3)split(x3,(x4,x5)when(x5,(x6)(x4,inr(eq)),(x6)
(succ(x4),inl(d(x2,x4,x6)))))))

Moving the fst and snd in as far as possible gives

(λ((x2)rec(x2,0,(x2,x3)split(x3,(x4,x5)when(x5,(x6)x4,(x6)succ(x4))))),
λ((x2)rec(x2,inl(eq),(x2,x3)split(x3,(x4,x5)when(x5,(x6)inr(eq),(x6)inl(d(x2,x4,x6)))))))

The first element is

λ((x2)rec(x2,0, (x2,x3)split(x3,(x4,x5)when(x5,(x6)x4,(x6)succ(x4)))))

which is a function in N → N which returns the whole number part of x2/2

The second element is

λ((x2)rec(x2,inl(eq),(x2,x3)split(x3,(x4,x5)when(x5,(x6)inr(eq),(x6) inl(d(x2,x4,x6))))))

which is a function which given x2 returns a proof that the function in the first part gives the right answer.

5. On using all available information

This next example is considered informally and says something rather general about algorithm specification and derivation - something which is not specific to CTT. Consider the function called 'lambo' (and invented by Lambeck and Moser). Given any function f : N → N such that f is unbounded we have

lambo f n = the least m such that f(m) ≥ n

lambo f n is easily computed by

```
i:=0;
while f(i) < n do
      i:=i+1
deliver i
```

However, the interesting question asks whether we can specify this function lambo in CTT. The problem seems to be that, since f is unbounded, lambo must necessarily embody some unbounded search, which CTT cannot do.

We start by asking for the type of lambo. The standard Milner typing would be

$$\text{lambo} : (N \to N) \to (N \to N)$$

However, with the far more expressive CTT we can do much better. In particular, we must not forget that the first argument to lambo must be an unbounded function. To say that this first argument, f, must be unbounded is to say that for all n in N, there is an m in N such that $f(m) \geq n$. In CTT this is

$$\Pi(N,(n)\Sigma(N,(m)(f(m) \geq n)))$$

where we assume that we have $\geq$ such that $a \geq b$ is inhabited iff a is greater than b.

Given this we can write a specification for lambo

$$\Pi(\Sigma(N \to N, (f)\Pi(N,(n)\Sigma(N,(m)(f(m) \geq n)))),$$
$$(g)\Pi(N, (p)\Sigma(N, (q)(\text{fst } g\ q \geq p \wedge \Pi(N, (r)(\text{fst } g\ r \geq p \ \to \ q \leq r))))))$$

So, note that lambo takes as its first argument a pair: the first element of the pair is the function f of type $N \to N$ which is unbounded and the second element, when applied to any number n, returns a pair consisting of a number m and a proof that $f(m) \geq n$.

This greater care about the type of lambo, then, has given us some more information which allows us to compute lambo using a bounded search. The fact that f is unbounded allows us to calculate the upper bound needed for our search. The computation for lambo can now be written as

```
lambo (f,h) n  =  for i:=0 to fst h(n) do
                        if f(i) ≥ n then return i
```

and so lambo could be derived within CTT.

6. Conclusion

CTT provides us with a single language (syntax, semantics and proof theory) within which we can specify and implement programs. Of course, this still leaves us with the problem of knowing whether or not the specification is correct. However, this will always be a problem until someone invents a formal system in which people will naturally think and wish to express and solve their computational problems and it which it will be as obvious as it currently is, when such 'requirements' are expressed in a natural language like English, that someone has asked for exactly what they want.

Given that we work in a single language, we do at least not have the problem of taking a specification language and an implementation language and trying to 'join' then together via some logic.

The logic that the proof rules express is constructive, which is to say that whenever we prove an existential statement we always construct the object whose existence is proved and whenever we prove a disjunction we always know which of the disjuncts holds. It is hard to see how a logic that deals with computation could be otherwise, without some elaborate apparatus to sort out the constructive from the non-constructive parts of the theorems proved.

One mode of attack on such an idea as CTT is to express incredulity that someone would ever expect to work completely within a formal system. Of course, such attacks are empty - no one does ever work completely within a formal system. Just like the formal notations (i.e. notations that look more mathematical than natural, e.g. Z or VDM, and whose semantics and proof theory are usually left informal or at least not used explicitly within the specification process) a formal system like CTT provides a means for being precise and expressing facts which might have been arrived at, initially, by any means whatsoever. What the proof theory in CTT gives us, in addition, is the means of checking these facts within the system itself. Again, the checking might initially be done informally and then expressed within CTT in order to both feed into the implementation process and also to check the checking itself.

The syntax of CTT is not attractive or useable. The first of these points is highly subjective and the second is untrue. However, the particular syntax used is just a convention and is not the most important part of the system. It certainly reflects the mathematical and logical background of its inventor, but could easily be changed.

As with any other formal system which deals with large formal objects (e.g. any programming language, specification language, proof checker, theorem prover) we run into problems with CTT as we become more ambitious. The more mundane problems of editing, keeping up-to-

date versions of, documenting and generally interacting with large formal objects are probably more important (and in some ways harder to solve) than the mathematical or logical problems that CTT and its like deal with.

Solving these sorts of large-scale problems requires experience and skills from many different areas within Computer Science - Software Engineering, HCI and Logic are the obvious ones. Finding people with these skills (or putting together groups with these skills) is hard, given the specialization that we are all forced to undergo as part of our training. Overcoming the limitations of our specialization is the real problem.

References

[Bee85] Beeson, M. J., "Foundations of Constructive Mathematics", Springer-Verlag, 1985.

[Dum77] Dummett, M., "Elements of Intuitionism", Oxford University Press, 1977.

[M-L84] Martin-Löf, P., "Intuitionistic Type Theory", Bibliopolis, 1984.

[NPS90] Nordström, B., Petersson, K. and Smith, J. "Programming in Martin-Löf's Type Theory: An Introduction", Oxford, 1990.

[Ree91] Reeves, S., "Programming as Constructive mathematics", in "Mathematical Structures for Software Engineering", ed. De Neumann, Simpson and Slater, Oxford, 1991.

[Ree94] Reeves, S., "Computer support for students' work in a formal system: MacPICT" to appear in the *International Journal on Mathematical Education in Science and Technology.*

[Tho91] Thompson, S., "Type Theory and Functional Programming", Addison-Wesley, 1991.

Program Derivation in the Refinement Calculus: An Introduction

Lindsay Groves

Department of Computer Science
Victoria University of Wellington
Wellington, New Zealand

Abstract

The refinement calculus is a formal calculus for deriving imperative programs from logical specifications, which formalises program construction by stepwise refinement. This paper introduces the basic concepts underlying the refinement calculus: a wide-spectrum language including non-executable constructs for expressing specifications, a refinement ordering on specifications/programs in this language, and a set of rules embodying laws about certain refinements that are always valid. The paper also shows how data refinement is formalised within the refinement calculus, presents some important theoretical results, and illustrates the techniques by deriving an algorithm to find the minimum and maximum elements of a set, using only $3n/2$ comparisons, and then data refining this to an array representation.

1 Introduction

The refinement calculus is a calculus for deriving imperative programs from logical specifications. Programs are derived in a wide-spectrum language including both executable and non-executable constructs. A derivation typically begins with a non-executable specification, and progresses via a number of intermediate forms combining both executable and non-executable constructs, ending up with a program containing only executable constructs.

In this way, the refinement calculus can be seen as a formalisation of stepwise refinement [30]. At each step in the derivation, some decision is made which takes the program nearer to an executable form, possibly giving rise to certain proof obligations. These proof obligations can be discharged as they arise, so that the program and proof are developed hand-in-hand, as advocated by Dijkstra [8], rather than the proof being attempted once the program has been completed. In fact, the refinement calculus doesn't have to be used in this way, and can also be used effectively to verify existing programs.

The refinement calculus has its origins in the early work of Ralph Back [1, 2], which was not widely known at the time, and later work by Carroll Morgan and others [17, 24, 25], and Joe Morris [26]. It can be seen as an extension of Dijkstra's program calculus [9, 10, 12], in which the partial programs corresponding to steps in the derivation, and the relationships between them, are made explicit. The refinement calculus is also related to formal development methods used with VDM [16] and Z [31].

As well as formalising stepwise refinement, the refinement calculus provides a framework within which a number of aspects of software development can be investigated. For example, data refinement, which was first formalised by Hoare [14] and explored within VDM [15], has flourished in the context of the refinement calculus [5, 18, 19, 23, 11, 27, 7, 32]. Furthermore,

the refinement calculus allows simpler proof obligations than are found, for example, with VDM and Z, and admits a more "calculational" style of development. The refinement calculus lends itself well to mechanical support, and the development of interactive refinement tools is now an active field. The refinement calculus also opens up new avenues for theoretical investigation, such as the lattice theoretic properties investigated by Back and von Wright [4].

In this paper we give a brief introduction to the refinement calculus. We begin by introducing a wide-spectrum language, along with its semantics, and discuss healthiness conditions. We then introduce the refinement relation which relates successive steps in a derivation, and explore some of its properties. Next, we introduce the idea of refinement rules, and illustrate these ideas by presenting a sample derivation. We then discuss data refinement briefly, and end with a few concluding remarks.

2 A Wide-Spectrum Language

The wide-spectrum language consists of an ordinary programming language, augmented with constructs for expressing specifications, which are (in general) not executable. The executable subset of the language is usually a variant of Dijkstra's guarded commands language, with semantics based on weakest precondition predicate transformers [9]. We refer to all constructs in this language as *programs*, though we sometimes call them *specification* when we expect them to be non-executable; executable programs are called *code*.

For any program S and predicate R, $wp(S,\ R)$ is the *weakest precondition* for S to establish R, i.e. the weakest condition that must hold in the initial state to ensure that S will terminate in a state satisfying R. A program which establish *true* always terminates (i.e. can terminate in any state); no program can establish *false* (i.e. terminate in no state), except by magic (see Section 2.2). If $wp(S,\ R) = \mathit{false}$, then S cannot be guaranteed to terminate in a state satisfying R, so for any initial state, S may do anything, including not terminate. Note that *wp* cannot distinguish between a program which always fails to terminate and one that only sometimes fails to terminate.

2.1 Executable constructs

The executable sublanguage we use is essentially Dijkstra's guarded commands language, with the addition of (untyped) local variable declarations. The weakest precondition semantics for these constructs are given in Figure 1.

There are three primitive statements: **skip** (do nothing), multiple assignment $\tilde{x} := \tilde{e}$ (where $\tilde{x}$ is a list of distinct variables, and $\tilde{e}$ is a list of expressions with the same length as $\tilde{x}$), and **abort** (do anything, including not terminate — usually to be avoided!). The composite statements are: sequential composition, alternation (**if**), repetition (**do**), and block with local variables.

As in Dijkstra's language, the alternation and repetition statements are nondeterministic. The statement[1] **if** $[]_i\ B_i \to S_i$ **fi** is executed by choosing any true guard B_j and executing the corresponding statement S_j. The choice is nondeterministic if more than one guard is true; the statement is equivalent to **abort** if no guard is true. The **do** statement is similar, except that after executing S_j it returns to choose another guard, and exits if no guard is true.

[1] **if** $[]_i\ B_i \to S_i$ **fi** is an abbreviation for **if** $B_1 \to S_1 [] \cdots [] B_n \to S_n$ **fi**.

$$wp(\textbf{skip},\ R) \ \widehat{=}\ R$$

$$wp(\textbf{abort},\ R) \ \widehat{=}\ \mathit{false}$$

$$wp(\tilde{x} := \tilde{e},\ R) \ \widehat{=}\ R_{[\tilde{x} \setminus \tilde{e}]}$$

where $R_{[\tilde{x} \setminus \tilde{e}]}$ is R with all free xs replaced by the corresponding es

$$wp(S_1\ ;\ S_2,\ R) \ \widehat{=}\ wp(S_1,\ wp(S_2,\ R))$$

$$wp(\textbf{if}\ \ [\!]_i\ B_i \rightarrow S_i\ \textbf{fi},\ R) \ \widehat{=}\ (\bigvee_i B_i) \wedge \bigwedge_i (B_i \Rightarrow wp(S_i,\ R))$$

$$wp(\textbf{do}\ \ [\!]_i\ B_i \rightarrow S_i\ \textbf{od},\ R) \ \widehat{=}\ \mu X \bullet (\bigwedge_i (B_i \Rightarrow wp(S_i,\ X))) \wedge\ (\bigvee_i B_i \vee R)$$

$$wp([\![\ \textbf{var}\ \ \tilde{x} \bullet S]\!],\ R) \ \widehat{=}\ \forall \tilde{x} \bullet wp(S,\ R)$$

Figure 1: *wp* definitions for executable constructs

The semantics of a block simply says that the meaning of the body of the block does not depend on the initial values of the new variables. This simple language can be extended to include constructs such as procedures, parameters, recursion and modules (e.g. see [20], [25]).

We are deliberately vague about what data types and operations are supported in the executable sublanguage. We generally assume that integers (or naturals), Booleans, and arrays of integers or Booleans, and the usual operations upon them, are available, but make other assumptions as appropriate. For example, when deriving an abstract algorithm, we may assume that certain abstract types (e.g. sets) and suitable operations upon them (e.g. union, intersection, etc.) are available, and perhaps additional primitive statements. We might then remove these in a subsequent data refinement (see Section 6). Defining functions and data types can be seen as extending the base language in which the program is to be expressed, rather than as part of constructing that program.

The definitions given in Figure 1 assume that expressions (and guards) are always defined, i.e. all variables have values and all functions used are total. In the absence of this assumption, we would need to introduce additional conjuncts to ensure that expressions and guards are defined. To ensure that the resulting predicates are well-defined when these conjuncts are false, we would also need to introduce conditional connectives, such as **cand** [10, 12], or use a three-valued logic (e.g. [6]. Assuming that expressions are always defined simplifies the semantics, and also simplifies refinement rules and the resulting proof obligations. This assumption is quite plausible when discussing specifications, but can lead to some odd anomalies, and does not capture the semantics of most real programming languages, where evaluating a division by zero or an array access with an out-of-bounds index will cause the program to terminate abnormally.

2.2 Non-executable constructs

The wide-spectrum language also provides a number of constructs which are useful for expressing specifications, but are (in general) non-executable. A variety of such constructs are used in different versions of the refinement calculus. We show some of the variations below, but generally we use the version described by Morgan [22]. The weakest preconditions for these constructs are given in Figure 2.

$$
\begin{array}{l}
wp(\tilde{w}{:}\left[\, P \;/\; Q \,\right],\ R) \ \widehat{=}\ P \wedge (\forall \tilde{w} \bullet Q \Rightarrow R) \\
wp(\tilde{w}{:}\left[\, Q \,\right],\ R) \ \widehat{=}\ ((\forall \tilde{w} \bullet Q \Rightarrow R) \\
wp(\mathbf{magic},\ R) \ \widehat{=}\ \mathit{true} \\
wp(\{P\},\ R) \ \widehat{=}\ P \wedge R \\
wp([P],\ R) \ \widehat{=}\ P \Rightarrow R \\
wp(\llbracket\ \mathbf{con}\ \ \tilde{x} \bullet S \rrbracket,\ R) \ \widehat{=}\ \exists \tilde{x} \bullet wp(S,\ R)
\end{array}
$$

Figure 2: *wp* definitions for non-executable constructs

- Specification statement: $\tilde{w}{:}\left[\, P \;/\; Q \,\right]$
  Establish Q, changing only variables in $\tilde{w}$, provided P holds initially. The precondition, P, and postcondition, Q, are predicates over state variables; the frame, $\tilde{w}$, is a list of distinct variables.
- Nondeterministic assignment: $\tilde{w}{:}\left[\, Q \,\right]$
  Assign values to variables in $\tilde{w}$ so as to establish Q, if such values exist[2].
- Magic: **magic**
  Do magic — makes any postcondition true!
- Assertion: $\{P\}$
  Assert that P holds; abort if it doesn't.
- Coercion: $[P]$
  Force P to be true, doing magic if necessary.
- Logical constant: $\llbracket\ \mathbf{con}\ \ \tilde{x} \bullet S \rrbracket$
  Execute S with values for variables in $\tilde{x}$ chosen so to make any preconditions, including implicit ones, in S true (if possible).

Expressions in assignment statements and guards may also be non-executable because they involve data types and/or operators which are not supported by the executable sublanguage. For example, guards may contain quantifiers. As with the non-executable constructs above, these must be removed in order to obtain an implementation.

These constructs allow us to write specifications that cannot possibly be satisfied, for example, the specification $x{:}\left[\, \mathit{true} \;/\; \mathit{false} \,\right]$, which is equivalent to **magic**. Such specifications are said to be miraculous or infeasible, where feasibility is defined as follows:

Definition A program S is *feasible* if $wp(S, \mathit{false}) = \mathit{false}$.

All of the executable constructs (i.e. code) are feasible, but some of the non-executable ones are not. Allowing infeasible statements, such as **magic**, is convenient in developing the theory of

[2]The weakest precondition for $\tilde{w}{:}\left[\, Q \,\right]$ is sometimes given as $(\exists \tilde{w} \bullet Q) \wedge (\forall \tilde{w} \bullet Q \Rightarrow R)$, which makes $\tilde{w}{:}\left[\, Q \,\right]$ abort when there are no values for $\tilde{w}$ which satisfy Q, whereas the definition given here makes it miraculous.

program refinement. These statements have some interesting properties, and can also be useful as an intermediate step in program derivations [18]. Part of the task of program derivation is to show that the specification is feasible, i.e. that it does not require magic.

It is often necessary for the postcondition of a specification statement to refer to the initial value of a variable as well as its final value. This can be expressed using the convention that zero-subscripted variables refer to initial values. Thus, $x{:}\left[\, true \;/\; x > x_0 \,\right]$ specifies a program that will increase the value of x by an arbitrary amount. This convention can be formalised by the use of logical constants; the above specification is equivalent to $[\![\ \mathbf{con}\ \ c \bullet x{:}\left[\, x = c \;/\; x > c \,\right]\ ]\!]$. Our weakest precondition definitions all ignore the possibility of zero-subscripted variables, since they can always be removed in this way.

2.3 Other extensions

Some versions of the refinement calculus also split Dijkstra's **if** and **do** statements into simpler and more general constructs by treating $\rightarrow$, $[]$, **if–fi** and **do–od** as separate constructors. We may also add the angelic choice operator $\diamond$. The weakest preconditions for these constructs are given in Figure 3.

$$
\begin{array}{l}
wp(B \rightarrow S,\ R) \ \widehat{=}\ B \Rightarrow wp(S,\ R) \\
wp(S \,[]\, T,\ R) \ \widehat{=}\ wp(S,\ R) \wedge wp(T,\ R) \\
wp(S \diamond T,\ R) \ \widehat{=}\ wp(S,\ R) \vee wp(T,\ R) \\
wp(\mathbf{if}\ \ S\ \mathbf{fi},\ R) \ \widehat{=}\ grd(S) \wedge wp(S,\ R) \\
wp(\mathbf{do}\ S\ \mathbf{od},\ R) \ \widehat{=}\ \mu X \bullet wp(S,\ X) \wedge (grd(S) \vee R) \\
\quad \text{where } grd(S) \ \widehat{=}\ \neg wp(S,\ false)
\end{array}
$$

Figure 3: *wp* definitions for generalized program constructors

The construct $B \rightarrow S$ is sometimes called a "naked guarded command". It says behave like S when B holds, otherwise do magic. Since it doesn't always require magic, this is sometimes also called a "minor miracle".

The construct $S \,[]\, T$ says choose either S or T to execute; either choice must then establish the required postcondition. Because we must be prepared for the worst, this is called "demonic choice".

By contrast, $S \diamond T$ says choose either S or T to execute, but only one of them needs to establish the required postcondition and the right one will be chosen. This is called "angelic choice"[3].

The effect of wrapping **if–fi** around a statement S is a statement which will execute S if it is feasible, and abort if S is miraculous. Similarly, **do** S **od** is a statement which repeats S as long as it is feasible, stopping when S becomes miraculous.

[3] Angelic and demonic choice are rather like "don't know" and "don't care" nondeterminism, respectively, as discussed in the logic programming literature.

Breaking the language down into simpler constructors like this allows a number of very elegant lattice-theoretic properties to be obtained (see Section 3).

2.4 Healthiness conditions

Dijkstra [10] proposed a number of "healthiness conditions" that any computational mechanism should obey. These can be expressed in terms of the following properties[4]:

Strictness: S is *strict* if $wp(S,\ false) \Leftrightarrow false$.

Monotonicity: S is *monotonic* (with respect to $\Rightarrow$) if $Q \Rightarrow R$ implies $wp(S,\ Q) \Rightarrow wp(S,\ R)$.

Disjunctivity: S is *disjunctive* if $wp(S,\ Q \vee R) \Leftrightarrow wp(S,\ Q) \vee wp(S,\ R)$.

Conjunctivity: S is *conjunctive* if $wp(S,\ Q \wedge R) \Leftrightarrow wp(S,\ Q) \wedge wp(S,\ R)$.

Continuity: S is *continuous* if for any infinite chain of predicates $R_0, R_1, \cdots$, such that $R_i \Rightarrow R_{i+1}$, $wp(S,\ \bigvee_i R_i) \Leftrightarrow \bigvee_i wp(S,\ R_i)$.

Dijkstra's healthiness conditions state that any computational mechanism S must be strict (this is known as the "Law of the Excluded Miracle"), monotonic, conjunctive and continuous. A mechanism that is nondeterministic is not disjunctive, so Dijkstra used a weaker version of disjunctivity ($wp(S,\ Q) \vee wp(S,\ R) \Rightarrow wp(S,\ Q \vee R)$), which follows from monotonicity. Continuity was required in order to avoid unbounded nondeterminism; this has since been discarded by most authors.

The constructs added in the refinement calculus further challenge these healthiness conditions. For example, specification statements can be non-strict (since they can have *false* as their postconditiona, which is equivalent to **magic**), and logical constants are not conjunctive. This leaves monotonicity as the only healthiness condition intact. The executable constructs, however, are still strict and conjunctive.

3 The Refinement Relation

A key component of the refinement calculus is the refinement ordering, written $\sqsubseteq$, on programs. This relation is defined so that $S \sqsubseteq S'$, read "S is refined by S'", if a client who asks for S will be happy if provided with code satisfying S'.

We define "satisfies" in the usual way (following Dijkstra):

Definition A program S *satisfies* a specification with precondition P and postcondition R iff $P \Rightarrow wp(S,\ R)$.

This still applies when S is any construct in the wide-spectrum language. We now define $\sqsubseteq$ as follows:

Definition $S \sqsubseteq S'$ iff for any postcondition R, $wp(S,\ R) \Rightarrow wp(S',\ R)$.

It is easy to see that with this definition, S' satisfies any specification that S satisfies, and any implementation of S' is also an implementation of S.

[4] We write $\Phi \Leftrightarrow \Psi$ to mean that predicates Φ and Ψ are equivalent for all states, i.e. $(\forall \sigma \bullet \Phi\sigma \equiv \Psi\sigma)$, and $\Phi \Rightarrow \Psi$ to mean that Φ implies Ψ for all states, i.e. $(\forall \sigma \bullet \Phi\sigma \Rightarrow \Psi\sigma)$.

If $S \sqsubseteq S'$, then S' can differ from S in either (or both) of two ways: S' may terminate more often than S (have a weaker precondition), or S' may be less deterministic than S (have a stronger postcondition). Note that the richness of this relationship is a consequence of the class of programs being considered: if programs were always total, it would not be possible to weaken preconditions; if programs were always deterministic (i.e. functions), it would not be possible to strengthen postconditions.

To derive a program T from a specification S, we construct a sequence of programs $S_0, \cdots, S_n$, such that $S_0 = S$, $S_{i-1} \sqsubseteq S_i$ for $i = 1, \cdots, n$, and $S_n = T$ is executable (i.e. code).

In order for this stepwise approach to work, we must have the following:

Theorem Transitivity of $\sqsubseteq$. If $S \sqsubseteq S'$ and $S' \sqsubseteq S''$, then $S \sqsubseteq S''$.

This allows us to construct a program via a sequence of steps, each of which is a refinement of the previous one, and be assured that the final program is a refinement of the initial one.

In order to be able to refine components of a program individually, we also require the following:

Theorem Monotonicity of program constructors (with respect to $\sqsubseteq$). Let $\mathcal{F}[S]$ be a program with S as a component, and $\mathcal{F}[S']$ be the result of replacing the designated occurrence of S in $\mathcal{F}[S]$ by S'. If $S \sqsubseteq S'$, then $\mathcal{F}[S] \sqsubseteq \mathcal{F}[S']$. More specifically:

If $S \sqsubseteq S'$ and $T \sqsubseteq T'$ then:

$$
\begin{array}{rcl}
S \,;\, T & \sqsubseteq & S' \,;\, T' \\
\textbf{if}\ \ A \to S \ [\!]\ B \to T\ \textbf{fi} & \sqsubseteq & \textbf{if}\ \ A \to S' \ [\!]\ B \to T'\ \textbf{fi} \\
\textbf{do}\ A \to S \ [\!]\ B \to T\ \textbf{od} & \sqsubseteq & \textbf{do}\ A \to S' \ [\!]\ B \to T'\ \textbf{od} \\
[\![\ \textbf{var}\ \ \tilde{x} \bullet S]\!] & \sqsubseteq & [\![\ \textbf{var}\ \ \tilde{x} \bullet S']\!] \\
[\![\ \textbf{con}\ \ \tilde{x} \bullet S]\!] & \sqsubseteq & [\![\ \textbf{con}\ \ \tilde{x} \bullet S']\!]
\end{array}
$$

When treated as separate constructors, $\to$, $[\!]$ and $\diamond$ are monotonic; but **if** –**fi** and **do** –**od**, as defined above, are not — they must therefore be used with great care.

The refinement ordering has a number of other interesting properties. In particular, it forms a complete lattice with **abort** and **magic** as extreme elements, and $[\!]$ and $\diamond$ as the meet and join operations.

Theorem For any program S, **abort** $\sqsubseteq S \sqsubseteq$ **magic**.

Theorem For all programs S, T and U, if $S \sqsubseteq T$ and $S \sqsubseteq U$, then $S \sqsubseteq T \,[\!]\, U$.

Theorem For all programs S, T and U, if $S \sqsubseteq U$ and $T \sqsubseteq U$, then $S \diamond T \sqsubseteq U$.

Now we see why it is important in deriving programs that we find T such that $S \sqsubseteq T$ and T is feasible — otherwise we could just refine everything to **magic**!

We can also find analogues of many other algebraic and lattice theoretic concepts, such as duals, inverses and adjoints (see [4]).

4 Refinement rules

Another important feature of the refinement calculus is a collection of rules embodying laws showing that certain refinements are always valid. These allow us to derive programs without

having to reason directly about weakest preconditions (except where it is convenient to do so). We write refinement rules as inference rules with the proof obligations or applicability conditions above the line, and the inferred refinement below.

As a basis, we need a rule to turn a specification statement into each of the executable constructs in the language:

Introduce SKIP:

$$\frac{P \Rrightarrow Q}{\tilde{w}{:}\left[\, P \;/\; Q \,\right] \sqsubseteq \mathbf{skip}}$$

Introduce Assignment:

$$\frac{\begin{array}{c}\text{Every variable in } \tilde{x} \text{ occurs in } \tilde{w} \\ P \Rrightarrow Q_{[\tilde{x} \setminus \tilde{e}]}\end{array}}{\tilde{w}{:}\left[\, P \;/\; Q \,\right] \sqsubseteq \tilde{x} := \tilde{e}}$$

Split Specification:

$$\tilde{w}{:}\left[\, P \;/\; Q \,\right] \sqsubseteq \tilde{w}{:}\left[\, P \;/\; M \,\right] \,;\, \tilde{w}{:}\left[\, M \;/\; Q \,\right]$$

Introduce IF:

$$\frac{P \Rrightarrow \bigvee_i B_i}{\tilde{w}{:}\left[\, P \;/\; Q \,\right] \sqsubseteq \mathbf{if} \quad \llbracket_i\, B_i \to \tilde{w}{:}\left[\, P \wedge B_i \;/\; Q \,\right] \;\mathbf{fi}}$$

Introduce DO:

$$\frac{\begin{array}{c}P \Rrightarrow I \\ I \wedge \neg(\bigvee_i B_i) \Rrightarrow Q\end{array}}{\tilde{w}{:}\left[\, P \;/\; Q \,\right] \sqsubseteq \mathbf{do} \quad \llbracket_i\, B_i \to \tilde{w}{:}\left[\, I \wedge B_i \;/\; I \wedge 0 \le t < t_0 \,\right] \;\mathbf{od}}$$

where I is the loop invariant, t is the variant function, and t_0 is obtained from t by replacing each variable x in $\tilde{w}$ by x_0.

Introduce Local Variables:

$$\frac{\tilde{x} \text{ are fresh names}}{\tilde{w}{:}\left[\, P \;/\; Q \,\right] \sqsubseteq \left[\!\!\left[\; \mathbf{var}\;\; \tilde{x} \bullet \tilde{x}, \tilde{w}{:}\left[\, P \;/\; Q \,\right] \right]\!\!\right]}$$

We also have three rules for manipulating specification statements:

Contract Frame:

$$\frac{\text{All variables in } \tilde{w} \text{ are in } \tilde{w}}{\tilde{w}{:}\left[\, P \;/\; Q \,\right] \sqsubseteq \tilde{w}'{:}\left[\, P \;/\; Q \,\right]}$$

Weaken Precondition:

$$\frac{P \Rrightarrow P'}{\tilde{w}{:}\left[\, P \;/\; Q \,\right] \sqsubseteq \tilde{w}{:}\left[\, P' \;/\; Q \,\right]}$$

Strengthen Postcondition:

$$\frac{Q' \Rrightarrow Q}{\tilde{w}{:}\left[\, P \;/\; Q \,\right] \sqsubseteq \tilde{w}{:}\left[\, P \;/\; Q' \,\right]}$$

It is often useful to have a rule which splits a specification into two parts, where we know what the second part is and can used *wp* to calculate the intermediate assertion. For example, if the second component is an assignment, we have:

Following Assignment:

$$\tilde{w}: \left[P \;/\; Q \right] \sqsubseteq \tilde{w}: \left[P \;/\; Q_{[\tilde{x} \setminus \tilde{e}]} \right] ; \tilde{x} := \tilde{e}$$

We also need rules for removing other non-executable constructs. For example, the following rule allows logical constants to be removed:

Remove Logical Constant:

$$\frac{\text{No variable in } \tilde{x} \text{ is free in } S}{[\![\textbf{con} \;\; \tilde{x} \bullet S]\!] \sqsubseteq S}$$

Most of these rules are closely related to corresponding rules in Hoare's logic (or a total correctness version thereof), and to Dijkstra's *wp* rules and associated techniques. If functions are not assumed to be total, we need additional proof obligations in the rules for assignment, **if** and **do** , to ensure that expressions and guards are defined.

The *Weaken Precondition* and *Strengthen Postcondition* rules are only required when the new precondition (postcondition) is strictly weaker (stronger) — we allow equivalence preserving transformations without the explicit application of a rule.

Many variations of these rules are possible. For example, the laws given in [24] typically have fewer proof obligations, but apply to more restrictive forms of specification statement. For instance, their rule for *Introduce Assignment* (ignoring zero-subscripts) is:

$$\tilde{w}: \left[Q_{[\tilde{w} \setminus \tilde{e}]} \;/\; Q \right] \sqsubseteq \tilde{w} := \tilde{e}$$

With those rules, there is frequent need for *Weaken Precondition*, *Strengthen Postcondition* and *Contract Frame*. Morgan [22] also gives different versions of some laws according to whether zero-subscripted variables are allowed and where. We omit zero-subscripted variables as they can be handled using logical constants.

We can also define a large number of rules which perform various kinds of transformations (thus refinement encompasses program transformation). In a sense, such rules are strictly unnecessary, since any program that can be derived using them can also be derived without them. These rules do, however, become important in conjunction with data refinement (see Section 6). Morgan [22] lists many more rules — far more than anyone would want to remember! In performing derivations, we may wish to introduce new rules corresponding to particular derivation patterns that occur frequently, or to introduce new primitive statements that are used.

5 Example derivation

To illustrate the way programs are derived in the refinement calculus, we will derive an algorithm to find the minimum and maximum elements in a non-empty set, S. The algorithm we derive is one which requires $3n/2$ comparisons, rather than the $2n$ required by the "obvious" algorithm. To avoid a few messy details, we assume that S has an odd number of elements.

To simplify the notation, we will write $u \leq S \leq v$ to mean that u and v are the minimum and maximum, respectively, of S, i.e.

$$u \leq S \leq v \;\widehat{=}\; \{u, v\} \subseteq S \wedge (\forall x \in S \bullet u \leq x \wedge x \leq v)$$

The specification can now be written as a specification statement:

$$lo, hi: \left[odd(\,|S|\,) \;/\; lo \leq S \leq hi \right] \tag{1}$$

Since the problem involves sets, we will assume that certain operations on sets are available. In particular, we will freely use set union and difference[5]. We will also assume a statement, written $x :\in S$, which selects an arbitrary element from a set S and assigns it to x. This statement is equivalent to the nondeterministic assignment $x: \left[(x \in S) \right]$, which in turn is equivalent to $x: \left[S \neq \varnothing \;/\; x \in S \right]$, so a suitable refinement rule is:

Introduce Set Selection:

$$\frac{\begin{array}{c} x \text{ occurs in } w \\ P \Rrightarrow S \neq \varnothing \\ (\forall x \bullet x \in S \Rightarrow Q) \end{array}}{w: \left[P \;/\; Q \right] \sqsubseteq x :\in S}$$

We will clearly need a loop, so we first look for a loop invariant. We can obtain a loop invariant by weakening the postcondition [12], so that *lo* and *hi* are the minimum and maximum of some (non-empty) subset of S, i.e.

$$Inv \;\widehat{=}\; T \subseteq S \wedge lo \leq T \leq hi$$

Since we replaced S by T to obtain *Inv*, we will take $T \neq S$ as the loop guard, and $|S - T|$ as the variant function. As S cannot change, we will not include the precondition $odd(\,|S|\,)$ explicitly in the loop invariant or other assertions, but will assume it to hold globally and draw on it as required; this can be formalised by the use of invariants (see [21]).

We first introduce T as a local variable:

(1) $\sqsubseteq$ (*Introduce Local Variables*)
var $T \bullet$
$lo, hi, T: \left[odd(\,|S|\,) \;/\; lo \leq S \leq hi \right]$ (2)

Since the loop invariant must be established, we split (2) into a sequence, with *Inv* as the intermediate assertion:

(2) $\sqsubseteq$ (*Split Specification*)
$lo, hi, T: \left[odd(\,|S|\,) \;/\; Inv \right]$; (3)
$lo, hi, T: \left[Inv \;/\; lo \leq S \leq hi \right]$ (4)

We can easily establish *Inv* by taking T to be any singleton subset of S, and its single member as both minimum and maximum:

[5] We won't assume that set minimum and set maximum are available — that would defeat the purpose of the exercise!

(3) $\sqsubseteq$ (*Strengthen Postcondition*)

$$lo, hi, T: \left[odd(\,|S|\,) \;/\; \exists x \in S \bullet lo = hi = x \wedge T = \{x\} \right] \tag{5}$$

This step has a simple proof obligation:

$$(\exists x \in S \bullet lo = hi = x \wedge T = \{x\}) \Rrightarrow T \subseteq S \wedge lo \leq T \leq hi$$

In a similar manner, we can refine (5) to:

$$\begin{array}{l} |[\ \textbf{var}\ x \bullet \\ \quad x :\in S; \\ \quad lo, hi, T := x, x, \{x\} \\ ]| \end{array}$$

The proof obligation for this refinement reduces to $S \neq \varnothing$, which follows from the precondition, $odd(\,|S|\,)$.

We now turn our attention to (4), which we refine to a loop. As noted above, we take $T \neq S$ as the guard, and $|S - T|$ as the variant:

(4) $\sqsubseteq$ (*Introduce DO*)

$$\begin{array}{l} \textbf{do}\ T \neq S \rightarrow \\ \quad lo, hi, T: \left[Inv \wedge T \neq S \;/\; Inv \wedge 0 \leq |S - T| < |S - T_0| \right] \\ \textbf{od} \end{array} \tag{6}$$

The proof obligations for this refinement are also trivial:

$$\begin{array}{l} Inv \Rrightarrow Inv \\ Inv \wedge \neg(T \neq S) \Rrightarrow lo \leq S \leq hi \end{array}$$

Now, in order for the loop to "make progress", the variant must be reduced, i.e. the size of T must be increased. An obvious choice would be to increase the size of T by one, which would lead to the "obvious" algorithm referred to above. We will do something a little less obvious, however, and increase the size of T by two.

We simply need to select two distinct elements from $S - T$, and compare the smaller with lo and the larger with hi. This will require two local variables:

(6) $\sqsubseteq$ (*Introduce Local Variables*)

$$\begin{array}{l} \textbf{var}\ u, v \bullet \\ lo, hi, T, u, v: \left[Inv \wedge T \neq S \;/\; Inv \wedge 0 \leq |S - T| < |S - T_0| \right] \end{array} \tag{7}$$

We now split (7), so as to first select u and v so that $u < v$, adjust lo and hi if necessary, and then update T:

(7) $\sqsubseteq$ (*Split Specification* (twice) and *Contract Frame*)

$$u, v{:}\left[\, Inv \wedge T \neq S \;/\; Inv \wedge \{u, v\} \subseteq S - T \wedge u < v \,\right]; \tag{8}$$

$$lo, hi{:}\left[\, Inv \wedge \{u, v\} \subseteq S - T \wedge u < v \;/\; \begin{array}{c} T \subseteq S \wedge \{u, v\} \subseteq S - T \wedge \\ lo \leq T \cup \{u, v\} \leq hi \end{array} \,\right]; \tag{9}$$

$$T{:}\left[\, \begin{array}{c} T \subseteq S \wedge \{u, v\} \subseteq S - T \wedge \\ lo \leq T \cup \{u, v\} \leq hi \end{array} \;/\; Inv \wedge 0 \leq |S - T| < |S - T_0| \,\right] \tag{10}$$

We wish to refine (8) as follows, assuming binary operators **min** and **max**:

(8) $\sqsubseteq$ (*Split Specification*, *Introduce Set Selection* and *Introduce Assignment*)

$u :\in S - T;$
$v :\in S - T - \{v\};$
$u, v := u \;\mathbf{min}\; v, u \;\mathbf{max}\; v$

The proof obligation for this reduces to $T \subset S \Rrightarrow |S - T| \geq 2$. Unfortunately, we cannot prove this!

At this point we observe that, because of the way we have chosen to implement (6), we can strengthen the loop invariant, adding a further conjunct to indicate that the size of T is odd. From this it follows (since $|S|$ is odd) that $|S - T|$ is even, so $T \subset S \Rrightarrow |S - T| \geq 2$ holds.

Thus, we modify the definition of *Inv* to:

$$Inv \;\widehat{=}\; T \subseteq S \wedge odd(\,|T|\,) \wedge lo \leq T \leq hi$$

We can easily check that the previous steps are not affected by this change, and the outstanding proof obligation can be discharged. This kind of revision is quite common in program derivations, and points to the need for tool support [28, 13]; an alternative approach would be to defer the choice of *Inv* until we knew precisely what constraints it needed to satisfy (see [29]).

Next, we refine (9) using the **min** and **max** operators:

(9) $\sqsubseteq$ (*Introduce Assignment*)

$lo, hi := lo \;\mathbf{min}\; u, hi \;\mathbf{max}\; v$

This refinement has a simple proof obligation:

$$lo \leq T \leq hi \wedge u < v \Rrightarrow (lo \;\mathbf{min}\; u) \leq T \cup \{u, v\} \leq (hi \;\mathbf{max}\; v)$$

Finally, we refine (10) to an assignment:

(10) $\sqsubseteq$ (*Introduce Assignment*)

$T := T \cup \{u, v\}$

Again, the proof obligation is straightforward.

This completes the derivation. We have shown that (1) refines to:

$$
\begin{array}{l}
[\![\ \textbf{var}\ T \bullet \\
\quad [\![\ \textbf{var}\ x \bullet \\
\qquad x :\in S; \\
\qquad lo, hi, T := x, x, \{x\} \\
\quad]\!]; \\
\quad \textbf{do}\ T \neq S \rightarrow \\
\qquad [\![\ \textbf{var}\ u, v \bullet \\
\qquad\quad u :\in S - T; \\
\qquad\quad v :\in S - T - \{u\}; \\
\qquad\quad u, v := u\ \textbf{min}\ v, u\ \textbf{max}\ v; \\
\qquad\quad lo, hi := lo\ \textbf{min}\ u, hi\ \textbf{max}\ v; \\
\qquad\quad T := T \cup \{u, v\} \\
\qquad]\!] \\
\quad \textbf{od} \\
]\!]
\end{array}
$$

If **min** and **max** are not considered to be executable, the two statements involving these operators can be further refined to **if** statements. For example:

$$
\begin{array}{ll}
 & u, v := u\ \textbf{min}\ v, u\ \textbf{max}\ v \\
\sqsubseteq & \textbf{if}\ \ u < v \rightarrow \textbf{skip} \\
 & [\!]\ \ u > v \rightarrow u, v := v, u \\
 & \textbf{fi}
\end{array}
$$

Since we know that $u \neq v$, the two branches are mutually exclusive (and even if $u = v$ was possible, it wouldn't matter which branch was executed). Therefore, in a language with an **if–then–else** construct, this could be implemented using a single test; the assignment $lo, hi := lo\ \textbf{min}\ u, hi\ \textbf{max}\ v$, however, would still require two tests. Thus, the algorithm only requires $3n/2$ comparisons.

6 Data refinement

An important aspect of program derivation is the replacement of "abstract" variables by "concrete" representations, and expressions involving abstract variables by equivalents involving the concrete ones. This process, known as data refinement, can be formalised in the refinement calculus in a number of ways [7]. The version we present here is that of Morgan and Gardiner [23] and Morris [27]. More general accounts can be found in [11] and [32].

When abstract variables are replaced by concrete ones, we must indicate in what sense the resulting program is "equivalent" to the original. This is done by specifying a relationship between concrete and abstract variables, known as a *coupling invariant* or *abstraction invariant*, showing how the concrete variables are used to represent the abstract ones. In early work on data refinement, this relationship was assumed to be a function from concrete values to abstract

(called the "retrieve function" in VDM literature [16]), and was also required to be "adequate" (i.e. onto); both requirements are relaxed in the current treatment.

We write $S \preceq_{\tilde{a},I,\tilde{c}} S'$ to mean that S' is obtained by data refining S, replacing abstract variables $\tilde{a}$ by concrete variables $\tilde{c}$, with coupling invariant I. This is defined formally as follows:

Definition $S \preceq_{\tilde{a},I,\tilde{c}} S'$ iff $(\exists \tilde{a} \bullet I \wedge wp(S,\ R)) \Rrightarrow wp(S',\ (\exists \tilde{a} \bullet I \wedge R))$, for all postconditions R not containing any variables in $\tilde{c}$.

Informally, this can be understood as saying that if an abstract state $\mathcal{A}'$ could be reached by executing S, starting in an abstract state $\mathcal{A}$ corresponding to concrete state $\mathcal{C}$, then $\mathcal{A}'$ corresponds to some state $\mathcal{C}'$ that could be reached by executing S' starting in $\mathcal{C}$.

With this definition, we get the following important relationship between data refinement and procedural refinement:

Theorem If $S \preceq_{\tilde{a},I,\tilde{c}} S'$ then $[\![\ \mathbf{var}\ \tilde{a} \bullet S]\!] \sqsubseteq [\![\ \mathbf{var}\ \tilde{c} \bullet S']\!]$

This shows that we can refine a block by replacing the variables it declares by new ones and data refining the body using a suitable coupling invariant.

It is easy to see that any statement which does not contain any abstract variables data refines itself. In particular, we get:

$$\begin{aligned}
\mathbf{abort} &\preceq_{\tilde{a},I,\tilde{c}} \mathbf{abort} \\
\mathbf{magic} &\preceq_{\tilde{a},I,\tilde{c}} \mathbf{magic} \\
\mathbf{skip} &\preceq_{\tilde{a},I,\tilde{c}} \mathbf{skip} \\
\tilde{x} := \tilde{e} &\preceq_{\tilde{a},I,\tilde{c}} \tilde{x} := \tilde{e}, \quad \text{if } \tilde{x} \text{ and } \tilde{e} \text{ contain no } a\text{'s}
\end{aligned}$$

We also get a number of laws showing how data refinement distributes through various constructs:

If $S \preceq_{\tilde{a},I,\tilde{c}} S'$, $S_i \preceq_{\tilde{a},I,\tilde{c}} S_i'$ for $i = 1, \cdots$, and $\tilde{x}$ contains no a's, then:

$$\begin{aligned}
S_1;\ S_2 &\preceq_{\tilde{a},I,\tilde{c}} S_1';\ S_2' \\
[\![\ \mathbf{var}\ \tilde{x} \bullet S]\!] &\preceq_{\tilde{a},I,\tilde{c}} [\![\ \mathbf{var}\ \tilde{x} \bullet S']\!] \\
[\![\ \mathbf{con}\ \tilde{x} \bullet S]\!] &\preceq_{\tilde{a},I,\tilde{c}} [\![\ \mathbf{con}\ \tilde{x} \bullet S']\!]
\end{aligned}$$

We data refine a specification statement as follows:

$$\tilde{a}, \tilde{x}: \left[P \ /\ Q \right] \preceq_{\tilde{a},I,\tilde{c}} \tilde{c}, \tilde{x}: \left[(\exists \tilde{a} \bullet I \wedge P) \ /\ (\exists \tilde{a} \bullet I \wedge Q) \right]$$

When the coupling invariant is functional from concrete to abstract, i.e. I can be written as $\tilde{a} = F(\tilde{x}) \wedge H(\tilde{c})$, where H is an invariant on $\tilde{c}$ (sometimes called the "representation invariant"), this simplifies to:

$$\tilde{a}, \tilde{x}: \left[P \ /\ Q \right] \preceq_{\tilde{a},I,\tilde{c}} \tilde{c}, \tilde{x}: \left[P[\tilde{a} := F(\tilde{c})] \wedge H(\tilde{c}) \ /\ Q[\tilde{a} := F(\tilde{c})] \wedge H(\tilde{c}) \right]$$

Assignment statements, assertions and coercions can be treated as special cases of specification statements.

To data refine **if** and **do** statements, we need to replace the guards containing a's by ones involving c's, and data refine the component statements. A guard B_i is replaced by the new guard $(\forall \tilde{a} \bullet I \Rightarrow B_i)$. We must also ensure that whenever one of the abstract guards would have been true, one of the concrete guards is true. Thus, we get:

If $S_i \preceq_{\tilde{a},I,\tilde{c}} S_i'$, and $B_i' = (\forall \tilde{a} \bullet I \Rightarrow B_i)$ and $(\exists \tilde{a} \bullet I \wedge \bigvee_i B_i) \Rightarrow (\bigvee_i B_i')$, then:

if $[]_i\, B_i \rightarrow S_i$ **fi** $\preceq_{a,I,c}$ **if** $[]_i\, B_i' \rightarrow S_i'$ **fi**

do $[]_i\, B_i \rightarrow S_i$ **od** $\preceq_{a,I,c}$ **do** $[]_i\, B_i' \rightarrow S_i'$ **od**

Again, these can be simplified when I is functional, giving $B_i' = B_i[a := F(c)] \wedge H(c)$ as the new guards.

It can be shown that the above rules give the "weakest" data refinement in each case. [23]. That is, if the rules give S' as the data refinement of S, then for any program S'' such that $S \preceq_{a,I,c} S''$, we also have $S' \sqsubseteq S''$. Thus, we can treat these rules as "data refinement calculators", and perform data refinement by first applying the calculators and then performing ordinary refinement on the result.

7 Data Refinement Example

Suppose we wish to data refine the algorithm derived in Section 5, so that S is represented by an array A of size n, where $n = |S|$.

We can describe the relationship between S and A as:

$$S = Elts(A)$$

where *Elts* is a function which returns the set of elements in an array (or array segment).

Since T is always a subset of S, we also represent T using the same array, so long as its elements are contiguous. We will assume that the elements of T occupy an initial segment of A, described by a variable k, so that the following holds:

$$T = Elts(A[1..k]) \wedge 1 \leq k \leq n$$

where $A[1..k]$ is the array segment from $A[1]$ to $A[k]$, inclusive. It follows that $S - T = Elts(A[k+1..n])$.

Thus, our coupling invariant is:

$$I \;\widehat{=}\; S = Elts(A) \wedge T = Elts(A[1..k]) \wedge 1 \leq k \leq n$$

To perform the data refinement, we need to replace the declaration of T by a declaration of k. The declaration of S (somewhere global) will be replaced by the declaration of A. We then replace constructs involving S and T by equivalents involving A and k, as outlined in Section 6.

The set selection $x :\in S$ data refines to x: $\left[(x \in Elts(A))\right]$, which can be refined by selecting any element of A and assigning it to x. At this point it is not clear which element of A we should choose.

The assignment $T := \{x\}$, however, data refines to k: $\left[Elts(A[1..k]) = \{x\}\right]$. We can refine this to $k := 1$, provided that $x = A[1]$.

Thus, we should refine x: $\left[(x \in Elts(A))\right]$ to $x := A[1]$. To justify this formally, we need to propagate the fact that $x = A[1]$ using assertions and laws for pushing them through other statements, or using some other technique such as those described in [18] and [7].

We replace the guard $T \neq S$ by $Elts(A[1..k]) = Elts(A)$, which simplifies to $k \neq n$.

In a similar fashion, we can data refine the remaining statements involving S and/or T, making the following replacements:

$$\begin{array}{lcl} u :\in S - T & \longrightarrow & u := A[k+1] \\ v :\in S - T - \{u\} & \longrightarrow & v := A[k+2] \\ T := T \cup \{u, v\} & \longrightarrow & k := k + 2 \end{array}$$

The resulting program is then:

$$\begin{array}{l} [\![\ \textbf{var}\ k \bullet \\ \quad [\![\ \textbf{var}\ x \bullet \\ \qquad x := A[1]; \\ \qquad lo, hi, k := x, x, 1 \\ \quad]\!]; \\ \quad \textbf{do}\ k \neq n \rightarrow \\ \qquad [\![\ \textbf{var}\ u, v \bullet \\ \qquad\quad u := A[k+1]; \\ \qquad\quad v := A[k+2]; \\ \qquad\quad u, v := u\ \textbf{min}\ v, u\ \textbf{max}\ v; \\ \qquad\quad lo, hi := lo\ \textbf{min}\ u, hi\ \textbf{max}\ v; \\ \qquad\quad k := k + 2 \\ \qquad]\!] \\ \quad \textbf{od} \\ ]\!] \end{array}$$

8 Conclusions

This paper has presented the basic elements of the refinement calculus, including data refinement, and illustrated its use in deriving a fairly simple algorithm. The use of nondeterminism in an abstract algorithm allows a number of different representations to be considered during data refinement. More details of the underlying theory and more extensive examples can be found in various sources listed in the bibliography.

Acknowledgements

Thanks to Peter Andreae, Gill Dobbie and Ray Nickson for their helpful comments on earlier versions of the paper.

References

[1] R. J. R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications.* Mathematical Centre Tracts 131, Mathematisch Centrum, Amsterdam, 1980.

[2] R. J. R. Back. "On correct refinements of programs". *Jl. Computer and System Sciences* **23** (1981), pp49–68.

[3] R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica* **25** (1988), pp593–624.

[4] R. J. R. Back and J. von Wright. "Duality in specification languages: a lattice-theoretical approach". *Acta Informatica* **27** (1990), pp583–625.

[5] R. J. R. Back. *Data Refinement in the Refinement Calculus.* Technical Report Series A, No. 68, Inst. för Informationsbehandling, Åbo Akademi, Turku, Finland, 1988.

[6] H. Barringer *et al.* "A logic covering undefinedness in program proofs". *Acta Informatica* **21** (1984), pp251–269.

[7] W. Chen and J. Y. Udding. Towards a calculus of data refinement. *Mathematics of Program Construction'*, J. L. A. van de Snepscheut (Ed.), Springer-Verlag, Lecture Notes in Computer Science 375, 1989, pp197–218.

[8] E. W. Dijkstra. "A constructive approach to the problem of program correctness". *BIT* **8** (1968), pp174–186.

[9] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs", *Communications of the ACM*, **18** (August 1975), pp453–457.

[10] E. W. Dijkstra, *A Discipline of Programming*, Academic Press, 1976.

[11] P. H. B. Gardiner and C. C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science* **87** (1991), pp143–162. Also in [25], pp86–102.

[12] David Gries, *The Science of Programming*, Springer-Verlag, 1981.

[13] Lindsay Groves, Raymond Nickson and Mark Utting. "A Tactic Driven Refinement Tool". *Proc. 5th Refinement Workshop*, Springer-Verlag, 1992.

[14] C. A. R. Hoare. "Proof of correctness of data representations". *Acta Informatica* **1** (1972), pp271–281.

[15] Cliff Jones. *Software Development: A Rigorous Approach.* Prentice-Hall, 1981.

[16] Cliff Jones. *Systematic software development using VDM.* Prentice Hall, 1990.

[17] Carroll Morgan. The specification statement. *TOPLAS* **10**, 3 (July 1988). Also in [25], pp7–30.

[18] C. C. Morgan. Data refinement by miracles. *Inf. Proc. Lett.* **26**, 5 (Jan. 1988), pp243–246. Also in [25], pp72–78.

[19] Carroll Morgan. Auxiliary variables in data refinement. *Information Processing Letters* **29** (1988), pp293–296. Also in [25], pp79–85.

[20] Carroll Morgan. Procedures, parameters, and abstraction: separate concerns. *Sci. Comp. Prog.* **11** (1988). Also in [25], pp58–71.

[21] Carroll Morgan. Types and invariants in the refinement calculus. *Mathematics of Program Construction*, J. L. A. van de Snepscheut (Ed.), Lecture Notes in Computer Science, Vol. 375, Springer-Verlag, 1989.

[22] Carroll Morgan. *Programming from Specifications.* Prentice-Hall, 1990.

[23] Carroll Morgan and P. H. B. Gardiner. Data refinement by calculation. *Acta Informatica* **27** (1990), pp481–503. Also in [25], pp103–134.

[24] Carroll Morgan and Ken Robinson. Specification statements and refinement. *IBM Jnl. Res. Dev.* **31**,5 (Sept. 1987). Also in [25], pp31–57.

[25] Carroll Morgan, Ken Robinson and Paul Gardiner. *On the Refinement Calculus*, Technical Monograph PRG-70, Oxfrod University, 1988.

[26] J. M. Morris. "A theoretical basis for stepwise rèfinement and the programming calculus". *Science of Computer Programming* **9** (1987), pp287–306.

[27] J. M. Morris. Laws of data refinement. *Acta Informatica* **26** (1989), 287–308.

[28] R. G. Nickson. *Tool Support for the Refinement Calculus.* PhD thesis, Victoria University of Wellington, 1994.

[29] Raymond G. Nickson and Lindsay J. Groves. "Metavariables and Conditional Refinements in the Refinement Calculus" *Proc. 6th Refinement Workshop*, British Computer Society, London, January 1994.

[30] Niklaus With. *Program development by stepwise refinement. C.A.C.M* **14** (April 1971), pp221–227.

[31] J. B. Wordsworth. *Software development with Z : a practical approach to formal methods in software engineering.* Addison-Wesley, 1992.

[32] J. von Wright. "The lattice of data refinements". *Acta Informatica* **31** (1994), pp105–135.

Pragmatics of the Action Semantics Approach: Fault Tolerance As An Example[1]

Padmanabhan Krishnan
Department of Computer Science
University of Canterbury
Private Bag 4800
Christchurch 1, New Zealand
E-Mail: paddy@cosc.canterbury.ac.nz

Abstract

In this paper we show how the action semantics framework can be used to describe a particular implementation of fault-tolerant systems. We also define a notion of simulation which can be the basis for relating a fault-tolerant implementation to an abstract (non-faulty, and non-fault tolerant) specification. The aim of the paper is to illustrate that good software engineering techniques can be applied to semantic descriptions. Issues such as modularity, extensibility of the semantic descriptions is illustrated.

1 Introduction

Software reuse is considered to be one of the principal areas from which productivity gains is expected. Various researchers [1, 2] have shown that there are various aspects to reuse and it is important to realise the scope and the effect of reuse. Just as reuse is important for software development, reuse can also be crucial in the development of formal specifications. In general, development and maintenance of formal specifications is no different from the development and maintenance of software systems.

In software systems, if one has to extend the domain of application, one does not rewrite the software from scratch. However, as far as we are aware, no software engineering principles are applied to the development of formal specifications. The traditional role of formal specification is as a fixed entity that guides issues such as verification, validation and implementation. The development of formal specifications itself could be error prone and it is essential to apply something like the waterfall model [15] to it. Furthermore, if a formal specification is to have a lasting value, it should be easy to update it to obtain extensions.

The aim of this paper to show that action semantics [9] is a good choice for formal specifications as it exhibits modularity which enables the reuse of parts of a specification. The scalability of action descriptions has been shown in [11] where the addition of concurrency had minimal influence on the semantic definitions used to describe sequential computation. The underlying notation and its semantics itself required no change. The addition of interrupts to the notation was essential to model certain aspects of fairness. This required a change to the operational semantics of the notation [6, 5]. But overall the nature of the changes were simple and the changes themselves were fairly small.

In this paper we focus on the semantics of a simple language in which certain types of fault-tolerant systems can be expressed. With safety critical systems gaining in importance [3], the software engineering aspects of formal specifications is becoming more relevant.

[1]Work in Progress: Supported by UoC Grant 1787123

We begin by considering a semantic description for a simple sequential language. An extension to specify the semantics of faults is considered which is followed by an extension to specify the semantics of fault-tolerance.

2 Notational Details

Action semantics uses a notation in which the semantics of realistic languages like Pascal or Ada can be defined. Such semantic descriptions are compositional (as in traditional denotational semantics) where the co-domain of the semantic functions contains actions (instead of λ-terms or higher order functions). Actions are objects that have an operational intuition and indeed the semantics of actions is described using the SOS approach [9][pages 278,295]. The SOS defined the actions induces an operational semantics for the language being developed and this is used to develop compiler generators [13, 14, 12].

The entire notation is based on a set of primitive actions and various combinators to create more complex actions. The application of the operational semantics for actions results in the processing of information. Depending on the type of information being processed actions are categorised into various facets.

Some predefined data notation, which includes numbers, characters, sets, tuples, is provided. This can be extended to define any data type required by the semantics. Certain classes of values which depend on the state of the computation (called yielders) are also identified. The yielders are evaluated to get a specific value of the appropriate type.

The following tables along with their intuitive descriptions summarise the various facets along with their primitive actions and a few combinators. The reader is referred to [9] (pages 261-277) for details.

Actions/Yielders	Combinators
complete, diverge, escape fail, commit, unfold	and then, or trap, unfolding

Table 1: Basic Actions

The action complete always terminates, while diverge never terminates. The action fail indicates abortive termination and is used to abandon the current alternative. The action commit corresponds to cutting away all alternatives, while the action escape corresponds to raising an exception. The combinator and then corresponds to sequential performance while the combinator and performs two actions with arbitrary interleaving. The combinator or represents non-deterministic choice. An alternative to the chosen action is performed when the chosen action fails (unless a commit has been performed). The combinator trap is used to handle exceptions raised using escape. The combinator unfolding along with the basic action unfold specifies iteration. unfolding A performs A, but when the action unfold is encountered in A, the action A is performed.

The action give D yields the datum D while the action give D#n yields the n'th component of the tuple represented by D. The action regive regenerates any data given to it and is useful to make copies of the given data. The action choose S gives an element of the data of sort S while check D completes if D is the boolean true; fails otherwise. The principal functional combinator is then. A_1 then A_2 corresponds to functional composition, i.e., A_2 is given the data produced by A_1.

Actions/Yielders	Combinators
give, choose, regive check	then

Table 2: Functional Actions

Actions/Yielders	Combinators
bind to , rebind, produce, bound to current bindings	moreover, hence, before

Table 3: Declarative Actions

The declarative actions process scoped information and associate tokens (identifiers in the semantic domain) with values. The action bind T to D, which produces a binding of token T to datum D, rebind which reproduces all the bindings it received and produce D which converts the data item D into a binding. Information from the current bindings can be extracted by the S bound to T returns the datum (if it is of sort S) bound to the token T. The data specification current bindings converts the entire set of bindings into data. This combined with produce permits the manipulation of bindings as data and reconverting data into bindings.

The action A_1 moreover A_2 corresponds to letting bindings produced by A_2 override those produced by A_1, i.e., bindings produced by A_2 have a higher precedence. The action furthermore A is similar and produces the same bindings as A along with any received bindings that is not overridden by A. The action A_1 hence A_2 restricts the bindings received by A_2 to those produced by A_1 and bindings produce by A_2 is propagated. This limits the scope of bindings produced by A_1 unless A_2 reproduces them.

Actions/Yielders
store, allocate, stored in deallocate

Table 4: Imperative Actions

The imperative actions deal with storage, consisting of individual cells, which is stable information. The action store D_1 in D_2 stores the datum D_1 in cell D_2 while allocate D corresponds to the allocation of a cell of sort D while the action deallocate D destroys the allocation of cell corresponding to D. Data of sort S that is stored in a cell D can be extracted by (S stored in D).

In many cases it is necessary to treat actions as data. For example, binding the body of a procedure to an identifier, the action representing the body needs to be treated as data. An abstraction is a data type that incorporates an action. Abstractions are created using the constructor abstraction of. References to transient data in an action is not evaluated when the abstraction is created. Transient information (i.e., parameters) can be given to the abstraction by application Abs to D, where the data D is supplied to Abs. Similarly bindings can be supplied to abstraction Abs using closure Abs. Actions converted into abstractions can be performed using the action enact. For example, the abstraction Abs is executed by enact Abs.

Actions/Yielders
enact, application to , closure, abstraction of

Table 5: Reflective Actions

The action notation also supports concurrency. Concurrent behaviour is represented by agents which evolve asynchronously. The agents can communicate via message passing which can be used to synchronise agents. This concludes our brief overview of the action notation. A reader who is interested in the more technical aspects of the notation is referred to [9] where the operational semantics for the notation and a number of algebraic laws that the actions satisfy are developed.

2.1 A Simple Sequential Language

In this section we describe the syntax and semantics of a simple language. We will assume that variables hold only numbers and are created statically. The purpose of this language is only to illustrate the use of the various actions/combinators. We will extend this language to address issues of fault-tolerance.

Towards defining the semantics of this language, we define four semantic functions. The first, establish, creates the necessary storage for all the variables that hold numbers. The second, evaluate describes the evaluation of expressions while the third, execute, defines the semantics of execution, i.e., the flow of control and state changes. The final equation, run, defines the semantics of programs which at first establishes bindings and then executes the program. The formal definitions are given below.

Id = ⟦ letter$^+$ ⟧

Expr = Id | ⟦ Expr "+" Expr ⟧ | ⟦ Expr "=" Expr ⟧

St = ⟦ Id ":= Expr ⟧ | ⟦ "if" Expr "then" St "else" St ⟧ |
⟦ "while" Expr "do" St ⟧ | ⟦ St ";" St ⟧

Pgm = St | ⟦ Id ":" Pgm ⟧

establish :: Id → action

establish I:Id = allocate a num-cell then bind it to the token of I

evaluate :: Expr → action

evaluate I:Id = give the contents of (the cell bound to token of I)

evaluate ⟦ E1 "+" E2 ⟧ = | evaluate E1 and evaluate E2
then
| give the sum(number #1, number #2)

evaluate ⟦ E1 "=" E2 ⟧ = | evaluate E1 and evaluate E2
then
| give same(number #1, number #2)

execute :: St → action

```
execute ⟦I:Id "::=" E:Expr⟧ = evaluate E then assign the value (to the datum bound to token of I)

execute ⟦ "if" E:Expr "then" S1:St "else" S2:St ⟧ = evaluate E then
                                                    | check (it is true) and then execute S1
                                                   or
                                                    | check (it is false) and then execute S2

execute ⟦ "while" E:Expr "do" S:St ⟧ = unfolding
                                       | evaluate E then
                                       | | check (it is true) and then
                                       | | | execute S1 and then unfold
                                       | or
                                       | | check (it is false)

execute ⟦S1:St ";" S2:St ⟧ = execute S1 and then execute S2

run :: Pgm → action

run S:St = execute St

run ⟦ I:Id ":" P:Pgm⟧ = establish I moreover run P
```

The semantic equations are quite straightforward. More details can be obtained from the action semantics tutorial [10].

3 Fault Specification

We extend the simple language to include faults. We consider two types of faults. They are garbling of state (i.e, values associated with variables) and crash failure (i.e., a cell becomes inaccessible). Following [4], we model faults as 'normal processing' which operates in asynchronous conjunction with the rest of the program. Thus the system has no control over when (if at all) the faults occur.

The formal extension to the grammar is given below.

```
Failure  = ⟦ "corrupt" Id ⟧ | ⟦ "fail" Id ⟧ | ⟨Failure+⟩

FPgm     = ⟦ St "|" Failure⟧ | ⟦ Id ":" FPgm ⟧
```

The inclusion of Failure is a pure addition to the original grammar while the old program had to be extended to include potential faults. This requires us to add a new semantic function, fexecute, for Failure and alter the semantics of run to obtain frun which uses the semantics of fexecute.

As this point we have not yet addressed the issue of fault-tolerance. All we have done is to specify the semantics of faults.

```
fexecute ⟦ "corrupt" I:Id ⟧ = | | give the datum bound to token of I
                              | and
                              | | choose a number
                             then
                              | assign the number #2 to the datum #1

fexecute ⟦ "fail" I:Id ⟧ = | give the datum bound to token of I
                           then
                           | deallocate it
```

fexecute ⟨F1:Failure F2:Failure$^+$⟩ = fexecute F1 and fexecute F2

frun :: FPgm → action

frun ⟦ S:St "|" F:Failure⟧ = execute S and fexecute F

frun ⟦ I:Id ":" P:FPgm⟧ = establish I moreover frun P

The semantic definitions are as expected with all the faults operating in asynchronous fashion. As the effect of the faults depends on the bindings of identifiers, the use of the combinator moreover after establish and and between execute and fexecute ensures that the statements and the faults get the same bindings.

4 Fault Tolerance

In this section we add features which are useful in building fault-tolerant systems. We use replication of a cell to withstand data corruption and failure. The degree of replication is specified by the programmer.

Protect = ⟦ "copies" Id Expr ⟧ | ⟦ Protect$^+$ ⟧

p-establish :: Protect → action

p-establish ⟦ "copies" I:Id E: Expr ⟧ = evaluate E then
| replicate (the given number#1) then
| | bind them to the token of I

replicate 1 = allocate a m-cell

replicate n = allocate a r-cell and (replicate (n-1))

The degree of replication is specified by an expression. Instead of using num-cells we now use m-cell and r-cells. The reason for the two types of cells will become clear later when we relate the extended semantics to the original one. Intuitively, we use the m-cell as an anchor to relate it to the num-cell.

As we have changed the structure of the bindings of identifiers, the semantic equations evaluate and execute need to be changed. These are specified below.

evaluate I:Id = give the datum bound to token of I then
vote-value them

execute ⟦I:Id ":=" E:Expr⟧ = | evaluate E and give the datum bound to token of I
then
| assign-forall (first of them) to (the rest of them)

We leave the exact specification of voted-value and assign-forall open. The intuition is that in voted-value all the legal cells are inspected. The values are then accumulated and a voting strategy applied to them. For example, one can adopt majority voting or an average value voting. The process of inspecting legal cells is given below.

inspect d = | choose (d & current-storage) and then give the contents of d
or
| choose (disjoint-union (d, current-storage)) and then complete

If a cell has been deallocated due to failure, no value is returned. Notice that we do not change the bindings as we do not have any technique of passing on the new bindings to the other statements. If a change in bindings has to affect most of the actions, it is almost essential to commit them to stable storage.

The intuition behind assign-forall is similar in that it assigns the same value to all the currently legal cells.

Due to a change in the representation of the cell, the semantic equations describing the meaning of faults also need to be changed.

```
fexecute ⟦"corrupt" I:Id⟧ = | | give the datum bound to I then (select-one #1)
                            | and
                            | | choose a number
                            then
                            | assign the number #2 to the datum #1

fexecute ⟦ "fail" I:Id ⟧ = | give the datum bound to I then (select-one #1)
                           then
                           | deallocate it
```

The nature of the change is similar to the changes in execute etc. We leave the exact semantics of select-one unspecified. The intuitive behaviour of select-one is to non-deterministically select a r-cell which will then be corrupted by the assignment. Note this effect could also be achieved by changing the semantics of assign and deallocate.

4.1 Discussion of Changes

The addition of faults and fault-tolerant aspects has altered a few of the original semantic equations. However the changes were very localised. More specifically they were only to equations (and furthermore restricted to parts of equations) that explicitly dealt with the representation of identifiers, values and the communication.

If the original semantic equations could have been written in a style which used abstract data types (e.g., never exposed the structure of a binding), the changes would have been to the semantic entities only. Following an abstract data type prescription for semantics results in overly verbose descriptions and in most situations, the semantics are not drastically altered very often. What we presented as the original definition is a realistic expectation of an action description. The price paid to extend the original description to cater to fault tolerance is not very high considering the radical nature of the change. For a large language many equations will not be altered. In our toy example, we have seen that the semantics of statement sequencing, while loops etc. required no change.

5 Simulation Relation

A notion of bisimulation and testing equivalences for actions is defined. These relations based on the notion of commitments which are either messages or changes to the store. The configurations for each agent are not just actions but is a combination of actions with their given information (such as transients, bindings) operating on the local information of store and messages. The details of this is specified in [9](pages 261-295). Here we present a slightly simplified view.

state = (Acting, local-info)

local-info = (storage, buffer)

Acting = (Action, data, bindings) | (Acting In-fix Acting) | (Pre-fix Acting)

Acting represents actions operating on their given information while local-info contains the state of the store, the incoming message buffer and the identity of the agent.

The operational semantics is defined by a function stepped. If an action associated with a state s can be performed, stepped s yields the states s can evolve into along with the communications generated by the performance. An auxiliary function simplified is defined which handles the propagation of transients and bindings and termination details. For example, state ⟦ completed and A ⟧ is simplified to state ⟦ A ⟧.

(1) stepped _ :: state → (state, commitment)

(2) commitment = list of [communication] | uncommitted

The empty list is used to indicate changes to the store. This is sufficient as the exact changes are recorded in the storage component of local-info. All other transitions, such as creating a transient or a binding, reading the store etc., are labelled by uncommitted.

The following rules help to define the semantics for and.

(1) stepped (state A1 (s h)) :- (state A1' (s' h')) c' ⇒
stepped (state ⟦ A1 "and" A2 ⟧ (s h)) :- simplified (state ⟦ A1' "and" A2 ⟧ (s' h')) c'

(2) stepped (state A2 (s h)) :- (state A2' (s' h')) c' ⇒
stepped (state ⟦ A1 "and" A2 ⟧ (s h)) :- simplified (state ⟦ A1 "and" A2' ⟧ (s' h')) c'

Recall that the and combinator defines the interleaved execution of two actions. The first rule states that if the state A1 (s h) can make a transition to the state A1' (s' h') c', ⟦ A1 "and" A2 ⟧(s h) can make a transition to ⟦ A1' "and" A2 ⟧ (s' h') c'. The second rule specifies the progress of A2.

Based on the above definition a state transition relation $\longrightarrow$ is defined as follows: s $\xrightarrow{c}$ s′ iff (s′,c) : stepped s. An observable transition $\Longrightarrow$ is defined as $\stackrel{c}{\Longrightarrow} = (\xrightarrow{\text{uncommitted}})^* \xrightarrow{c} (\xrightarrow{\text{uncommitted}})^*$

Definition: 1 *Two actions A1 and A2 are equivalent iff forall local information l*

(A1 l) $\xrightarrow{c}$ *(A1′,l′) then (A2 l)* $\stackrel{c}{\Longrightarrow}$ *(A2′, l′) and (A1′, l′) is equivalent to (A2′, l′) and*

(A2 l) $\xrightarrow{c}$ *(A2′,l′) then (A1 l)* $\stackrel{c}{\Longrightarrow}$ *(A1′, l′) and (A1′, l′) is equivalent to (A2′, l′)*

The above definition is similar to the observational equivalence defined in [7].

This definition is too strong for our purposes. It requires an exact match of all state changes. This is clearly not the case in the fault tolerant setting. Due to replication of cells, a single assignment is translated into multiple assignments. However, we would still like to relate the extended semantics (faults and fault tolerance) with the original semantics. Towards this we introduce a notion of simulation which ignores certain aspects of the behaviour.

5.1 Derived Relation

As we would like to relate the original 'perfect world' semantics to the 'fault-tolerant' semantics, what is necessary is a relationship between the data structures used in the two semantics. Influenced by the work described in [8], we introduce a function similar to the notion of abstraction invariant. This function provides a map between the various sorts used in the semantics.

As we have introduced replication, a single cell access has been translated to multiple cell accesses. In order to use the ideas behind observational equivalence, we have rename all the extra accesses to uncommitted. The derived relation is indexed by such a function and the usual definition of observational equivalence can be used.

Definition: 2 *Two actions A1 and A2 are equivalent under an abstraction function F iff for all local information l*

- $(A1\ l) \xrightarrow{c} (A1', l')$ *and* $F(c) \neq$ uncommitted *then* $(A2\ l) \stackrel{cs}{\Longrightarrow} (A2', l')$ *where*

  $cs = c_1, c_2, \ldots c_n$ *with* $F(c_i) = c$ *and for all other* j, $F(c_j) =$ uncommitted
  $(A1', l')$ *is equivalent to* $(A2', l')$

- $(A1\ l) \xrightarrow{c} (A1', l')$ *and* $F(c) =$ uncommitted *then either*

  $[(A1\ l) \stackrel{\text{uncommitted}^*}{\Longrightarrow} (A1', l')$ *and* $(A1', l')$ *is equivalent to* $(A2', l')]$ *or*
  $[(A1, l)$ *is equivalent to* $(A2', l')]$

- *Similarly for A2*

In the above definition, the effect of F is to interhalise the actions that F maps to uncommitted.

This definition can be applied to our example to show that the semantics of a given program is identical to another given certain fault assumptions and fault-tolerant techniques. For example, to withstand one fault, a triple replication with majority voting suffices.

By defining F to be such that F(m-cell) = num-cell and F(r-cell) = uncommitted, one can show that the fault tolerant semantics can be related to the perfect semantics. This is not always true, as the degree of replication may not be enough to withstand the number of faults injected into the system.

6 Conclusions and Future Work

Here we have added aspects of fault tolerance and again the good pragmatic features of action semantics have been demonstrated. We have also defined a general framework in which various extensions to an existing semantics can be related to the original one. This forms the basis for proofs of correctness of extensions and is under further investigation.

The technique we have outline can be adapted to suit other situations. For example, we can translate an array assignment (which can be a single assignment to a complex cell) into a series of assignments to individual simpler cells. By making one of the cells in the collection as a distinguished one, one can relate the sequence of assignments to the single array assignment. By increasing the domain of the function F to include messages, it is possible to specify various parallel implementations of languages.

References

[1] B. Barnes and T. Bollinger. Making reuse cost-effective. *IEEE Software*, pages 13–24, January 1991.

[2] V. Basili and H. Rombach. Support for comprehensive reuse. *The IEE Software Engineering Journal*, 6:303–316, Sept 1991.

[3] J. Bowen. Formal methods and safety-critical standards. *IEEE Computer*, 27(8):68–71, August 1994.

[4] F. Cristian. A Rigorous Approach to Fault-Tolerant Programming. *IEEE Transactions on Software Engineering*, 11(1):23–31, 1985.

[5] P. Krishnan. Specification of Systems with Interrupts. *Journal of Systems and Software: Special Issue on Applying Specification, Verification and Validation Techniques to Industrial Software Systems*, 21(3):291–304, June 1993.

[6] P. Krishnan and P. D. Mosses. Specifying Asynchronous Transfer of Control. In J. Vytopil, editor, *Proceedings of the Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems: LNCS 571*, pages 291–306, Nijmegen, Netherlands, January 1992. Springer Verlag.

[7] R. Milner. *Communication and Concurrency.* Prentice Hall International, 1989.

[8] C. C. Morgan and P. H. B. Gardiner. Data refinement by calculation. *Acta Informatica*, 27:481–503, 1990.

[9] P. D. Mosses. *Action Semantics.* Number 26 in Tracts in Theoretical Computer Science. Cambridge University Press, August 1992.

[10] P. D. Mosses. A Tutorial on Action Semantics. In *Proceedings of FME*, Barcelona, Spain, September 1994.

[11] P. D. Mosses and M. Musicante. An Action Semantics for ML Concurrency Primitives. In *Proceedings of FME*, Barcelona, Spain, September 1994.

[12] Peter Ørbæk. OASIS: An Optimizing Action-based Compiler Generator. In Peter Fritzon, editor, *Proceedings of the 1994 Conference on Compiler Construction , Edinburgh*, volume 786 of LNCS, pages 1–15. Springer-Verlag, April 1994.

[13] J. Palsberg. A Provably Correct Compiler Generator. In *European Symposium On Programming (ESOP): LNCS 582*, pages 418–434, Rennes, France, February 1992. Springer-Verlag.

[14] J. Palsberg. An Automatically Generated and Provably Correct Compiler for a Subset of Ada. In *Fourth IEEE International Conference on Computer Languages*, San Fransisco, California, April 1992.

[15] Ian Sommerville. *Software Engineering.* Addison Wesley, third edition, 1989.

The Massey Paradigms and Languages Group: Projects and Plans

Neil Leslie and Nigel Perry
Department of Computer Science
Massey University
Palmerston North
N.{Leslie,Perry}@massey.ac.nz

Abstract

The Massey Paradigms and Languages (PAL) Group's primary interest is in the development of a concurrent functional language paradigm, and of a reference language implementation. Subsidiary to this interest we are interested in: type systems; concurrent runtime systems and garbage collection; high performance "functional" arrays; and state in functional programming systems. This talk will give an overview of our current work and future plans in these areas.

1. Introduction

In this paper we present an introduction to the work of the Massey Paradigms and Languages (PAL) group. The paper has the following structure: in the first part we motivate our interest in combining concurrency and functional programming, recap some history, give a short introduction to continuations and present some techniques which make use of them; in the second part we discuss object/process orientation, formal semantics and state in functional programming; finally we draw some tentative conclusions.

2. Concurrent Functional Programming Systems

The interests of the Paradigms and Languages (PAL) group at Massey are:

- functional programming languages;
- concurrency.

We re-iterate the usual arguments as to the advantages of declarative programming languages: programs written in such languages are mathematically simpler, are easier to reason about and hence are easier to write and prove correct (or derive) and transform. We are particularly interested in functional , as opposed to relational, programming.

We also believe that concurrency is an important issue to tackle for variety of reasons:

- multi-processor hardware is now widely available commercially;
- conventional (imperative) languages do not support concurrent programming well;
- extensions to these languages have not reduced the programming burden greatly.

We distinguish between *concurrency* and *parallelism*:

- in a concurrent system two or more tasks are executed at the same time, hence concurrency requires explicit control over processes;
- in a parallel system one task may be executed as a number of sub-tasks run at the same time, hence parallelism provides no (explicit) control over processes.

Our overall aim is therefore to see how the advantages of the functional programming style can be brought to the world of concurrency.

3. Background and History

Hope is a typed, pure functional language and, like all the best programming languages, can trace its origins to Edinburgh [8]. However, in this project we are continuing work which started at Imperial College in London [18, 19], and has been continuing at Massey since 1991, where Massey Hope$^+$C was implemented. We will use "Hope" to refer to Massey Hope$^+$C where we think no confusion can arise.

Massey Hope$^+$C has:

- existential types,
- a secure module system;
- a referentially transparent, continuation based I/O system;
- not a monad in sight.

Some of our work is inspired and motivated by lessons learned from Massey Hope$^+$C. In particular we expect to make heavy use of continuations.

4. Continuations

The use of continuations is important in our work so we shall give a short introduction here. A continuation is a function k which embodies the rest of the computation after a given point. For any function we can produce a version of in written in continuation passing style (CPS). We illustrate this with some examples.

Given the following definition, in Hope syntax, of *fib*:

```
let fib : num -> num == fun
    0 => 1
  | 1 => 1
  | n => fib(n-1) + fib(n-2)
end;
```

We can produce a CPS conversion of this function:

```
let cps_fib : num # (num -> alpha) -> alpha == fun
    (0, k) => k(1)
  | (1, k) => k(1)
  | (n, k) => cps_fib(n-1,
                      fun r => cps_fib(n-2,
                                       fun s => k(r + s)
                                       end)
                      end)
end;
```

We have added more function calls, which may be seen as a retrograde step. The resulting function stands out in a number of ways: first it seems to have been turned inside out; second we have imposed an order on the evaluation of the arguments to + that was not specified before.

As a second example we will consider *listrec* and two CPS variants of it. *Listrec* is the structural induction operator on lists. *Listrec* may be written in Hope as:

```
let listrec : list(alpha) #
              beta #
              (alpha # list(alpha) # beta -> beta) ->
              beta == fun
   (nil, d, _) => d
 | (h::t, d, e) => e(h, t, listrec(t, d, e))
end;
```

Again we can produce a CPS conversion of this function:

```
let cps_listrec : list(alpha) #
                  beta #
                  (alpha #
                     list(alpha) #
                     beta ->
                     beta) #
                  (beta -> gamma) ->
                  gamma == fun
   (nil, d, _, k) => k(d)
 | (h::t, d, e, k) => cps_listrec(t, d, e,
                                  fun r => k(e(h, t, r))
                                  end)
end;
```

This is a rather half-hearted CPS conversion. One of the advantages of CPS conversion is that we can do away with the stack, but the evaluation of *e(h, t, r)* above does not do this. To address this we can make *e* itself be in CPS:

```
let cps_listrec2 : list(alpha) #
                   beta #
                   (alpha #
                     list(alpha) #
                     beta #
                     (beta -> gamma) ->
                     gamma) #
                   (beta -> gamma) ->
                   gamma == fun
   (nil, d, _, k) => k(d)
 | (h::t, d, e, k) => cps_listrec2(t, d, e,
                                   fun r => e(h, t, r, k)
                                   end)
end;
```

Continuations turn up in our work in two distinct places:

- during a stage in compilation;
- as the basis for the I/O system, and consequently as a foundation for much else.

We discuss these below.

4.1. Compiling using continuations

A Hope program is compiled first by translating it to an intermediate code called FPM [4, 19, 20]. The FPM code is subject to a number of optimising transformations before code for the specific target machine is produced. One of the transformations which we will perform on the FPM will be that of CPS conversion. There are a number of existing compilers which use CPS conversions, for instance, the NJML compiler, as described in [3], and the ORBIT compiler for Scheme [14]. Some interesting work on the utility of CPS conversions for program optimisation is described in [23].

Previous work using continuations internally has discussed the following advantages:

- better code generation is possible;
- some optimisations on code become apparent (although others may be obscured).

Previous work has concentrated on strict sequential languages, such as ML and Scheme. We hope to extend this work to non-strict, concurrent ones.

4.2. Continuations and I/O, foreign language function calls, remote procedure calls, objects/processes

As [19] shows continuations enable us to provide referentially transparent I/O and inter-language calling in a functional setting. We can extend this idea, as is shown in [21], to allow concurrency, object/process orientation.

One way to view Massey Hope$^+$C's continuation based I/O system is to think, rather than of the execution of one single program, of the execution of a sequence of programs, connected *via* their continuations and between which I/O is performed. These continuations are of a special form, each one being a *data constructor* (rather than a function) in a special type called *Result*. As a simple example suppose we have the following *Result* type:

```
data Result == Stop
            ++ ReadChar(char -> Result)
            ++ WriteChar(char # Result);
```

Each of the constructors of this type has some special meaning for the O/S:

- *Stop* means "stop";
- *ReadChar(f)* means "read a character and apply *f* to it";
- *WriteChar(c, r)* means "print *c* and evaluate *r*".

As an example of how continuation-based I/O works in practice, suppose we now make the following definitions:

```
! forward declarations for mutual recursion !
dec one : Result;
dec two : char -> Result;

let one == ReadChar(two);
let two == fun
   c => WriteChar(c, Stop)
end;
```

and then evaluate *one*.

One reduces to *ReadChar(two)*, which the O/S interprets by reading a char *a* and then evaluating *two(a)*. *Two(a)* reduces to *WriteChar(a, Stop)*. The O/S interprets this by writing *a* and then evaluating *Stop*, which terminates execution. So we have a *sequence* of *referentially transparent* programs which are capable of performing I/O.

For the continuation-based I/O system we need two things:

- an appropriate *Result* type;
- an O/S capable of interpreting the constructors of this type.

We can extend the technique to allow us to handle:

- foreign language function calls;
- co-routines;
- remote function calls (RFCs);
- objects;
- processes.

The use of the existential types of Hope$^+$C enhances all of these things. The existential types give greater flexibility in, for example, foreign language function calls where the result of a function call can be of any type, as opposed to Haskell where a C function is required to return an integer. For a comparison of some different functional I/O systems see [11].

Having presented continuations and illustrated some of their uses we shall now speculate about other aspects of our work.

5. Object and process orientation

An object is a named piece of state. The name of an object does not change, but its contents may.

Why are we interested in object orientation anyway? Partly because this is a good word to put on papers/grant applications. There are more sensible reasons: in some of the imperative approaches to concurrency objects have been useful; it is natural to think of a file as an object and it is natural to think of a window as an object. If we can deal well with objects then we may be able to provide good graphical user interface facilities.

When considering objects we must decide whether we want objects which *inherit* or objects which *delegate*.

To illustrate the difference consider the following example. Suppose we wish to describe two different birds, *Tweety* and *Pingu*. In an inheritance based system we might start by defining a class Bird with attributes, say, of name, colour, covering, number of limbs, flying ability and so on. When we create objects corresponding to Tweety and Pingu we will *copy* the values for the attributes covering and number of limbs into the new objects, i.e we will have the following objects:

```
B1 == Bird(name == Tweety,
           colour == yellow,
           covering == feathers,
           limbs == 4,
           can_fly == true);
```

```
B2 == Bird(name == Pingu,
           colour == black and white,
           covering == feathers,
           limbs == 4,
           can_fly == false);
```

In a delegation based system we define a *ProtoBird* object:

```
ProtoBird == Bird(name == UnNamed,
                  colour == Unknown,
                  covering == feathers,
                  limbs == 4,
                  can_fly == true);
```

Tweety and Pingu are then defined in terms of this:

```
B1 == Bird(name == Tweety,
           colour == yellow,
           covering == ProtoBird:covering,
           limbs == ProtoBird:limbs,
           can_fly == ProtoBird:can_fly);

B2 == Bird(name == Pingu,
           colour == black and white,
           covering == ProtoBird:covering,
           limbs == ProtoBird:limbs,
           can_fly == false);
```

These objects *delegate* the task of supplying a value for some of their attributes to the ProtoBird object.

Where we have inheritance we tend to find large objects at the leafs of the inheritance tree as methods and instance variables are accumulated. With delegation objects do not tend to become so bloated. It is also easier to make delegation dynamic. Object orientation using delegation seems more suited to concurrent execution: but this question remains open. We call an object oriented system using delegation *process oriented.*

6. Formal Semantics

We are more interested in operational than denotational semantics. What should the operational semantics of such a language be based on? There are two obvious candidates. Recently the has been a lot of interest in the relationship between what has been called classical linear logic and concurrent computation [1, 2, 10]. We are hopeful that there are insights in this work in on how we may provide a formal semantics for a practical concurrent functional language. The other obvious candidate is one of the descendants of CCS, such as the π-calculus [16]. Work [6] has been done on the relationship between the π-calculus and linear logic. We may find that one or other of these formalisms may prove more tractable in practice.

7. State in functional programming systems

The issue of how state may be effectively handled has become important in the functional programming community. This has been driven by both theoretical and practical pressures.

On the one hand we now (think we) have ways to handle state without losing the mathematical properties of functional programs which we claimed to be such great benefit. On the other hand if we are to encourage the use of functional programming languages by "external" users we must be able to deliver a certain degree of efficiency. Run-time efficiency is not the be-all and end all: in our own estimation efficiency comes a very poor second to correctness, but inefficiency is something which we should strive to eliminate. Many scientific and business computing tasks involve manipulating large amounts of data, typically held as arrays or tables. If we can avoid needless copying we may be able to introduce more people to the benefits of functional programming.

There are a number of ways which we may choose to handle mutable state. We consider some of them very briefly. For a fuller survey see [22].

7.1. Augmented type systems

Concurrent Clean [7, 9, 13, 17] is a functional language which has been developed at the University of Nijmegen. It has a type system which allows annotated types, in particular:

- strictness and
- uniqueness

annotations can be added to a type. Strictness annotations are essentially an aid to efficiency, Uniqueness tells the compiler that it is safe to perform destructive updates. The notion of a unique type is closely related to that of a linear type [25], which were inspired by linear logic. Various languages have been implemented which make use of linear types [5]. If only the linear fragment of the type system is used the compiler is always free to perform destructive updates and garbage collection is trivial. Unfortunately the purely linear fragment is not expressive enough, so there is no free lunch here. However some of the insights from linear logic may prove to be of use.

7.2. Imperative Lambda Calculus

Another approach is that taken in the Imperative Lambda Calculus (ILC) [24] of Swarup, Reddy and Ireland. The type system is augmented with *Obs* and *Ref* constructors (which themselves can be seen as annotations) which again tell us when destructive updates are safe.

One drawback of unique and linear types and of the ILC is that we are asking programmers to deal with a more subtle type system. We may be in danger of placing too large a burden on them.

7.3. Mutable Abstract Data Types

Hudak [12] introduced the notion of a mutable abstract data type (MADT). A MADT is "any ADT whose rewrite semantics permits 'destructive re-use' of one or more of its arguments while still retaining confluence in a general rewrite system". Furthermore, given an ADT and an axiomatisation with a linearity property a MADT can be automatically generated. A graph rewrite semantics can be given for a MADT which guarantees efficient implementation. Hudak explains how to generate a MADT, and presents a number of examples.

MADTs have a number of desirable features:

- we retain the usual Hindley/Milner type system;
- no analysis is required to investigate whether destructive update is possible;
- referential transparency is preserved.

7.4. Or, we could use a monad

As monads have been proposed as a solution to the problems of handling I/O they have also been proposed as a solution to handling state in a functioanl programming language [15]. We believe that there are still problems to be overcome in this area.

8. Conclusions

As this paper is an overview of our plans, and a description of what we intend to do it is perhaps premature to have much in the way of conclusions, however we offer the following observations:

- CPS provides solutions to a lot of problems, whether these are "optimal" or even "good" is still an open question;
- we are not pursuing the use of monads at present, other people are however and may well find solutions to the current problems involving monads;
- linear type systems look promising;
- put 'object-oriented' on project proposals.

References

[1] Abramsky, S. *Computational Interpretations of Linear Logic*. Preprint of paper to appear in TCS, Department of Computing, Imperial College, 1992.

[2] Alexiev, V. *Applications of Linear Logic to Computation: An Overview*. Tech. Report, TR93-18. University of Alberta, 1993.

[3] Appel, A. W. *Compiling with Continuations*. Cambridge University Press, Cambridge, 1992.

[4] Bailey, R. *FP/M Abstract Syntax Description, Revision 6.0*. Tech. Report, Department of Computing, Imperial College, 1985.

[5] Baker, H. G. Linear Logic and Permutation Stacks-The Forth Shall Be First. *ACM Sigarch Computer Architecture News*, 22(1), 34-43, 1994.

[6] Bellin, G., and P. J. Scott. *On the „-Calculus and Linear Logic*. Preprint, obtained by ftp from a forgotten source!, 1992.

[7] Brus, T., M. C. J. D. v. Eekelen, M. v. Leer and M. J. Plasmeijer. Clean - A Language for Functional Graph Rewriting. In *Proc. of the Third International Conference on Functional Programming Languages and Computer Architecture (FPCA '87)*, 274 (pp. 364 - 384). Portland, Oregon. Springer-Verlag, 1987.

[8] Burstall, R., D. Sanella and D. McQueen. *Hope: An experimental applicative language*. Tech. Report, CSR-62-80. The University of Edinburgh, 1980.

[9] Eekelen, M. v., H. Huitema, E. Nöcker, S. Smetsers and R. Plasmeijer. *The Concurrent Clean Language Manual*. University of Nijmegen, 1992.

[10] Girard, J.-Y. Linear Logic. *Theoretical Computer Science*, 50, 1-102, 1987.

[11] Gordon, A. D. *Functional Programming and Input/Output*. Ph.D. thesis, University of Cambridge, 1992.

[12] Hudak, P. *Mutable Abstract Datatypes*. Research Report, YALEU/DCS/RR-914. Yale University, 1993.

[13] Huitema, H. S., and M. J. Plasmeijer. *The Concurrent Clean System User s Manual, version 0.8*. Technical Report, 92-19. University of Nijmegen, 1992.

[14] Kranz, D. A. *ORBIT: An opimising Compiler for Scheme*. Tech. Report, YALEU/DCS/RR-632. Yale University, Department of Computer Science, 1988.

[15] Launchbury, J., and S. L. Petyon Jones. *Lazy Functional State Threads*. Tech. Report, Unversity of Glasgow, 1994.

[16] Milner, R. *The Polyadic „-calculus: A tutorial*. Tech. Report, ECS-LFCS-91-180. LFCS, Edinburgh University, 1991.

[17] Nöcker, E. G. J. M. H., J. E. W. Smetsers, M. C. J. D. Eekelen and M. J. Plasmeijer. Concurrent Clean. In Leeuwen, J. v., M. Rem and E. H. L. Aarts (Eds.), *PARLE*, LNCS 506 (pp. 202-220). Conferentie, Eindhoven. Springer Verlag, 1991.

[18] Perry, N. *Hope+*. IC/FPR/LANG/2.5.1/7. Imperial College of Science, Technology and Medicine, University of London, 1988.

[19] Perry, N. *The Implementation of Practical Functional Programming Languages*. Ph.D. thesis, Imperial College of Science, Technology and Medicine, University of London,

1990.

[20] Perry, N. Non-Strict Fpm. A High Performance Lazy Abstract Machine. In Gupta, G. K., and C. D. Keen (Eds.), *Fifteenth Australian Computer Science Conference*, . Hobart, Tasmania, Australia. Computer Science Association, 1992.

[21] Perry, N. Towards a Concurrent Object/Process Oriented Functional Language. In Gupta, G. K., and C. D. Keen (Eds.), *Fifteenth Australian Computer Science Conference*, (pp. 715-730). Hobart, Tasmania, Australia. Computer Science Association, 1992.

[22] Perry, N., and N. Leslie. State in functional programming: a survey. Submitted to*Australian Computer Science Conference, 95*. Adelaide, SA, Australia. Computer Science Association, 1995.

[23] Sabry, A., and M. Felleisen. Reasoning About Programs In Continuation Passing Style. In *1992 Conference on LISP and Functional Programming*. San Francisco, California. 1992.

[24] Swarup, V., U. Reddy and E. Ireland. Assignments for Applicative Languages. In Hughes, J. (Ed.), *Functional Programming Languages and Computer Architecture*, LNCS 523 (pp. 192-214). Cambridge, MA, USA. Springer Verlag, 1991.

[25] Wadler, P. L. Linear types can change the world! In *IFIP TC 2 Working Conference on Programming Concepts and Methods*. Sea of Galilee. 1990.

Building Formal Models of Graphical User Interfaces

Steve Reeves
Department of Computer Science
University of Waikato
stever@waikato.ac.nz

Abstract

In this paper we propose, in outline, a system which will allow the construction of a completely formal description of a software system even if it has a graphical user-interface as a component. The design is suggested by considering a system which has already been successfully used for teaching the language of formal logic. It allows the construction of a completely formal description (in first-order logic) of the look of any display which might appear as part of the interaction that a user might have with a system.

1. Introduction

There are now many, well-documented uses of formal specification in the program development process ([Dil, Dr, Gr] are a few of the many texts in this area) that show that not only is formal specification of software necessary, for all the well-rehearsed reasons (efficiency of construction, demonstration of correctness and ease of maintenance), but it puts the design, construction and use of software on a basis that truly allows us to speak of software engineering (SE) as a discipline that is as principled, successful and well-founded as other branches of engineering such as civil and mechanical.

However, there is one area of great importance within SE that is particularly problematical. It is where the graphical meets the textual. Currently, the methods of formal specification which are used for describing the function of software (what it does, not how it does it) are textual and give us a basis for reasoning about that function. When we describe the graphical user-interface (GUI) part of some software system we currently have a problem, though. Although we can describe the interaction (via dialogues, say) in a way similar to the way in which we can describe the function, we cannot similarly describe the way the system looks. We cannot say what it displays, at the same level of abstraction and formality.

We can, of course, say how it displays what it displays by providing the code to do it. We could draw pictures of what it displays, or describe what it displays in English, say. These are

not, however, ways of describing what the display looks like which are at the same level of abstraction and formalisation as the ways we have of describing the function of the software. Formally, the displays are second-class citizens.

Putting this another way: since we are talking about stages prior to implementation here, we do not want to have to describe the look of our systems by using any coding or other such low-level notation, since that says *how* the look comes about, not *what* the look is. However, we do want the description of what the system looks like to be formal, since later we want to reason about it and, in particular, to prove that when the system runs it really does look like what the designers and specifiers said it should look like.

Put simply, we have the problem of formally describing in words and symbols, i.e. textually, what something looks like.

This paper is motivated by what appears to be a gap in current work on the use of formalisation in interfaces. There has been much good work done in this area in recent years and particularly appealing is the work described in, for example, [Ha1].

However, though most parts of a system might be formally described, the display, i.e. what the screen shows, never seems to be (apart from at the level of implementation, but we have already said that is not what we want when doing specifications). A good example of this is the very well-presented paper [Ha2] where the set "D" is used within discussion of formal specifications for interactions, where D is the set of displays Algebraic properties of D are discussed in various parts of the paper, but the elements of D themselves are never discussed, though we are told that a display, i.e. an element of D, is "a visual representation of some or all of the state...and might be, for example, an array of pixels (the details are not important)". This is very definitely relegating displays to a lower-level, and we have no hope of reasoning about them.

Another paper in the same collection [Al] goes some way towards overcoming the problem by allowing a designer to see how the interaction of a dialogue causes changes in a display, but it still leaves open the problem that what the display looks like cannot be reasoned about within the specification. A point from Lieberman, that it is difficult for designers to visualise a dialogue from a static description, is used in [Al]; we would paraphrase this by saying that it is also difficult for a designer to visualise a display from a static, or from a low-level, description. It is certainly impossible to reason about a display given such a description.

There are many other examples of work in this area which do use formal descriptions, e.g. Petri nets (though they suffer from the further problem that there is not a logic of Petri nets) or

modal logics, but again these describe the way the system behaves during dialogue, not what it looks like during these dialogues. So, all this work is really focussed on formally describing dialogues, not with describing what each stage of a dialogue looks like, which must surely be important for the designer.

We aim to suggest one way that might help to bridge the current gap between the textual nature of designs and formal specifications and the graphical nature of the look of a system. The way we suggest looks, at first, very surprising. We consider a piece of software that has been developed over several years with the aim (which it achieves very well [Go1]) of supporting the teaching of formal logic to undergraduates (and senior school and college students).

This might seem to be far removed from the problem being addressed, so first of all, in order to be clear about what follows, this software will be illustrated in the next section. In order to see the point of this paper it is important to resist the temptation to skip over it!

In the third section we will look at how some of the ideas behind the software described in section two can help with our problem of providing formal, textual descriptions of the look of systems.

In the fourth and final section we will consider how this work might be taken forward and improved.

2. Tarski's World

Tarski's World [Ba], abbreviated to TW in the rest of this paper, was developed to support the teaching of logic. Some descriptions of other systems with the same aims, as well as TW, are given in [Go2]; suffice to say that TW was one of the best. The author has had the pleasant experience of using it for teaching first-year undergraduates in computer science for a number of years. It is a robust, well-designed system and achieves its aims very well.

As far as understanding TW and the rest of this paper is concerned, the important point to note is that the objects and relationships that exist in a certain picture, called a situation, are described by a set of formal logic sentences, called the description. The situation gives a truth-value of True to each of the sentences in the description, so another way of thinking of the relationship between the situation and the description is to consider the situation as giving an interpretation of the sentences in the description which makes them all true.

Consider the situation and associated description in figure 2.1. Here each of the sentences in the description is true in the given situation. So, the description describes the situation. At a

certain level of precision we have reduced the graphical information in the situation to the formal, textual information in the description.

The phrase "at a certain level" must occur in the previous sentence since other, different, descriptions may be given for the same situation; figure 2.2 gives an example where all the sentences are true in the situation in figure 2.1 too (note that the symbol '^' means 'and'). So, there can be more than one description for some situations. However, descriptions do enjoy the property of being consistent: that is, all the sentences in any set of descriptions of some situation will all be true in that situation. No two descriptions of a situation can contradict one another and a bigger, more complete, description can always be made by collecting together all the sentences in a set of descriptions of some situation.

Also, we often find that one description A is stronger than another B, which means that description A contains all the sentences that description B does, plus some more. The general case is where description A entails description B, which means that the sentences in description B follow logically from those in A, given some suitable logical definitions of the relations.

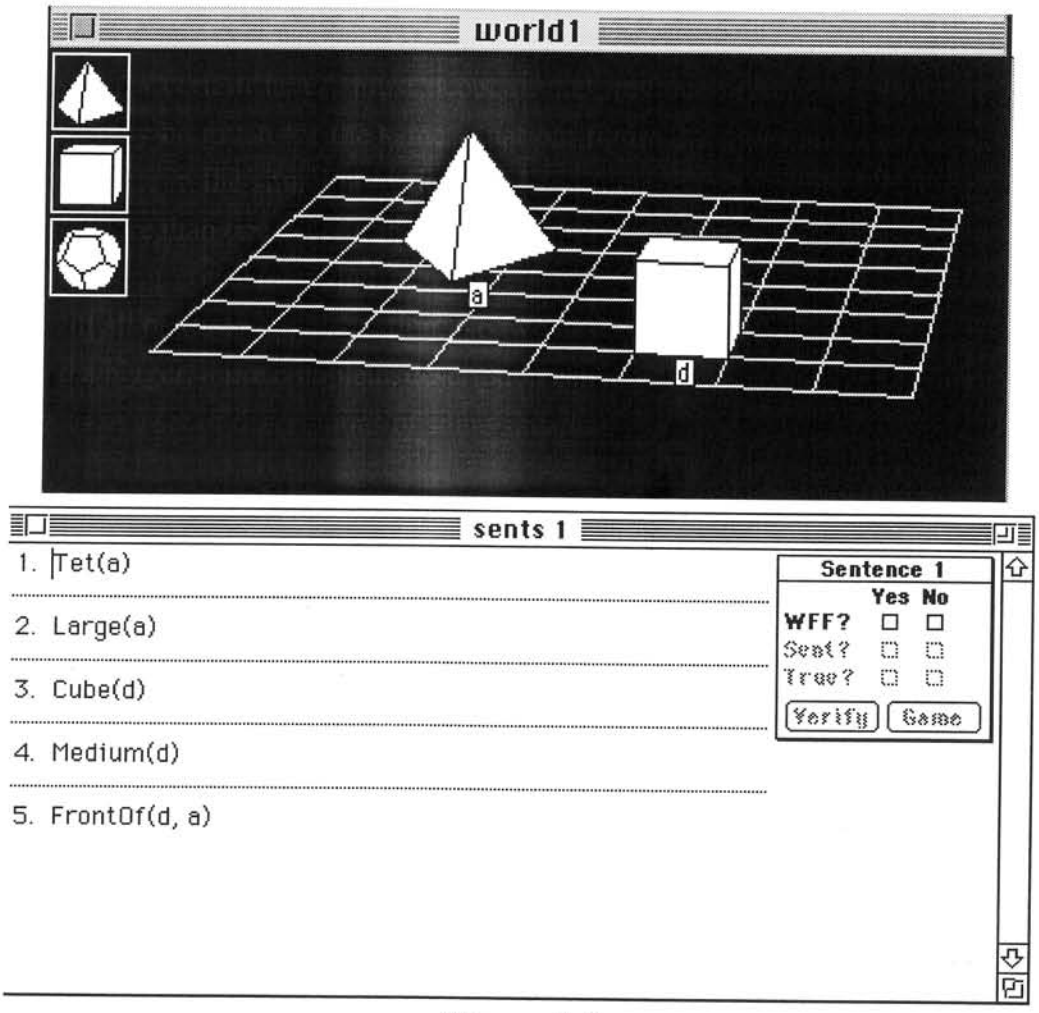

Figure 2.1

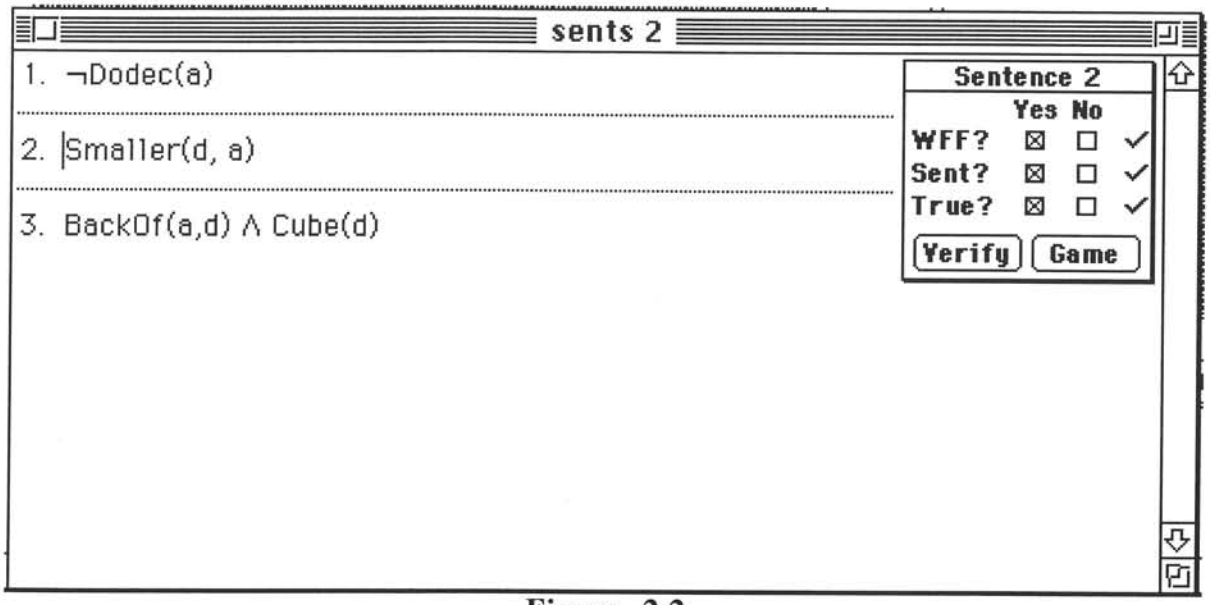

Figure 2.2

An important part of TW is the game. As we shall see, it allows the user to understand why a sentence that they thought was true (or false) in the situation is actually false (or true). In this way the user can learn about the relationship between the situation and the description by exploration; the user can play the game in order to see why a given sentence has the truth-value it has in the given situation and to see how changes in one causes changes in the other.
For example, if we add the sentence

$\exists x(Cube(x) \wedge BackOf(x,a))$

which means "there exists an x such that x is a cube and x is behind a" to the description in figure 2.2 then, given the same situation as given in figure 2.1, we find that we get a cross against our guess that this sentence is true (see figure 2.3).

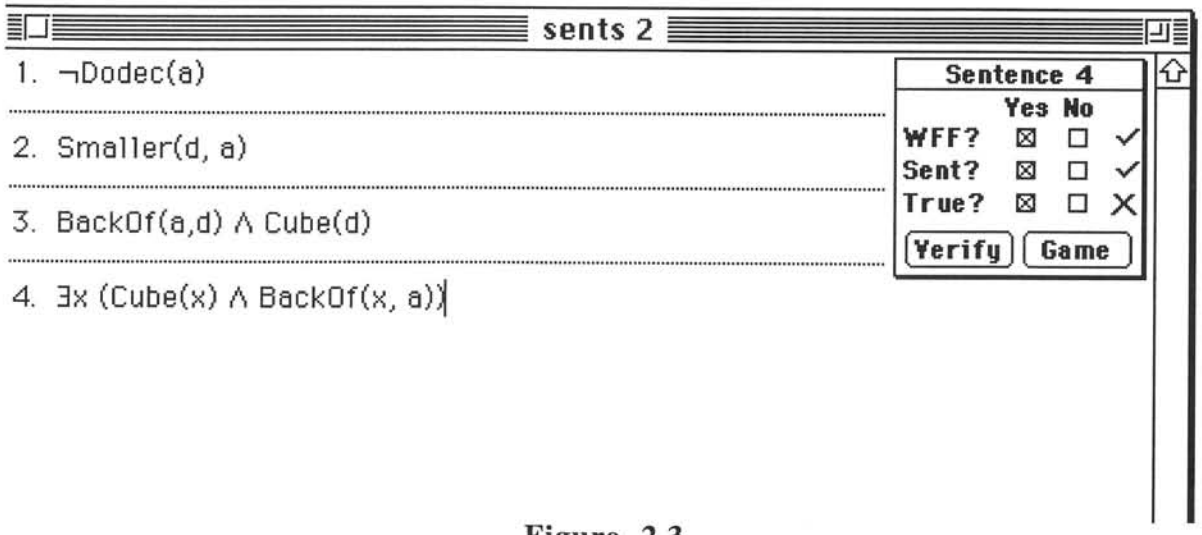

Figure 2.3

If we cannot see why our guess was wrong, i.e. why the sentence is false, we can elect to play the game by pressing the appropriate button. If we do this we are driven, by a series of interaction with TW, to see exactly why our guess was wrong, as figure 2.4 shows.

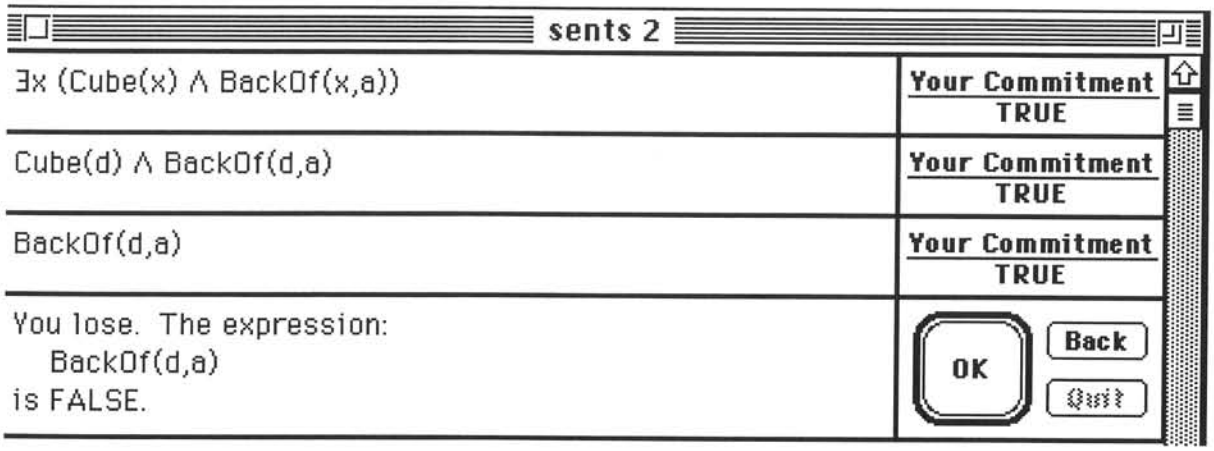

Figure 2.4

As we can see, TW gradually homes-in on the reason why the user's guess was wrong and leaves the user in no doubt about the incorrectness of their guess at the truth-value. So, the game allows the user to see exactly why they were wrong and clearly adds a great deal to the task of learning the language of first-order logic.

However, what the user finally gets, to a greater or lesser degree (where the degree depends on how precisely the description determines exactly what the situation looks like), is a formal description of what the situation, a picture, looks like.

3. Modelling a GUI

The idea taken from TW is that of describing, textually and formally, what a situation that is given graphically looks like. In this section we build on this idea and develop a design for a system that allows us to build, graphically, a display and then gradually build-up a formal description of that display, sentence by sentence.

At each addition of a new sentence to the formal description we can check that the sentence correctly describes part of the display by checking that what we have said in the sentence is true when we view the display as a situation, as in TW. If during the building of this description we cannot see why a sentence we have added is not true (that is, it does not correctly add to the description of the display), then the system can guide us to an explanation of why that sentence of the description is not true of the display by using the mechanism of the game.

The system we propose allows a designer to experiment with the look of a display and then go on to develop a formal description of it. The system can be used to check at each stage that the

description really is describing the display constructed by allowing the designer to check that all of the description's sentences are true of that display.

Allowing experimentation with the look and description of a display are an important part of this use of such a system - the fact that such an open-ended, relatively unconstrained exploration can result in a formal description is the main goal of this work.

Later, during the process of programming the interface, the existence of this display and its formal description will allow the software engineer to strengthen the specification if necessary. This might arise if the software engineer is not able to prove that a certain piece of code works correctly from the current specifications, but can for stronger ones. They can then go back to the display and its description and use the system to show that adding sentences to the description, to get a stronger one, still results in all the sentences being true, so that the stronger description can be used to further the specification process.

It also allows us to go the other way: given a formal description, within the system you can build up, checking correctness all along, a graphical representation of the sentences, i.e. a display that the description correct formalises. This would be an approach used when, for example, animating a specification for a user or client who need not understand the formal description.

Instead of the objects and their various attributes and relationships present in TW we would have windows, menus and buttons of various sizes and with their own appropriate attributes and relationships. Their positions would be modelled just as in Tarski's World, as would, say, the text written on them, the procedures or methods that were connected to them etc.

Below is given an example of how such a description building enterprise might look. Figure 3.1 is a menu that we want to specify. In an analogy with TW, we can imagine the menu expressed in its component parts as in figure 3.2. By viewing the situation in 3.2 from the front we see the menu as required in 3.1.

We can then go on to build a description of the situation in 3.2, as given in figure 3.3. Notice that, as we want with specifications, the description is an abstract form of the menu in the sense that some things are left out; we do not need to give all of the detail (like exact positions) of the menu in order to usefully specify it. Later, of course, during the refinement process (the process that takes us from a specification to an implementation) a programmer will have to be specific about such things. However, the point of specification is that we can leave out any unnecessary detail, i.e. perform abstraction, in order to see clearly what is being asked for. In particular, we do not have to say how the menu is drawn, just what it looks like.

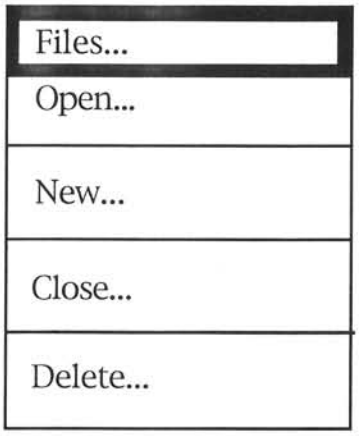

Figure 3.1

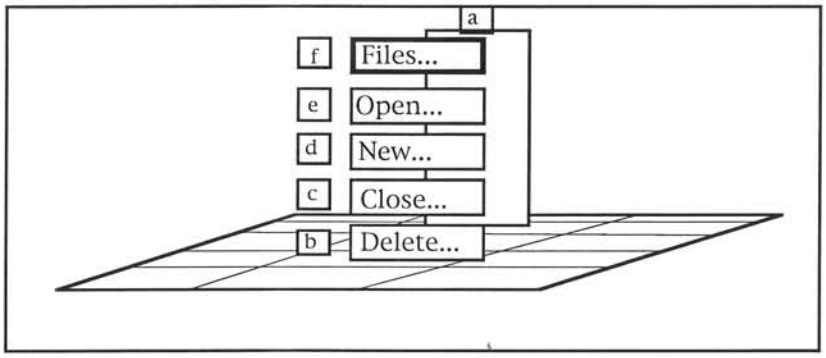

Figure 3.2

1. FrontOf(b,a) /\ Text(Files...)
2. Above(c,b) /\ Text(Delete...)
3. Above(d,c) /\ Text(Close...)
4. Above(e,d) /\ Text(New...)
5. Above(f,e) /\ Text(Open...)

Figure 3.3

In the simple examples here we have confined ourselves to simple relations between objects which form a display (FrontOf, Text); clearly we will wish to have available other relations, such as those that for a button, say, describing not just its physical attributes but the way it is linked with the program being built.

What we have here, then, is a way of formally and textually, describing the look of a display. This is done at the same level of abstraction as we would usually want to work when

specifying systems. The fact that it is textual allows us to use it with the sorts of methods discussed in section one. The fact that it is formal allows us to reason (either within or outside the sorts of systems mentioned in section one) about the look of the display. The display is now, formally, a first-class citizen.

Using this system we can bridge the gap we identified in section one. Now, the system we have proposed here, together with the techniques for formalising dialogues and other aspects of interaction, allows us to give a completely formal description of a GUI, and hence, because we already have ways of formally specifying the function of a system, of a complete system

4. Conclusions

We have proposed a way of helping a designer build-up a formal, textual description of a display. The next step is to implement a prototype of a system which performs this function in much the same way as TW does for designers of blocks worlds. This would allow us to build and experiment with, in a natural way, formal descriptions of displays for GUIs.

Since we have the role model of TW to follow we can be certain that such a system can be implemented, which is our current task in this research. When we have finished the implementation, of course, we will then have to consider the limitations of the system and experiment to see how the system can be strengthened to overcome them.

One point on which we are undecided is whether or not to allow relations which constrain objects to be in certain definite positions, for example at the top, left-hand corner of the display. We are undecided about this because, on the one hand, we want our displays to be as general as possible so that the programmer is free to use his or her creative talents as fully as possible. On the other hand, we want our descriptions to be strong enough to allow the specifier to be definite about things like the exact positions of menus and buttons, say, when that is appropriate. Of course, going too far down this path leads to the sort of very-low level descriptions that we are trying to avoid.

In the end it is likely that the system should be configurable so that, according to the requirements of the software team using it, the descriptions will have just the right strength.

References

[Al] Alexander, H., Structuring Dialogues using CSP, in [Ha1].

[Ba] Barwise, J., Etchemendy, J. *The Language of First-Order Logic*, Center for the Study of Language and Information.

[Di1] Diller, A. *Z An Introduction to Formal Methods*. Wiley. 2nd. ed. 1994.

[Di2] Dix, A., Finlay, J., Abowd, G., Beale, R., *Human-Computer Interaction*. Prentice Hall. 1993.

[Dr] Dromey, G. *Program Derivation The Development of Programs from Specifications*. Addison-Wesley. 1989.

[Go1] Goldson, D., Reeves, S., Bornat, R. A Review of several systems for the Support of Logics, The Computer Journal, vol. 36, no. 4. 1993.

[Go2] Goldson, D., Reeves, S., Review of "The Language of First-order Logic" by Barwise and Etchemendy, in The Philosophical Quarterly, Blackwells, Oxford, April 1994.

[Gr] Gries, D. *The Science of Programming*. Springer-Verlag. 1981.

[Ha1] Harrison, M., Thimbleby H. (editors), Formal Methods in Human-Computer Interaction, Cambridge University Press, 1990.

[Ha2] Harrison, M., Dix A., A State Model of Direct manipulation in Interactive Systems, in [Ha1].

Two Approaches to Formal Program Development

Doug Goldson

Department of Computer Science
Massey University, Private Bag 11222,
Palmerston North, New Zealand
D.Goldson@massey.ac.nz

Abstract

Two approaches to formal program development are illustrated with a worked example. One approach uses the specification language Z, the other uses a sugared functional programming language. The implementation language is C. Z is a recognised notation for program development and is used as a standard against which the functional specification can be compared.

1 Introduction

The belief which underlies this paper is that modern functional programming systems are sufficiently abstract that, in many cases, they can be used as systems for the specification and design of programs. The advantage of doing this is that a single formal system is used for program specification, design and prototyping and this goes some way to allaying criticism that formal specification is too complicated to be of practical value.

Opinion about how programs should be specified is wide ranging but there is common agreement that it has to do with saying what a program should do and not how it should it. Because functional systems have powerful mechanisms for data and operational abstraction, this level of abstraction can often be achieved. Functional, procedural and data abstraction are widely recognised design techniques and widely used in program design methods. Less well known in the mainstream software engineering community, perhaps, is the value of functional systems as design tools which support these techniques. Modern functional systems have mechanisms for enforcing data abstraction by hiding the implementation of data types as well as the

ability to handle functions as first-class data objects allowing higher-order functions (functionals) to be used for operational abstraction [2].

Landin used these techniques in the 1980's to develop a characteristically imaginative approach to teaching program design [3] and the functional specifications described in §3 are derived from these ideas but recast in an explicitly functional framework. Functional programmers will make the obvious connections with well-known notations and techniques such as comprehensions and folding [1].

The purpose of this paper is to advocate the use of this style of specification and design by making a favourable comparison with the specification language Z which is widely used [4].

2 An Example Specification in Z

I begin with an example of formal progam development from Spivey who uses it as a tutorial introduction to Z in [4]. The example is a birthday book which records the birthdays of friends and issues reminders on request. The basic sets (types) are `NAME` and `DATE` and a state of the book is given by the schema

```
BirthdayBook------------
known : P NAME
birthday : NAME -|-> DATE
------------------------
known = dom birthday
```

Here, `BirthdayBook` is the name of the schema, the next two lines are variable declarations (`P` is the powerset operator and "`-|->`" says that `birthday` is a partial function) and the last line is a state invariant: the people who are known are those with their names in the book (`known` is the domain of the function `birthday`).

Z uses many notational conventions of which this two-dimensional representation of a schema is an example. The `BirthdayBook` is defined one-dimensionally as

```
BirthdayBook =^= [known : P NAME; birthday : NAME -|-> DATE |
                  known = dom birthday]
```

Another convention primes the variables of schemas which describe after-states. Thus

```
BirthdayBook'------------
known' : P NAME
birthday' : NAME -|-> DATE
------------------------
known' = dom birthday'
```

represents the state of the book after an operation has been applied to it.

The operations that apply to the book, and may change its state, are `AddBirthday`, `FindBirthday` and `Remind`. `AddBirthday` is specified

```
AddBirthday----------
ΔBirthdayBook
name? : NAME
date? : DATE
-------------
name? ∉ known
birthday' = birthday ∪ {name? |-> date?}
```

The last line specifies the operation (`name? |-> date?` is an ordered pair), the second-to-last, its precondition. Two further Z conventions are illustrated. Input variables are decorated with `?` (output variables with `!`) and a destructive operation, which changes the state, represents the before and after states of the change as `ΔBirthdayBook`. This schema is therefore a shorthand for

```
AddBirthday----------
known, known' : P NAME
birthday, birthday' : NAME -|-> DATE
name? : NAME
date? : DATE
--------------------
known = dom birthday
known' = dom birthday'
name? ∉ known
birthday' = birthday ∪ {name? |-> date?}
```

Horizontally, this is

```
AddBirthday =^=
BirthdayBook & BirthdayBook' &
[name? : NAME; date? : DATE |
 name? ∉ known & birthday' = birthday ∪ {name? |-> date?}]
```

Or, in full

```
AddBirthday =^=
[known, known' : P NAME; birthday, birthday' : NAME -|->DATE;
 name? : NAME; date? : DATE |
 known = dom birthday & known' = dom birthday' & name? ∉ known &
 birthday' = birthday ∪ {name? |-> date?}]
```

FindBirthday is specified

```
FindBirthday-----------
ΞBirthdayBook
name? : NAME
date! : DATE
-------------
name? ∈ known
date! = birthday(name?)
```

The last convention ΞBirthdayBook indicates that the operation is non-destructive and no change of state occurs. The schema is a shorthand for

```
FindBirthday-----------
ΔBirthdayBook
name? : NAME
date! : DATE
-------------
name? ∈ known
date! = birthday(name?)
known = known'
birthday = birthday'
```

The final operation is Remind

```
Remind-----------
ΞBirthdayBook
today? : DATE
cards! : P NAME
----------------------------------------
cards! = {n : known | birthday(n) = today?}
```

One more schema gives the initial state of the book

```
InitBirthdayBook-----------
BirthdayBook
------------
known = {}
```

2.1 Program Development (Refinement)

In Spivey's refined specification, two arrays, names and dates, for entries in the book, are modelled as functions. hwm is a counter which marks the next available array cell. A concrete description of the birthday book is therefore

```
BirthdayBook1-----------
names : N1 -> NAME
dates : N1 -> DATE
hwm : N
-----------------------------------------
Λi,j : 1..hwm • i ≠ j => names(i) ≠ names(j)
```

BirthdayBook1 tells us that names does not contain duplicates.

The concrete versions of AddBirthday and FindBirthday are

```
AddBirthday1-----------
ΔBirthdayBook1
name? : NAME
date? : DATE
------------------------------
name? ∉ {i : 1..hwm • names(i)}
hwm' = hwm + 1
names' = names ⊕ {hwm' |-> name?}
dates' = names ⊕ {hwm' |-> date?}

FindBirthday1-----------
ΞBirthdayBook1
name? : NAME
date! : DATE
-----------------------------------------
∃i : 1..hwm • name? = names(i) & date! = dates(i)
```

"f ⊕ g" denotes a function which behaves like g on g's domain, otherwise like f.

The output of Remind is a set which is imitated by an array and an integer.

```
Remind1-----------
ΞBirthdayBook1
today? : DATE
cardlist! : N1 -> NAME
ncards! : N
--------------------------------
{i : 1..ncards! • cardlist!(i)} =
{j : 1..hwm | dates(j) = today? • names(j)}
```

Finally, the initial state of this representation of the book is

```
InitBirthdayBook1-----------
BirthdayBook1
------------
hwm = 0
```

indicating that both arrays are empty.

2.2 The Program

With a bit more effort a C program can be 'read off' this specification.

```
#include <string.h>
#define SIZE 11

typedef char *String;

int hwm = 0;
String names[SIZE];
String dates[SIZE];

void addBirthday(String name, String date)
{
     hwm = hwm+1;
     names[hwm] = name;
     dates[hwm] = date;
}

String findBirthday(String name)
{
     int i;
     for (i=1; strcmp(names[i], name); i = i+1)
         ;
     return dates[i];
}

int remind(String today, String cardlist[])
{
     int i, ncards;
     ncards = 0;
     for (i = 1; i <= hwm; i = i+1)
         if (!strcmp(dates[i], today)) {
             ncards = ncards+1;
             cardlist[ncards] = names[i];
         }
     return ncards;
}
```

2.3 Program Correctness

The tie-up in [4] between the two specifications of the book is made by two abstraction schemas `Abs` and `AbsCards`. The first describes the relation between `BirthdayBook` and `BirthdayBook1` and shows how the two arrays model the `birthday` function.

```
Abs-----------
BirthdayBook
BirthdayBook1
------------
known = {i : 1..hwm • names(i)}
Λi : 1..hwm • birthday(names(i)) = dates(i)
```

The first 'predicate' says that the domain of the function is just the defined portion of `names`, the second says that the value of the function at `names(i)` is `dates(i)`.

`Remind` and `Remind1` are linked with a second abstraction schema because sets are not a basic datatype.

```
AbsCards-----------
cards : P NAME
cardlist : N1 -> NAME
ncards : N
-------------------------------------
cards = {i : 1..ncards • cardlist(i)}
```

Equational reasoning is used to show that the concrete specification is a correct implementation of the abstract one. That is, corresponding schemas define equal outputs for equal inputs, or, more generally, have the same effects for the same inputs. In the case of `Remind` and `Remind1`, we have that

```
{n : known | birthday(n) = today?}                          =
{n : {i : 1..hwm • names(i)}| birthday(n) = today?}         =
{i : 1..hwm | birthday(names(i)) = today? • names(i)}       =
{i : 1..hwm | dates(i) = today? • names(i)}
```

3 Functional Specification

I begin with some of the underlying ideas of specification in the style I am advocating before repeating Spivey's example as an illustration of this style. Landin coined the word "fordo" to describe a kind of iterative operation over composite data objects [3]. In general terms, a fordo specification can be written

```
(\i -> e, v, x in_ob s)
```

where `e`, `v` and `s` are expressions and `i` and `x` are distinct variables, `x` binds its free occurrences in `e` and `i` may not occur free in `s`. `v` is the initial value of the specification, `s` is a composite object and `in_ob` is a generator for elements `x` in `s`. The value of the specification is got by repeatedly applying the function `\i -> e` to an accumulated result, starting with the initial value `v`, until `s` is empty.

The meaning becomes clearer with an example. Suppose `s` is a set, `v` is `{}`, and the function is `\i -> extend e i`, then we have a set comprehension

```
{e | x in_set s}                                    =
(\i -> extend e i, {}, x in_set s)
```

Filters and loop terminators are introduced in a straightforward way, using "∈" for in_set

```
{e | x ∈ s, b}                                      =
(\i -> extend e i, {}, (x ∈ s, b))                  =
(\i -> if b then extend e i else i, {}, x ∈ s)

(∃x ∈ s) b                                          =
(\_ -> b, False, x ∈ s until b)
```

And in combination

```
(∃x ∈ s) b                                          =
(\_ -> True, False, (x ∈ s until b, b))
```

The use of mathematical shorthand (set comprehensions and quantifiers) for fordos results in a notation for specification not too far removed from Z. In addition to this, we also have, at least implicitly, a computational interpretation of this notation. In fact, the relationship to high-level programs (program designs) which meet these specifications is very direct. It is made explicit in a simple program calculus of for-loop introduction and elimination rules which tie fordo specifications to equivalent imperative and functional designs. In general terms, these rules are

```
(\i -> e, v, x in_ob s)                             =
i = v; for (x in_ob s)
           i = e;
       return i;                                    =
for_ob (\x i -> e) v s

(\i -> e, v, (x in_ob s, b))                        =
i = v; for (x in_ob s)
           if (b) i = e;
       return i;                                    =
for_ob (\x i -> if b then e else i) v s

(\i -> e, v, x in_ob s until b)                     =
i = v; for (x in_ob s)
           if (b) return e; else i = e;
       return i;                                    =
foruntil_ob (\x i -> (b,e)) v s
```

```
(\i -> e, v, (x in_ob s until b, b'))          =
i = v; for (x in_ob s)
           if (b') if (b) return e; else i = e
       return i;                                =
foruntil_ob (\x i -> (b, if b' then e else i)) v s
```

There is a proviso on the introduction rule that `i` doesn't occur free in `x`, `s`, `b` or `b'`. `=` is an assignment operator and an assignment `i = e` has value `e`. `for_ob` and `foruntil_ob` are iterative higher-order functions which repeatedly apply their functional argument to elements of `s` and an accumulated result until `s` is empty or, in the case of `foruntil_ob`, the function returns a result which indicates that the iteration may be terminated. `v` is the initial value of the accumulator. The types of `for_ob` and `foruntil_ob` are

```
(tx -> tv -> tv) -> tv -> ts -> tv
(tx -> tv -> (Bool,tv)) -> tv -> ts -> tv
```

where `\x i -> e` and `\x i -> (b,e)` have types `tx -> tv -> tv` and `tx -> tv -> (Bool,tv)` and `v` and `s` have types `tv` and `ts`. Their definition depends on how the datatype `ts` is imitated (see §3.2.1).

Designs for `{e | x ∈ s, b}` and `(∃x ∈ s)b` using the for-loop rules are obtained as follows

```
(\i -> extend e i, {}, (x ∈ s, b))                =
i = {}; for (x ∈ s)
           if (b) i = extend e i;
       return i;                                  =
for_set (\x i -> if b then extend e i else i) {} s

(\_ -> b, False, x ∈ s until b)                   =
i = False; for (x ∈ s)
               if (b) return b; else i = b;
           return i;                              =
i = False; for (x ∈ s) {
               i = b;
               if (i) return i;
           } return i;                            =
foruntil_set (\x _ -> let b' = b in (b', b')) False s
```

3.1 The Birthday Book

Z is a typed set theory, so, to make a direct comparison with Z, we give a fordo specification of the birthday book using sets. First of all, we need a type of sets

```
interface Set where
data Set a
     empty    :: Set a
     extend   :: a -> Set a -> Set a
     for_set  :: (a -> b -> b) -> b -> Set a -> b
     foruntil_set :: (a -> b -> (Bool,b)) -> b -> Set a -> b
```

Assume that this abstract support for the book is implemented and its operations are augmented as required (`member`, `union`, ...).

Suppose the state of the book is represented as a pair `(known, birthday)` where `known`, the domain of `birthday`, is a set and the object-level 'function' is a set of pairs, then the operations on the book are defined as functions `addBirthday`, `findBirthday` and `remind`.

```
addBirthday (known,birthday) name date =
     if not (member name known) then
        (known ∪ {name}, birthday ∪ {(name,date)})
     else ⊥

findBirthday (known,birthday) name =
     if member name known then
        (\_ -> d, ⊥, ((n,d) ∈ birthday, n == name))
     else ⊥

remind (_,birthday) today = {n | (n,d) ∈ birthday, d == today}
```

The two components of the specification are the book `(known,birthday)` and the operations on the book: `addBirthday`, `findBirthday` and `remind`. I have made their definitions agree as closely as possible with the counterpart Z schemas so that a schema which defines an operation on a state is modelled by a function from state to result (which is another state in the case of the state-changer `addBirthday`).

3.2 Program Development

In Z, a program is reached by following a path of successive refinements of the initial specification. To some extent this process can be imitated in the functional style by considering more 'efficient' specifications. In the present case by refining the definitions of `addBirthday`, `findBirthday` and `remind`. An obvious change is to do away with `known` since this is just the domain of `birthday`. This gives a number of alternatives for `addBirthday`

```
     if not (member name (dom birthday)) then
        birthday ∪ {(name,date)}
     else ⊥
```

```
if not (member name (dom birthday)) then
   extend (name,date) birthday
else ⊥

if not ((∃(n,_) ∈ birthday) name == n) then
   extend (name,date) birthday
else ⊥

if (Λ(n,_) ∈ birthday) name /= n then
   extend (name,date) birthday
else ⊥

extend (name,date)
       (\_ -> ⊥, birthday,
        ((n,_) ∈ birthday until name == n, name == n))
```

`findBirthday` is changed in the same way

```
findBirthday birthday name =
       (\_ -> d, ⊥, ((n,d) ∈ birthday until n == name, n == name))
```

The fordo in `findBirthday` is replaced by a fordountil; `birthday` is a function so once we find an entry for `name` we can stop looking. Like the Z description of §2.1, these definitions are starting to look like a program. The next step is to explicitly infer program designs.

3.2.1 A Functional Prototype

A functional design is obtained using the for-loop rules

```
addBirthday birthday name date =
    extend (name,date)
           (foruntil_set (\(n,_) i -> let b = name == n in
                                      (b, if b then ⊥ else i))
                         birthday birthday)

findBirthday birthday name =
    foruntil_set (\(n,d) i -> let b = name == n in
                              (b, if b then d else i))
                 ⊥ birthday

remind birthday today =
    for_set (\(n,d) i -> if d == today then extend n i else i)
            {} birthday
```

All that is now missing is an implementation of sets which is the counterpart of the abstraction schema of §2.3. The simplest is to treat a set as a duplicate-free list.

```
module Set where
type Set a = [a]
...
empty = []
extend x l = if (∃y <- l) x == y then l else x:l
fordo_set = fold
fordountil_set = folduntil
```

`fold` and `folduntil` are defined

```
fold f v [] = v
fold f v (x:xs) = fold f (f x v) xs

folduntil f v []     = v
folduntil f v (x:xs) = case f x v of
                           (True, v') -> v'
                           (False,v') -> folduntil f v' xs
```

Equivalent definitions of `extend` are

```
extend x l = if folduntil (\y i -> let b = x == y in (b,b)) False l
             then l else x:l

extend x [] = [x]
extend x (y:ys) | x == y = y:ys
                | True   = y:extend x ys
```

3.2.2 The Program

A functional design offers the advantage of early prototyping. An imperative design offers a half-way point of reference between the specification and its implementation. The for-loop rules give

```
addBirthday birthday name date =
extend (name,date) {i = birthday; for ((n,_) ∈ birthday)
                                      if (name == n) return ⊥;
                                  return i;}

findBirthday birthday name =
    i = ⊥; for ((n,d) ∈ birthday)
               if (name == n) return d;
           return i;

remind birthday today =
    i = {}; for ((n,d) ∈ birthday)
                if (d == today) i = extend n i;
            return i;
```

Neither lists nor tuples are basic types in C so more imitation of datatypes is required than in the relatively straightforward translation into the functional prototype. This is to be expected. The first step to making the design C-fit, as in the prototype, is to replace sets with lists, then to replace for-loops with while-loops.

```
i = v; for (x <- l)
           i = e;
       return i;              =
i = v; for (;l /= [];) {
           x = hd l;
           l = tl l;
           i = e;
       } return i;
```

This gives

```
addBirthday birthday name date =
    extend (name,date) {i = birthday; for (;birthday /= [];) {
                                          (n,_)     = hd birthday;
                                          birthday = tl birthday;
                                          if (name == n) return ⊥;
                                      } return i}
```

The next step eliminates the pattern-matching assignment and moves the application of `extend` inside the block.

```
addBirthday birthday name date =
    i = birthday; for (;birthday /= [];) {
                        n  = fst (hd birthday);
                        birthday = tl birthday;
                        if (name == n) return ⊥;
                    } return extend (name,date) i
```

Notice that, in this case, `extend` can be replaced by `:`. The rest is just transliteration.

```
#include <string.h>  /* for strcmp */
#include "types.h"   /* for Book, String, Pair, List */
#include "list.h"    /* for cons, hd, tl */
#include "pair.h"    /* for fst, snd, pair */
```

```
Book addBirthday(String name, String date, Book birthday)
{
    String n;
    Book book = birthday;
    for (;!isempty(birthday);) {
        n = fst(hd(birthday));
        birthday = tl(birthday);
        if (!strcmp(name,n)) return;
    };
    return cons(pair(name,date),book);
}
String findBirthday(String name, Book birthday)
{
     Pair p;
     Book book = birthday;
     for (;!isempty(birthday);) {
        p = hd(birthday);
        birthday = tl(birthday);
        if (!strcmp(name,fst(p))) return snd(p);
     }
}
List remind(String today, Book birthday)
{
     Pair p;
     List rems = empty();
     for (;!isempty(birthday);) {
        p = hd(birthday);
        birthday = tl(birthday);
        if (!strcmp(today,snd(p))) rems = cons(fst(p),rems);
     };
     return rems;
}
```

4 Catching Errors

The specifications of §2 and §3 do not say what should happen in the case of erroneous input when the preconditions of `addBirthday` and `findBirthday` are not met. The programs of §2.2 and §3.2.2 are underspecified.

In what circumstances will these programs fail? The array version of `addBirthday` illustrates two kinds of failure. The program limits the size of the book to only 10 entries. What happens when we try to add an 11th? The `AddBirthday` schema needs to be strengthened by adding `#birthday < 10` as a precondition and defining program behaviour when this condition is not met. A second kind of failure

arises when the precondition `name? ∉ known` is not met. Again, the specification must define program behaviour when this condition fails.

Spivey addresses the second of these limitations by strengthening the Z specification in the following way

```
Success---------
result! : STRING
--------------
result! = "Ok"

AlreadyKnown---------
ΞBirthdayBook
name? : NAME
result! : STRING
----------------
name? ∈ known
result! = "Already known"

NotKnown---------
ΞBirthdayBook
name? : NAME
result! : STRING
----------------
name? ∉ known
result! = "Not known"
```

The new schemas are then combined with those of §2

```
RAddBirthday =^= AddBirthday & Success ∨ AlreadyKnown
RFindBirthday =^= FindBirthday & Success ∨ NotKnown
RRemind =^= Remind & Success
```

Expanded out, `RFindBirthday` looks like

```
RFindBirthday-----------
ΞBirthdayBook
name? : NAME
date! : DATE
result! : STRING
-------------
(name? ∈ known &
date! = birthday(name?) &
result! = "Ok") ∨
(name? ∉ known &
result! = "Not known")
```

The functional specification can be strengthened in a similar way.

```
data Result a = Success a | Failure String

findBirthday birthday name =
    (\_ -> Success d, Failure "Not known",
     ((n,d) ∈ birthday until n == name, n == name))

addBirthday birthday name date =
    if (Λ(n,_) ∈ birthday) name /= n then
        Success extend (name,date) birthday
    else Failure "Already known"
```

Or perhaps

```
addBirthday birthday name date =
    case (\_ -> Failure ⊥, Success ⊥,
          ((n,_) ∈ birthday until name == n, name == n))
    of
    Success _ -> Success extend (name,date) birthday
            _ -> Failure "Already known"
```

5 Conclusion

The phrase "program specification" has a range of different interpretations. One interpretation treats a specification as an abstract or 'high-level' solution to a programming task which is not constrained by the limitations of an implemented programming language. The specification language should be expressive, flexible and extensible to allow the programmer to express their thoughts unhindered and it should be precise and unambiguous so that gaps and mistakes become apparent.

The conclusion, based on one small example!, is that the fordo style of specification described in §3 compares well with Z in terms of this notion of specification and, to the extent that it is a "broad spectrum" approach, it is more flexible. It offers a framework for reasoning about functional as well as imperative programs which Z's state-based, pre- and post-condition approach seems to lack.

References

[1] Bird R, Wadler P, Introduction to Functional Programming, Prentice Hall, 1988.

[2] Hughes J, "Why Functional Programming Matters" in The Computer Journal, Vol 32, No 2, 1989.

[3] Landin P, Programming 2 Lecture Notes, 1987-88, esp. ch3, Department of Computer Science, Queen Mary and Westfield College, University of London, London E1 4NS.

[4] Spivey J M, The Z Notation: A Reference Manual, Prentice-Hall, 1989.

Deriving a Predictive Parsing Algorithm

Lindsay Groves

Department of Computer Science
Victoria University of Wellington
Wellington, New Zealand
lindsay@comp.vuw.ac.nz

Abstract

We show how a predictive parsing algorithm can be derived by first deriving a generic language recognition algorithm, and then specialising this to a class of context-free languages. This strategy allows us to work out the basic structure of the algorithm, based on very general properties of the specification, and then develop more intricate parts of the algorithm and introduce suitable data representations afterwards. The resulting derivation is more intelligible than if we considered all aspects of the specification from the outset, and allows the derivation to be more easily adapted to handle other classes of languages.

1 Introduction

One of the major problems in software development, formal or otherwise, is that of controlling complexity. In order to make formal software development feasible for anything other than toy problems, we need to find ways of structuring the development so as to control complexity.

One strategy for controlling complexity in software development is the use of data abstraction. We initially ignore details of data representation and derive an abstract program which does not depend on any particular data representation; we then introduce the details of a particular data representation, and perhaps make various optimisations admitted by the chosen representation. This process has been studied extensively and has become a major part of modern software design methods. The use of data abstraction has been formalised within the refinement calculus as *data refinement* (e.g. [Morgan and Gardiner, 1988], [Morris, 1989]).

Another strategy for controlling complexity is to begin by considering a *generalisation* of the problem, obtained by ignoring some of the requirements; in particular we can ignore some of the assumptions about the inputs. We can then develop software for the generalised problem, and later *specialise* it to address the omitted requirements. Making use of these additional assumptions often allows the software to be made more efficient — in particular it may allow data refinements that were not previously possible — and may make it feasible.

The use of program specialisation in the context of program refinement has been described by Gravell [Gravell, 1991], who shows that a number of different search algorithms and some arithmetic algorithms can be obtained by specialising a generic search algorithm. In [Groves, 1994a], we illustrated the use of program specialisation in tandem with data refinement to derive a family of language recognition algorithms by deriving a generic language recognition algorithm, and then specialising it obtain algorithms for some specific classes of languages.

In this paper we again derive a generic language recognition, then specialise it to obtain a table-driven predictive parsing algorithm for a class of context free languages. The generic algorithm differs from the one in [Groves, 1994a] in a number of ways: the notation used and the derivation are (hopefully) more elegant, and the resulting algorithm is more general. The derivation of the predictive parsing algorithm is presented in more detail than in [Groves, 1994a], identifying more carefully the properties on which the algorithm depends.

We begin by formulating the language recognition problem in a very general way (Section 2) and deriving a generic language recognition algorithm (Section 3). In Section 4 we specialise the algorithm to recognise a class of context free languages and introduce suitable data refinements to obtain a concrete algorithm. In Section 5 we present our conclusions.

The paper assumes some familiarity with the refinement calculus, as given, for example in [Morgan, 1990] or [Groves, 1994b]. We also assume standard terminology and definitions from language theory (see, for example, [Hopcroft and Ullman, 1979]).

2 Defining the problem

We begin by considering the general problem of recognising sentences in a given language, making no assumptions about the kind of language being recognised. We will derive a generic algorithm for this generalised version of the problem, and then specialise it to a particular class of languages. The generic algorithm, however, provides the basic structure of the specialised algorithm, and can be similarly specialised to provide recognisers for other classes of languages.

We initially assume only that we are given some alphabet $\mathcal{A}$ (i.e. a finite set of symbols), a string s over $\mathcal{A}$ and a language L over $\mathcal{A}$, and we require to determine whether s is a sentence in L. We can formalise this as a specification statement [Morgan, 1990]. A specification statement $\tilde{w}: \left[\mathit{Pre} \;/\; \mathit{Post} \right]$ will establish the postcondition Post altering only variables in $\tilde{w}$, provided the precondition Pre is true initially. If we return the result as a boolean variable, r, we can express our specification as:

$$r: \left[s \in \mathcal{A}^* \;\wedge\; L \subseteq \mathcal{A}^* \;/\; r \equiv s \in L \right] \tag{1}$$

We will assume that string comparison is not a primitive operation, so we are not able to simply compare s with each string in L — in any case, L may not be finite. Instead, we will inspect one symbol of s at a time, and use that to successively narrow down the part of L in which s could occur. We will follow standard convention and inspect symbols in s from left to right, though other choices are certainly possible.

3 Deriving a generic language recognition algorithm

We will now proceed to derive a generic language recognition algorithm to implement the above specification. In doing the derivation, we will need to introduce a number of concepts and notations relating to strings and languages, and give properties required in deriving the algorithm. We introduce some of these now, and present others as they are needed, so their introduction is clearly motivated.

Since our algorithm will consider prefixes of s, we need notation for the prefix relation; we also introduce a suffix relation here, since it will be required later. We write $p \preceq s$ to mean that p is a *prefix* of s, and $s \succeq p$ to mean that p is a *suffix* of s, defined as follows:

Definition $p \preceq s \;\widehat{=}\; (\exists u \bullet p \frown u = s)$
$s \succeq p \;\widehat{=}\; (\exists u \bullet u \frown p = s)$

As indicated above, the algorithm will look at symbols in s one at a time, from left to right. We observe that each symbol of s inspected effectively reduces the part of L in which s could occur. We capture this idea by introducing a function d which gives the *derivative* of a language with respect to a string[1], defined thus:

Definition For any string u and language L, the *derivative* of L with respect to u, written $d(u, L)$, is the set of all strings which, when right concatentated with u, give a sentence in L; i.e. $d(u, L) \;\widehat{=}\; \{ v \mid u \frown v \in L \}$, where $\frown$ is the string concatenation operator.

For the purposes of the abstract algorithm, we will consider d to be executable only when u is a singleton (i.e. $|u| = 1$).

The derivative operation has a number of important properties. The following theorem captures an essential relationship between prefixes and deriviatives.

Theorem For any strings u and v, $u \frown v$ is in L iff v is in $d(u, L)$; i.e. $u \frown v \in L \equiv v \in d(u, L)$.

This theorem has two important consequences:

Corollary 1 Taking $u = s$ and $v = \lambda$ (where λ is the empty string), we see that $s \in L$ is equivalent to $\lambda \in d(s, L)$.

Corollary 2 If $d(p, L) = \varnothing$ for any prefix p of s, then $d(q, L) = \varnothing$ for any longer prefix q of s. In particular, $d(s, L) = \varnothing$; i.e. $p \preceq s \wedge d(p, L) = \varnothing \Rightarrow d(s, L) = \varnothing$.

The first corollary shows that we can determine whether s is in L by computing $d(s, L)$ and checking whether it contains λ. The second shows that s is not in L if $d(p, L)$ is empty for any prefix p of s. Thus, once $d(p, L)$ is found to be empty, for some prefix p, there is no point in looking at further elements of s.

Together, these corollaries suggest that we can determine whether s is in L by looking for a prefix p of s such that either $\lambda \in d(p, L)$ or $d(p, L) = \varnothing$[2]. Then, $s \in L$ iff $p = s$ and $\lambda \in d(p, L)$.

In fact, it turns out to be easier to consider progressively shorter suffixes of s. If t is a suffix of s, we write $s - t$ to mean the prefix of s remaining after suffix t is removed:

Definition For all strings u and v, $(u \frown v) - v = u$.

We can now substitute $s - t$ for p above[3]. If t is a suffix of s such that either $\lambda \in d(s - t, L)$ or $d(s - t, L) = \varnothing$, then $s \in L$ iff $t = \lambda$ and $\lambda \in d(s - t, L)$.

We can now refine the specification to introduce a variable t, for suffixes of s; we also introduce a variable C, for values of $d(s - t, L)$. At the same time, we split the specification statement into two parts: one to find a suffix t satisfying the above condition, and the other using this to determine the value of r. We will also omit the precondition and treat it a global assumption.

[1] This notion is motivated by the idea of derivatives of regular expressions [Brzozowski, 1964].

[2] We could insist that the program find the shortest such prefix, but it is not necessary to do so; our algorithm will in fact find the shortest such prefix.

[3] This could be treated formally as a data refinement, but there seems little point in doing so.

(1) $\sqsubseteq$ (*Introduce Local Variables, Split Specification, Contract Frame*)

var t, C •

t, C: $\left[s \succeq t \wedge C = d(s-t, L) \wedge (t = \lambda \vee C = \varnothing) \right]$; (2)

r: $\left[s \succeq t \wedge C = d(s-t, L) \wedge (t = \lambda \vee C = \varnothing) \;/\; r \equiv s \in L \right]$ (3)

We can immediately refine (3) to an assignment:

(3) $\sqsubseteq$ (*Introduce Assignment*)

$r := (t = \lambda \wedge \lambda \in C)$

The following proof obligation follows from corollaries 1 and 2:

$$s \succeq t \wedge C = d(s-t, L) \wedge (t = \lambda \vee C = \varnothing) \Rrightarrow ((t = \lambda \wedge \lambda \in C) \equiv s \in L)$$

3.1 Loop design

We now want to refine (2) to introduce a loop which computes $d(s-t, L)$ for decreasing suffixes t of s. Thus, we will take as our loop invariant:

$$Inv \;\widehat{=}\; s \succeq t \;\wedge\; C = d(s-t, L)$$

The postcondition of (2) suggests $t \neq \lambda \wedge C \neq \varnothing$ as the loop guard, and $|t|$ as the variant function.

We split (2) into two parts, one of which will initialise t and C so as to establish I, the other will be the loop.

(2) $\sqsubseteq$ (*Split Specification*)

t, C: $\left[Inv \right]$; (4)

t, C: $\left[Inv \;/\; Inv \wedge (t = \lambda \vee C = \varnothing) \right]$ (5)

We can easily establish Inv by establishing $t = s \;\wedge\; C = L$, since $s \succeq s$ and $L = d(\lambda, L)$.

(4) $\sqsubseteq$ (*Introduce Assignment*)

$t, C := s, L$

We now introduce the loop:

(5) $\sqsubseteq$ (*Introduce DO*)

do $t \neq \lambda \wedge C \neq \varnothing \rightarrow$

t, C: $\left[Inv \wedge t \neq \lambda \wedge C \neq \varnothing \;/\; Inv \wedge 0 \leq |t| \leq |t_0| \right]$ (6)

od

We further split (6) into two parts: one will update t to ensure that the loop makes progress; the other will update C to ensure that the invariant is maintained.

In order to "make progress" towards termination, the loop body must decrease $|t|$. Since t must always be a suffix of s, this can only be done by removing some prefix of t. In particular, we can decrease $|t|$ by one, by removing its head, i.e. a prefix of length one. We will assume the availability of two (executable) functions hd and tl such that, for any non-empty string t, $t = \mathsf{hd}\, t \frown \mathsf{tl}\, t$ and $|\mathsf{hd}\, t| = 1$. Thus, we wish to remove $\mathsf{hd}\, t$ from t, which can be done using the assignment statement $t := \mathsf{tl}\, t$.

We must now determine how to update C so as to maintain the invariant when t is updated like this. To see how C should be updated, we will refine (6) using the *Following Assignment* rule, and calculating the intermediate assertion:

$$\begin{aligned} & wp(t := \mathsf{tl}\, t,\ s \succeq t \ \wedge\ C = d(s - t, L) \wedge 0 \leq |t| \leq |t_0|\ ) \\ & \quad = s \succeq \mathsf{tl}\, t \ \wedge\ C = d(s - \mathsf{tl}\, t, L) \wedge 0 \leq |\mathsf{tl}\, t| \leq |t| \end{aligned}$$

We can drop the termination condition, since $t \neq \lambda \Rightarrow 0 \leq |\mathsf{tl}\, t| < |t|$ and $t \neq \lambda$ is given by the precondition.

We can also eliminate the condition $s \succeq \mathsf{tl}\, t$, which follows easily from the definitions:

$$\begin{aligned} & Inv \wedge t \neq \lambda \wedge C \neq \varnothing \\ & \quad \Rightarrow\ s \succeq t \wedge t \neq \lambda \\ & \quad \Rightarrow\ (\exists p \bullet s = p \frown t) \wedge t \neq \lambda \\ & \quad \Rightarrow\ (\exists p \bullet s = p \frown \mathsf{hd}\, t \frown \mathsf{tl}\, t) \\ & \quad \Rightarrow\ (\exists p' \bullet s = p' \frown \mathsf{tl}\, t) \\ & \quad \Rightarrow\ s \succeq \mathsf{tl}\, t \end{aligned}$$

This just leaves $C = d(s - \mathsf{tl}\, t, L)$ as the intermediate assertion:

$$\begin{aligned} (6) \quad \sqsubseteq \quad & (\textit{Following Assignment},\ \textit{Contract Frame}) \\ & C{:}\left[\, Inv \wedge t \neq \lambda \wedge C \neq \varnothing \ /\ C = d(s - \mathsf{tl}\, t, L)\,\right]; \qquad (7) \\ & t := \mathsf{tl}\, t \end{aligned}$$

In order to refine (7), we observe that $s - \mathsf{tl}\, t = (s - t) \frown \mathsf{hd}\, t$, and $d(x \frown y,\ L) = d(y,\ d(x,\ L))$, for all strings x and y, and any language L.

Thus, we get:

$$\begin{aligned} C &= d(s - \mathsf{tl}\, t, L) \\ &= d((s - t) \frown \mathsf{hd}\, t, L) \\ &= d(\mathsf{hd}\, t, d(s - t, L)) \end{aligned}$$

Since we have $C = d(s - t, L)$ in the precondition, we can refine (7) as follows:

$$\begin{aligned} (7) \quad \sqsubseteq \quad & (\textit{Introduce Assignment}) \\ & C := d(\mathsf{hd}\, t, C) \end{aligned}$$

The algorithm at this stage is:

$$
\begin{array}{l}
[\![\ \textbf{var}\ t, C \bullet \\
\quad t, C := s, L; \\
\quad \textbf{do}\ t \neq \lambda \wedge C \neq \varnothing \rightarrow \\
\qquad C := d(\mathsf{hd}\, t, C); \\
\qquad t := \mathsf{hd}\, t \\
\quad \textbf{od}; \\
\quad r := (t = \lambda \wedge \lambda \in C) \\
]\!]
\end{array}
$$

This algorithm is still not really executable because it is expressed in terms of the set variables L and C, and the function d which cannot be implemented for arbitrary sets. In order to obtain an executable program, we will have to place some restriction on L, and then explore suitable representations for L and C, and corresponding implementations for d. Obviously we won't be able to turn this into a concrete algorithm for all Ls: If L is not decidable, we won't be able to complete the data refinement.

4 Recognising Context Free Languages

In order to turn the above abstract algorithm into a more concrete form, we need to find some way to describe L — restricting the original specification to suit a particular descriptive formalism if necessary. Here we will consider the case where L is a non-empty context free language. In this case, we know that L can be described using a context free grammar, G, where G is a 4-tuple (V_N, V_T, S, P) giving the nonterminals, terminals, start symbol and productions, respectively (see, for example, [Hopcroft and Ullman, 1979]); since $L \subseteq \mathcal{A}^*$, we can assume $V_T = \mathcal{A}$. We assume that G is a "proper" grammar (i.e. all nonterminals are defined and there are no circular rules), and that G has no λ-rules (i.e. there are no rules with empty right-hand sides).

We specialise the initial specification with the additional assumption $L \neq \{\} \ \wedge \ L = \mathcal{L}(G)$, where $\mathcal{L}(G)$ is the language generated by G, defined as follows:

$$\mathcal{L}(G) \ \widehat{=} \ \{\, w \in V_T^* \mid S \Rightarrow_G^* w \,\}$$

and $\Rightarrow_G^*$ is the usual "produces" relation.

To specialise a program, we introduce additional assumptions about the inputs, and propagate these through the resulting derivation. This process is justified, since we can show that if $w{:}[\,P\,,\,R\,] \sqsubseteq w'{:}[\,P'\,,\,R'\,]$ then, for any additional assumptions A, we have $w{:}[\,P \wedge A\,,\,R\,] \sqsubseteq w'{:}[\,P' \wedge A\,,\,R'\,]$.

4.1 Representing derivatives

Having assumed that L is context free, we must also find a way to represent C, such that we can readily implement the required operations, especially $d(\mathsf{hd}\, t, C)$. For any context free language L and any string s, the derivative $d(x, L)$ is also context free. Thus, we could describe C using a context free grammar. In this case, the implementation of $d(\mathsf{hd}\, t, C)$ would need to construct a

new grammar; in particular, it would need to introduce new nonterminals and new rules, which promises to be rather messy.

An alternative approach is suggested by considering sentential forms, which can also be thought of as describing languages: perhaps we can represent C using a string of terminal and/or nonterminal symbols. It is not immediately clear whether we can always represent C in this way (it may depend on how we construct the algorithm), but it seems quite reasonable. So we will pursue this approach and see where it leads. We will ignore, for now, the question of how to represent the set of production rules, P.

To allow for the possibility that $C = \varnothing$, we introduce a special symbol ϕ, not in V_N or V_T, to cover this case (c.f. the use of ϕ in regular expressions). For any string $\sigma \in V^*$, where $V \mathrel{\widehat{=}} V_N \cup V_T \cup \{\phi\}$, we define $\mathcal{L}(\sigma)$ to be the set of all terminal strings that can be produced from σ:

$$\mathcal{L}(\sigma) \mathrel{\widehat{=}} \{\, w \in V_T^* \mid \sigma \Rightarrow_G^* w \,\}$$

Note that $\mathcal{L}(\lambda) = \{\lambda\}$, while $\mathcal{L}(\phi) = \varnothing$.

Clearly, L can be represented by the start symbol, S. We will endeavor to represent C by some string σ over V such that $C = \mathcal{L}(\sigma)$.

4.2 Data refinement

We can now data refine the generic algorithm using this data representation. Our coupling invariant will be:

$$CI \mathrel{\widehat{=}} L = \mathcal{L}(S) \wedge C = \mathcal{L}(\sigma) \wedge \sigma \in V^*$$

To perform the data refinement, we replace the declaration of C by a declaration of σ, then data refine the body. Since the coupling invariant is functional, this essentially involves replacing occurrences of L and C by $\mathcal{L}(S)$ and $\mathcal{L}(\sigma)$, respectively, and appending the representation invariant $\sigma \in V^*$ as appropriate ([Morgan and Gardiner, 1988], [Morris, 1989]). We then perform further refinements to remove occurrences of $\mathcal{L}$, since it is not executable. To simplify the presentation, we will treat $\sigma \in V^*$ as a global invariant and not mention it explicitly.

Data refining the body involves the following steps:

- Data refine the assignment $C := L$:

$$\begin{array}{lll} C := L & \equiv & C{:}\left[\, C = L \,\right] \\ & \preceq & \sigma{:}\left[\, \mathcal{L}(\sigma) = \mathcal{L}(S) \,\right] \\ & \sqsubseteq & \sigma{:}\left[\, \sigma = S \,\right] \\ & \sqsubseteq & \sigma := S \end{array}$$

- Replace the loop guard, $C \neq \varnothing$, by $\mathcal{L}(\sigma) \neq \varnothing$, which simplifies to $\sigma \neq \phi$.

- Data refine the statement $C := d(\mathsf{hd}\, t, C)$:

$$C := d(\mathsf{hd}\, t, C) \equiv C{:}\left[\, C = d(\mathsf{hd}\, t, C_0)\,\right]$$
$$\preceq\ \sigma{:}\left[\mathcal{L}(\sigma) = d(\mathsf{hd}\, t, \mathcal{L}(\sigma_0))\right] \qquad (i)$$

We will consider how to refine this below.

- Data refine the assignment $r := (t = \lambda \wedge \lambda \in C)$:

$$r := (t = \lambda \wedge \lambda \in C) \ \preceq\ r := (t = \lambda \wedge \lambda \in \mathcal{L}(\sigma))$$
$$\sqsubseteq\ r := (t = \lambda \wedge \sigma = \lambda)$$

The last simplification is possible because we have assumed that there are no λ-rules in G, so $\lambda \in \mathcal{L}(\sigma) \equiv \sigma = \lambda$.

4.3 Computing the derivative

We will now consider how to refine (i) so as to remove $\mathcal{L}$. We will do this by first introducing a new function d', such that $\mathcal{L}(d'(x,\sigma)) = d(x, \mathcal{L}(\sigma))$. We can thus rewrite (i) as:

$$\sigma{:}\left[\sigma = d'(\mathsf{hd}\, t, \sigma_0)\right] \qquad (i')$$

Our problem now is how to compute $d'(x,\sigma)$. We begin by observing that $d'(x,\sigma)$ is easy to compute when $\mathsf{hd}\,\sigma$ is not a nonterminal, since, for any string σ, and any terminals x and y:

$d'(x, x \frown \sigma') = \sigma'$,
$d'(x, y \frown \sigma') = \phi$ if $x \neq y$, and
$d'(\phi, \sigma) = \phi$

Thus, our first aim is to get σ into a form where $\mathsf{hd}\,\sigma$ is not a nonterminal. In doing this, we can modify σ in any way so long as $\mathcal{L}(d'(\mathsf{hd}\, t, \sigma))$ remains unchanged. We refine (i') to achieve this as follows:

$(i') \ \sqsubseteq$ (*Split Specification*)
$$\sigma{:}\left[\, d'(\mathsf{hd}\, t, \sigma) \cong d'(\mathsf{hd}\, t, \sigma_0) \ \wedge\ \mathsf{hd}\,\sigma \notin V_N \,\right]; \qquad (ii)$$
$$\sigma{:}\left[\mathsf{hd}\,\sigma \notin V_N \ /\ \sigma = d'(\mathsf{hd}\, t, \sigma_0)\right] \qquad (iii)$$

The relation $\cong$ is defined such that, for all strings σ and σ' over V, $\sigma \cong \sigma' \equiv \mathcal{L}(\sigma) = \mathcal{L}(\sigma')$.

Using the easy cases for d' given above, we can now refine (iii) as follows:

$(iii) \ \sqsubseteq$ (*Introduce IF, Introduce Assignment* (twice))
if $\sigma \neq \phi \wedge \mathsf{hd}\, t = \mathsf{hd}\,\sigma \rightarrow \sigma := \mathsf{tl}\,\sigma$
$[\!]$ $\sigma = \phi \vee \mathsf{hd}\, t \neq \mathsf{hd}\,\sigma \rightarrow \sigma := \phi$
fi

We now consider how to transform σ so that $\mathsf{hd}\,\sigma$ is not a nonterminal. The easiest way to change σ without changing $\mathcal{L}(d'(\mathsf{hd}\, t, \sigma))$ is to replace $\mathsf{hd}\,\sigma$ by the right-hand side of some

rule defining $\mathsf{hd}\,\sigma$. Thus, we want to find some string, say α, such that $\mathsf{hd}\,\sigma ::= \alpha \in P$ and $d'(\mathsf{hd}\,t, \sigma) \cong d'(\mathsf{hd}\,t, \alpha \frown \mathsf{tl}\,\sigma)$; we can then replace σ by $\alpha \frown \mathsf{tl}\,\sigma$.

Since the resulting value of σ may still start with a nonterminal, we might need to repeat the process. Thus, we want a loop, with $\mathsf{hd}\,\sigma \in V_N$ as the guard, and a loop invariant which ensures that $d'(\mathsf{hd}\,t, \sigma)$ remains unchanged, i.e.

$$Inv \;\widehat{=}\; d'(\mathsf{hd}\,t, \sigma) \cong d'(\mathsf{hd}\,t, \sigma_I)$$

where σ_I is the value of σ at the beginning of the loop, and will be declared as a logical constant surrounding the loop. We will also need a variant function, which we will call f, providing a well-founded ordering on σ; this will be discussed later.

$(ii) \;\sqsubseteq$ *(Introduce CON, Introduce DO)*

$$\begin{array}{l}\textbf{con } \sigma_I \bullet \\ \textbf{do } \mathsf{hd}\,\sigma \in V_N \rightarrow \\ \quad \sigma\colon \left[Inv \wedge \mathsf{hd}\,\sigma \in V_N \;/\; Inv \wedge 0 \leq f(\sigma) < f(\sigma_0) \right] \\ \textbf{od}\end{array} \qquad (iv)$$

The loop body needs to find a suitable α, and update σ as indicated above. Thus, we introduce a local variable, α, and split the specification into two parts. Since we know how to update σ, we use *Following Assignment* to find the condition the must be satisfied by the chosen value of α, as follows:

$$\begin{array}{l} wp(\sigma := \alpha \frown \mathsf{tl}\,\sigma,\; d'(\mathsf{hd}\,t, \sigma) \cong d'(\mathsf{hd}\,t, \sigma_I) \wedge 0 \leq f(\sigma) < f(\sigma_0)) \\ \quad = d'(\mathsf{hd}\,t, \alpha \frown \mathsf{tl}\,\sigma) \cong d'(\mathsf{hd}\,t, \sigma_I) \wedge 0 \leq f(\alpha \frown \mathsf{tl}\,\sigma) < f(\sigma)) \end{array}$$

Thus, we refine (iv) as follows:

$(iv) \;\sqsubseteq$ *(Introduce Local Variable, Following Assignment, Contract Frame)*

$$\begin{array}{l}\textbf{var } \alpha \bullet \\ \alpha\colon \left[\begin{array}{c} d'(\mathsf{hd}\,t, \sigma) \cong d'(\mathsf{hd}\,t, \sigma_I) \wedge \\ \mathsf{hd}\,\sigma \in V_N \end{array} \;/\; \begin{array}{c} d'(\mathsf{hd}\,t, \alpha \frown \mathsf{tl}\,\sigma) \cong d'(\mathsf{hd}\,t, \sigma_I) \wedge \\ 0 \leq f(\alpha \frown \mathsf{tl}\,\sigma) < f(\sigma) \end{array} \right]; \\ \sigma := \alpha \frown \mathsf{tl}\,\sigma \end{array} \qquad (v)$$

Since (v) cannot change σ and $\cong$ is an equivalence relation, we can simplify (v):

$(v) \;\sqsubseteq$ *(Weaken Precondition, Strengthen Postcondition)*

$$\alpha\colon \left[\mathsf{hd}\,\sigma \in V_N \;/\; \begin{array}{c} d'(\mathsf{hd}\,t, \alpha \frown \mathsf{tl}\,\sigma) \cong d'(\mathsf{hd}\,t, \sigma) \wedge \\ 0 \leq f(\alpha \frown \mathsf{tl}\,\sigma) < f(\sigma) \end{array} \right] \qquad (vi)$$

4.4 Selecting rules

We now consider how to refine (vi) so as to find a suitable value for α, and how to show that the loop terminates. We begin by investigating the first condition in the postcondition (replacing $\mathsf{hd}\,t$ by x):

$$
\begin{aligned}
& d'(x, \alpha \frown \mathsf{tl}\,\sigma) \cong d'(x, \sigma) \\
& \quad \Longleftrightarrow \quad \mathcal{L}(d'(x, \alpha \frown \mathsf{tl}\,\sigma)) = \mathcal{L}(d'(x, \sigma)) \\
& \quad \Longleftrightarrow \quad d(x, \mathcal{L}(\alpha \frown \mathsf{tl}\,\sigma)) = d(x, \mathcal{L}(\sigma)) \\
& \quad \Longleftrightarrow \quad (\forall \beta \bullet \beta \in d(x, \mathcal{L}(\alpha \frown \mathsf{tl}\,\sigma)) \equiv \beta \in d(x, \mathcal{L}(\sigma))) \\
& \quad \Longleftrightarrow \quad (\forall \beta \bullet x \frown \beta \in \mathcal{L}(\alpha \frown \mathsf{tl}\,\sigma) \equiv x \frown \beta \in \mathcal{L}(\sigma)) \\
& \quad \Longleftrightarrow \quad (\forall \beta \bullet \alpha \frown \mathsf{tl}\,\sigma \Rightarrow_G^* x \frown \beta \equiv \sigma \Rightarrow_G^* x \frown \beta)
\end{aligned}
$$

Thus, in order to satisfy $d'(\mathsf{hd}\,t, \alpha \frown \mathsf{tl}\,\sigma) \cong d'(\mathsf{hd}\,t, \sigma)$, we require that any string in $\mathcal{L}(\sigma)$ starting with $\mathsf{hd}\,t$ is also in $\mathcal{L}(\alpha \frown \mathsf{tl}\,\sigma)$, i.e. any string starting with $\mathsf{hd}\,t$ that can be produced from σ can also be produced from $\alpha \frown \mathsf{tl}\,\sigma$. This can be formalised by introducing a function *first* such that, for any string $\gamma \in V^*$,

$$
\mathit{first}(\gamma) \mathrel{\widehat{=}} \{\, x \in V_T \mid \exists \gamma' \in V_T^* \bullet \gamma \Rightarrow_G^* x \frown \gamma' \,\}
$$

Thus, we require $\mathsf{hd}\,t ::= \alpha \in P \wedge \mathsf{hd}\,t \in \mathit{first}(\alpha)$, if such an α exists. If $d(x, \mathcal{L}(\sigma)) = \varnothing$, we want $\alpha = \phi$.

The above condition also requires that this choice of α be unique, since $\sigma \Rightarrow_G^* x \frown \beta \Rightarrow \alpha \frown \mathsf{tl}\,\sigma \Rightarrow_G^* x \frown \beta$ says that *any* string beginning with x that can be produced from σ can be produced from $\alpha \frown \mathsf{tl}\,\sigma$. If there is more than one such α, the postcondition will be *false*, so (vi) is miraculous!

We will therefore add a further assumption, that for any nonterminal N and terminal x, there is at most one rule $N ::= \alpha$ in P, such that $x \in \mathit{first}(\alpha)$. This is equivalent to the assumption that G is an LL(1) grammar[4].

We thus refine (vi) as follows:

$$
\begin{aligned}
(v) \quad \sqsubseteq \quad & (\textit{Strengthen Postcondition}) \\
& \alpha{:}\left[\mathsf{hd}\,\sigma \in V_N \;\middle/\; \begin{array}{c} (\ \mathsf{hd}\,t ::= \alpha \in P \wedge \mathsf{hd}\,t \in \mathit{first}(\alpha)\ ) \ \vee \\ (\ \alpha = \phi \wedge \neg(\exists \beta \bullet N ::= \beta \in P \wedge \mathsf{hd}\,t \in \mathit{first}(\alpha))\ ) \end{array}\right] \qquad (vi)
\end{aligned}
$$

4.5 Termination

As the variant function, f, we take the length of the longest sequence $\sigma_1, \cdots, \sigma_n$, such that:

1. $\sigma_1 = \sigma$,
2. for $i = 1, \cdots, n-1$, $\mathsf{hd}\,\sigma_i \in V_N$ and for some α, $\mathsf{hd}\,\sigma_i ::= \alpha \in P$ and $\sigma_{i+1} = \alpha \frown \mathsf{tl}\,\sigma_i$, and
3. $\mathsf{hd}\,\sigma_n = \mathsf{hd}\,t$.

This function will be bounded below, provided that the grammar has no left-recursive rules (e.g. rules of the form $N ::= N\beta$), which follows from the above assumption that G is an LL(1) grammar; this also prohibits circular rules.

Alternatively, we can define an ordering, $>$, on symbols in $V_N \cup V_T$, such that $x > y$ iff $x ::= y\beta \in P$, for some string β. This is a well-founded ordering provided G has no left-recursive rules.

[4]The other condition usually applied to LL(1) grammars isn't needed, since we have assumed there are no λ-rules in G.

4.6 Representing rules

In order to implement (vi), we need to be able to compute *first* for any given nonterminal ($\mathsf{hd}\,\sigma$) and input symbol ($\mathsf{hd}\,t$). We may, however, require the value of *first* many times for the same pair of symbols. This suggests that we should compute all possible values for *first* and store them in a table.

We can introduce such a table, which we will call RR, as a further data refinement. The coupling invariant in this case is:

$$\begin{aligned} CI \;\widehat{=}\; (\forall N \in V_N, x \in V_T \;\bullet\; & \\ & (T(N,x) = \phi \equiv \neg(\exists \alpha \in V^* \;\bullet\; N ::= \alpha \in P \wedge x \in \mathit{first}(\alpha))\;) \wedge \\ & (\forall \alpha \in V^* \;\bullet\; T(N,x) = \alpha \equiv N ::= \alpha \in P \wedge x \in \mathit{first}(\alpha))) \end{aligned}$$

A table satisfying this relationship is guaranteed to exist when G is an LL(1) grammar; in particular, the LL(1) property ensures that $RR(N,x)$ is uniquely defined for all N and x.

We will assume that the grammar is provided in this form, since we may wish to construct the table for a given grammar once and then use it many times for recognising different strings. In this case, we can take the above invariant as a specification for a program to construct RR.

Thus, we have:

$$(vi) \quad \preceq \quad \alpha := RR(\mathsf{hd}\,t, \mathsf{hd}\,\sigma)$$

4.7 The final program

The resulting algorithm after performing this data refinement is:

$$\begin{array}{l} [\![\ \textbf{var}\ t, \sigma \bullet \\ \quad t, \sigma := s, S; \\ \quad \textbf{do}\ t \neq \lambda \wedge \sigma \neq \phi \rightarrow \\ \qquad \textbf{do}\ \sigma \neq \phi \wedge \mathsf{hd}\,\sigma \in V_N \rightarrow \\ \qquad\quad \alpha := RR(\mathsf{hd}\,t, \mathsf{hd}\,\sigma); \\ \qquad\quad \sigma := \alpha \frown \mathsf{tl}\,\sigma \\ \qquad \textbf{od}; \\ \qquad \textbf{if}\ \ \sigma \neq \phi \wedge \mathsf{hd}\,t = \mathsf{hd}\,\sigma \rightarrow \sigma := \mathsf{tl}\,\sigma \\ \qquad [\!]\ \ \sigma = \phi \vee \mathsf{hd}\,t \neq \mathsf{hd}\,\sigma \rightarrow \sigma := \phi \\ \qquad \textbf{fi}; \\ \qquad t := \mathsf{hd}\,t \\ \quad \textbf{od}; \\ \quad r := (t = \lambda \wedge \sigma = \lambda) \\ ]\!] \end{array}$$

The algorithm we have arrived at is a table-driven LL(1) parser or "predictive" parser, in which σ acts as a stack. The operation $\sigma := \alpha \frown \mathsf{tl}\,\sigma$ pushes the symbols in α onto the top

of the stack, after popping it; while $\sigma := \mathsf{tl}\,\sigma$ pops the stack. It remains to choose suitable implementations for *First* and RR, most likely as tables, and for the stack; but these details need not concern us here.

It is interesting to compare this algorithm with the predictive parsing algorithm usually given in texts such as [Aho, Sethi and Ullman, 1986]. That algorithm uses a single loop, with cases inside the loop according to whether the top of stack is a terminal or nonterminal. The version derived here makes the nature of the parsing process clearer: for each input symbol, we have to reduce nonterminals at the top of the stack until the top of stack is the same as the input symbol. The termination proof is also easier with this version, since the outer loop advances once for each input symbol (which is a consequence of the design of our abstract algorithm) and the inner loop reduces nonterminals — it is more difficult to find a suitable variant function (or ordering) when these both happen in a single loop. It should, however, be straightforward to transform the algorithm into a single loop version if required.

The use of ϕ to denote the empty language simplifies the handling of errors. The fact that we have ended up with tests for $\sigma = \phi$ in both the inner loop and the following conditional, suggests that we might have been better to check for this earlier. Indeed, we could have separated this case when we refined (i'), which would have also simplified the postcondition of (vi) — allowing the second conjunct to be dropped. This did not seem natural at the time, since we would have needed the *first* function, which we had not yet introduced.

5 Conclusions

We have illustrated the use of data refinement and program specialisation in deriving a reasonably complex algorithm. We believe that the derivation is more intelligible that would have otherwise been obtained using program refinement. In particular, we arrived at a parser for LL(1) grammars by adding restrictions to the initial problem in order to simplify implementation problems, rather than starting with that class of grammars in the initial specification. We believe that this is typical of the way in which new algorithms are discovered.

The places where we introduced these assumptions indicate the places where we would have to modify the derivation to handle a larger class of languages. To handle grammars with λ-rules, we would need a more elaborate way of determining whether $\lambda \in \mathcal{L}(\sigma)$, and the code to select a rule would need to be able to determine when to choose a λ-rule. If we wish to handle all context free grammars, we would need to either introduce backtracking (which is one way of avoiding the need for miracles), or represent C by a set of strings over V, or equivalently, using strings of the form $\sigma_1 \mid \sigma_2$, with the interpretation that $\mathcal{L}(\sigma_1 \mid \sigma_2) = \mathcal{L}(\sigma_1) \cup \mathcal{L}(\sigma_2)$. The resulting representations can, however, become very large.

Our decision to represent $d(s - t, L)$ by a string of terminal and/or nonterminal symbols lead us directly to a top-down parsing algorithm; at every step, $(s - t) \frown \sigma$ is a left-sentential form. If, instead, we represented $s - t$ by a string σ of terminal and/or nonterminal symbols, in such a way σ is a prefix of a right-sentential form, we will end up with a bottom-up parsing algorithm (cf [Hesselink, 1992]).

Acknowledgments

Thanks to Ray Nickson for his helpful comments on earlier versions of the paper.

References

[Aho, Sethi and Ullman, 1986] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Brzozowski, 1964] J. A. Brzozowski. "Derivatives of Regular Expressions". *J.A.C.M.* **11** (1964), pp481–494.

[Gravell, 1991] A. Gravell. "Specialising Abstract Programs". *Proc. 4th Refinement Workshop*, J. M. Morris & R. C. Shaw (Eds), Springer-Verlag, 1991, pp34–50.

[Groves, 1994a] Lindsay Groves. "Deriving language recognition algorithms: A case study in combining program specialisation and data refinement". *Proc. 6th Refinement Workshop*, British Computer Society, London, January 1994.

[Groves, 1994b] Lindsay Groves. "Program Derivation in the Refinement Calculus: An Introduction" *New Zealand Formal Program Development Colloquium*, Hamilton, 30 November–1 December, 1994.

[Hesselink, 1992] Wim H. Hesselink. "LR-parsing derived". Science of Computer Programming **19** (1992), pp171–196.

[Hopcroft and Ullman, 1979] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[Morgan, 1990] Carroll Morgan. *Programming from Specifications.* Prentice-Hall, 1990.

[Morgan and Gardiner, 1988] Carroll Morgan and Paul Gardiner. "Data refinement by calculation". *Acta Informatica* **27** (1990), pp481–503.

[Morris, 1989] J. M. Morris. "Laws of data refinement". *Acta Informatica* **26** (1989), pp287–308.

How to derive tidy drawings of trees

JEREMY GIBBONS
Department of Computer Science
University of Auckland
Private Bag 92019, Auckland, New Zealand.
Email: jeremy@cs.auckland.ac.nz

ABSTRACT. The *tree-drawing problem* is to produce a 'tidy' mapping of elements of a tree to points in the plane. In this paper, we derive an efficient algorithm for producing tidy drawings of trees. The specification, the starting point for the derivations, consists of a collection of intuitively appealing *criteria* satisfied by tidy drawings. The derivation shows constructively that these criteria completely determine the drawing. Indeed, there is essentially only one reasonable drawing algorithm satisfying the criteria: its development is almost mechanical.

The algorithm consists of an *upwards accumulation* followed by a *downwards accumulation* on the tree, and is further evidence of the utility of these two higher-order tree operations.

KEYWORDS. Derivation, program transformation, trees, upwards and downwards accumulations, drawing, layout.

1 Introduction

The *tree drawing problem* is to produce a mapping from elements of a tree to points in the plane. This mapping should correspond to a drawing that is in some sense 'tidy'. Our definition of tidiness consists of a collection of intuitively appealing criteria 'obviously' satisfied by tidy drawings.

We derive from these criteria an efficient algorithm for producing tidy drawings of binary trees. The derivation process is a constructive proof that the tidiness criteria completely determine the drawing. In other words, there is only one tidy drawing of any given tree. In fact, the derivation of the algorithm is a completely reasonable and almost routine calculation from the criteria: the algorithm itself, like the drawing, is essentially unique.

The algorithm that we derive (which is due originally to Reingold and Tilford (1981)) consists of an *upwards accumulation* followed by a *downwards accu-*

 An earlier version appears in *Proceedings of Salodays in Auckland*, C. Calude, M. J. J. Lennon and H. Maurer, eds., Auckland, 1994. Full paper submitted for publication and available as Computer Science Report No. 82 from the above address. This work has been partially supported by University of Auckland Research Committee grant number A18/XXXXX/62090/3414013.

mulation (Gibbons, 1991, 1993b) on the tree. Basically, an upwards accumulation on a tree replaces every element of that tree with some function of that element's descendents, while a downwards accumulation replaces every element with some function of that element's ancestors. These two higher-order operations on trees are fundamental components of many tree algorithms, such as tree traversals, the parallel prefix algorithm (Ladner and Fischer, 1980), evaluation of attributes in an attribute grammar (Deransart et al., 1988), evaluation of structured queries on text (Skillicorn, 1993), and so on. Their isolation is an important step in understanding and modularizing a tree algorithm. Moreover, work is progressing (Gibbons, 1993a; Gibbons et al., 1993) on the development of efficient *parallel* algorithms for evaluating upwards and downwards accumulations on a variety of parallel architectures. Identifying the accumulations as components of a known algorithm shows how to implement that algorithm efficiently in parallel.

For the purposes of exposition, we make the simplifying assumption that tree elements are unlabelled or, equivalently, that all labels are the same size. It is easy to generalize the algorithm to cover trees in which the labels may have greatly differing widths. A more interesting generalization covers the case in which tree labels may also have different *heights*. Bloesch (1993) gives two algorithms for this case. It is slightly more difficult to adapt the algorithm to cope with *general* trees, in which parents may have arbitrarily but finitely many children. Radack (1988) and Walker (1990) present two different approaches. Radack's algorithm is derived in (Gibbons, 1991). We do not discuss it here, because to do so would entail a significant increase in the number of definitions required.

The rest of this paper is organized as follows. In Section 2, we briefly describe our notation. In Section 3, we summarize the ideas behind upwards and downwards accumulations on trees; more of the motivation for these definitions is given in the full paper. In Section 4, we present the tidiness criteria, and outline a simple but inefficient tree-drawing algorithm. The derivation of an efficient algorithm, the main part of the paper, is sketched in Section 5; the details can be found in the full paper.

2 Notation

We will use the *Bird-Meertens Formalism* or 'BMF' (Meertens, 1986; Bird, 1987, 1988; Backhouse, 1989), a calculus for the construction of programs from their specifications by a process of equational reasoning. This calculus places great emphasis on notions and properties of *data*, as opposed to *program*, structure. The BMF is known colloquially as 'Squiggol', because its protagonists make heavy use of unusual symbols and syntax. This approach is helpful to the cognoscenti, but tends to make their work appear unnecessarily obscure to the uninitiated. For this reason, we will use a more traditional notation here. We will use mostly words rather than symbols, and mostly prefix functions rather than infix operators, simply to make

expressions easier to parse for those unfamiliar with the calculus. We hasten to add two points. First, this translation leaves the BMF 'philosophy' intact. Second, the presentation here, although more accessible, will be marginally less elegant than it might otherwise have been.

2.1 *Basic combinators*

Sectioning a binary operator involves providing it with one of its arguments, and results in a function of the other argument. For example, $(2+)$ and $(+2)$ are two ways of writing the function that adds two to its argument. The *constant function* $\mathsf{always}(a)$ returns a for every argument; for example, $\mathsf{always}(1)(2) = 1$. (Function application is left-associative, so that this parses as '$(\mathsf{always}(1))(2)$'.) Function composition is written '$\circ$'; for example, $\mathsf{always}(1) \circ \mathsf{always}(2) = \mathsf{always}(1)$. The *identity function* is written 'id'. The *converse* $\mathsf{conv}(\oplus)$ of a binary operator $\oplus$ is obtained by swapping its arguments; for example, $\mathsf{conv}(-)(x, y) = y - x$.

The *product type* $A \times B$ consists of pairs (a, b) of values, with $a \in A$ and $b \in B$. The *projection functions* fst and snd return the first and second elements of a pair. The *fork* $\mathsf{fork}(f, g)$ of two functions f and g takes a single value and returns a pair; thus, $\mathsf{fork}(f, g)(a) = (f(a), g(a))$.

2.2 *Promotion*

The notion of *promotion* comes up repeatedly in the BMF. We say that function f is '$\oplus$ to $\otimes$ promotable' if, for all a and b,

$$f(a \oplus b) \;=\; f(a) \otimes f(b)$$

Promotion is a generalization of distributivity:, f distributes through $\oplus$ iff f is $\oplus$ to $\oplus$ promotable. We say that f 'promotes through $\oplus$' if there is a $\otimes$ such that f is $\oplus$ to $\otimes$ promotable.

2.3 *Lists*

The type $\mathsf{list}(A)$ consists of lists of elements of type A. A list is either a singleton $[a]$ for some a, or the (associative) concatenation $x +\!\!+ y$ of two lists x and y. In this paper, all lists are non-empty. We write '$[\cdot]$' for the function taking a to $[a]$, and write longer lists in square brackets too—for example, '$[a, b, c]$' is an abbreviation for $[a] +\!\!+ [b] +\!\!+ [c]$. For every initial datatype such as lists, there is a higher-order function map, which applies a function to every element of a member of that datatype; for example, $\mathsf{map}(+1)([1, 2, 3]) = [2, 3, 4]$. We will use map for other datatypes such as trees later, and will trust to context to reveal which particular map is meant.

2.4 *Homomorphisms*

An important class of functions on lists are those called *homomorphisms*. These are the functions that promote through list concatenation. That is, h is a list homomorphism iff there is an associative operator $\otimes$ such that, for all x and y,

$$\mathsf{h}(\mathsf{x} +\!\!+ \mathsf{y}) \;=\; \mathsf{h}(\mathsf{x}) \otimes \mathsf{h}(\mathsf{y})$$

The condition of associativity on $\otimes$ is no great restriction. If h is $+\!\!+$ to $\otimes$ promotable then $\otimes$ is necessarily associative, at least on the range of h. In fact, if h is $+\!\!+$ to $\otimes$ promotable, then it is completely determined by its action on singleton lists; for example,

$$\mathsf{h}([\mathsf{a},\mathsf{b},\mathsf{c}]) \;=\; \mathsf{h}([\mathsf{a}] +\!\!+ [\mathsf{b}] +\!\!+ [\mathsf{c}]) \;=\; \mathsf{h}([\mathsf{a}]) \otimes \mathsf{h}([\mathsf{b}]) \otimes \mathsf{h}([\mathsf{c}])$$

If h is $+\!\!+$ to $\otimes$ promotable and $\mathsf{h} \circ [\cdot] = \mathsf{f}$, then we write h as $\mathsf{lh}(\mathsf{f}, \otimes)$ ('lh' stands for 'list homomorphism').

Stated another way, we have the *Promotion Theorem on Lists*, a special case of the *Promotion Theorem* (Malcolm, 1990):

THEOREM (1) If h is $\oplus$ to $\otimes$ promotable, then

$$\mathsf{h} \circ \mathsf{lh}(\mathsf{f}, \oplus) \;=\; \mathsf{lh}(\mathsf{h} \circ \mathsf{f}, \otimes)$$

$\diamondsuit$

Since $\mathsf{lh}([\cdot], +\!\!+) = \mathsf{id}$, this gives us a vehicle for proving the equality of a function h and a homomorphism $\mathsf{lh}(\mathsf{f}, \otimes)$, in that we need only show that h is $+\!\!+$ to $\otimes$ promotable, and that $\mathsf{h} \circ [\cdot] = \mathsf{f}$.

For each f, $\mathsf{map}(\mathsf{f})$ is a homomorphism, for

$$\mathsf{map}(\mathsf{f})(\mathsf{x} +\!\!+ \mathsf{y}) \;=\; \mathsf{map}(\mathsf{f})(\mathsf{x}) +\!\!+ \mathsf{map}(\mathsf{f})(\mathsf{y})$$

Indeed, $\mathsf{map}(\mathsf{f}) = \mathsf{lh}([\cdot] \circ \mathsf{f}, +\!\!+)$, because $\mathsf{map}(\mathsf{f})([\mathsf{a}]) = [\mathsf{f}(\mathsf{a})] = ([\cdot] \circ \mathsf{f})(\mathsf{a})$. Another example of a homomorphism is the function len, which returns the length of a list:

$$\mathsf{len} \;=\; \mathsf{lh}(\mathsf{always}(1), +)$$

The functions head and last, returning the first and last elements of a list, are also homomorphisms. For example,

$$\mathsf{head}(\mathsf{x} +\!\!+ \mathsf{y}) \;=\; \mathsf{head}(\mathsf{x}) \;=\; \mathsf{fst}(\mathsf{head}(\mathsf{x}), \mathsf{head}(\mathsf{y}))$$

and so $\mathsf{head} = \mathsf{lh}(\mathsf{id}, \mathsf{fst})$. Similarly, $\mathsf{last} = \mathsf{lh}(\mathsf{id}, \mathsf{snd})$. Other examples that we will encounter are the functions $\mathsf{smallest}$ and $\mathsf{largest}$, which return the smallest and largest elements of a list, respectively:

$$\begin{aligned} \mathsf{smallest} &= \mathsf{lh}(\mathsf{id}, \mathsf{min}) \\ \mathsf{largest} &= \mathsf{lh}(\mathsf{id}, \mathsf{max}) \end{aligned}$$

and the function sum, which returns the sum of the elements of a list:

$$\mathsf{sum} \;=\; \mathsf{lh}(\mathsf{id}, +)$$

2.5 Binary trees

Finally, we come to binary trees. The type $\mathsf{btree}(\mathsf{A})$ consists of binary trees labelled with elements of type A. A binary tree is either a leaf $\mathsf{lf}(\mathsf{a})$ labelled with a single

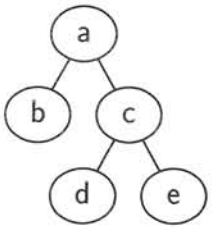

Figure 1: The tree five

element a, or a branch br(t, a, u) consisting of two children t and u and a label a. For example, the expression

$$\mathsf{br}(\mathsf{lf}(\mathsf{b}), \mathsf{a}, \mathsf{br}(\mathsf{lf}(\mathsf{d}), \mathsf{c}, \mathsf{lf}(\mathsf{e})))$$

corresponds to the tree in Figure 1, which we will call five and use as an example later.

Homomorphisms on binary trees bh(f, ⊕) ('binary tree homomorphism') promote through br. That is, they satisfy the equations:

$$\begin{aligned} \mathsf{bh}(\mathsf{f}, \oplus)(\mathsf{lf}(\mathsf{a})) &= \mathsf{f}(\mathsf{a}) \\ \mathsf{bh}(\mathsf{f}, \oplus)(\mathsf{br}(\mathsf{t}, \mathsf{a}, \mathsf{u})) &= \mathsf{bh}(\mathsf{f}, \oplus)(\mathsf{t}) \oplus_{\mathsf{a}} \mathsf{bh}(\mathsf{f}, \oplus)(\mathsf{u}) \end{aligned}$$

Note that for binary trees, the second component of a homomorphism is a *ternary* function. We write its middle argument as a subscript, for lack of anywhere better to put it. When instantiated to trees, Malcolm's Promotion Theorem states:

THEOREM (2) If h satisfies

$$\mathsf{h}(\mathsf{br}(\mathsf{t}, \mathsf{a}, \mathsf{u})) = \mathsf{h}(\mathsf{t}) \oplus_{\mathsf{a}} \mathsf{h}(\mathsf{u})$$

then $\mathsf{h} = \mathsf{bh}(\mathsf{h} \circ \mathsf{lf}, \oplus)$. ♢

The function map on binary trees satisfies

$$\begin{aligned} \mathsf{map}(\mathsf{f})(\mathsf{lf}(\mathsf{a})) &= \mathsf{lf}(\mathsf{f}(\mathsf{a})) \\ \mathsf{map}(\mathsf{f})(\mathsf{br}(\mathsf{t}, \mathsf{a}, \mathsf{u})) &= \mathsf{br}(\mathsf{map}(\mathsf{f})(\mathsf{t}), \mathsf{f}(\mathsf{a}), \mathsf{map}(\mathsf{f})(\mathsf{u})) \end{aligned}$$

and so

$$\mathsf{map}(\mathsf{f}) = \mathsf{bh}(\mathsf{lf} \circ \mathsf{f}, \oplus) \quad \text{where} \quad \mathsf{v} \oplus_{\mathsf{a}} \mathsf{w} = \mathsf{br}(\mathsf{v}, \mathsf{f}(\mathsf{a}), \mathsf{w})$$

The function root is a binary tree homomorphism:

$$\begin{aligned} \mathsf{root}(\mathsf{lf}(\mathsf{a})) &= \mathsf{a} \\ \mathsf{root}(\mathsf{br}(\mathsf{t}, \mathsf{a}, \mathsf{u})) &= \mathsf{a} \end{aligned}$$

and so

$$\mathsf{root} = \mathsf{bh}(\mathsf{id}, \oplus) \quad \text{where} \quad \mathsf{v} \oplus_{\mathsf{a}} \mathsf{w} = \mathsf{a}$$

So are the functions size and depth:

$$\begin{aligned} \mathsf{size} &= \mathsf{bh}(\mathsf{always}(1), \oplus) \quad \text{where} \quad \mathsf{v} \oplus_\mathsf{a} \mathsf{w} = \mathsf{v} + 1 + \mathsf{w} \\ \mathsf{depth} &= \mathsf{bh}(\mathsf{always}(1), \oplus) \quad \text{where} \quad \mathsf{v} \oplus_\mathsf{a} \mathsf{w} = 1 + \mathsf{max}(\mathsf{v}, \mathsf{w}) \end{aligned}$$

and the function brev, which reverses a binary tree:

$$\mathsf{brev} = \mathsf{bh}(\mathsf{lf}, \oplus) \quad \text{where} \quad \mathsf{v} \oplus_\mathsf{a} \mathsf{w} = \mathsf{br}(\mathsf{w}, \mathsf{a}, \mathsf{v})$$

2.6 Variable-naming conventions

To help the reader, we make a few conventions about the choice of names. For alphabetic names, single-letter identifiers are typically 'local', their definitions persisting only for a few lines, whereas multi-letter identifiers are 'global', having the same definitions throughout the paper. Elements of lists and trees are denoted $\mathsf{a}, \mathsf{b}, \mathsf{c}, \ldots$. Unary functions are denoted $\mathsf{f}, \mathsf{g}, \mathsf{h}$. Lists and paths (introduced in Section 3.2) are denoted $\mathsf{w}, \mathsf{x}, \mathsf{y}, \mathsf{z}$. Trees are denoted t, u. The letters v and w are used as the 'results' of functions, for example, in the definitions of homomorphisms such as brev above.

We define a few infix binary operators such as $\oplus$ and $\boxtimes$, just as we might use alphabetic names for variables and unary functions. Round binary operators such as $\oplus$ and $\otimes$ are 'local', and square binary operators such as $\boxplus$ and $\boxtimes$ are 'global'.

3 Upwards and downwards accumulations on trees

The material in this section is presented only in summary; a more complete description, including motivation, is given in the full paper and in (Gibbons, 1991).

3.1 Upwards accumulations

Upwards and downwards accumulations arise from considering the list function inits, which takes a list x and returns the list of lists consisting of the non-empty initial segments of x in order of increasing length. On trees, the obvious analogue of inits is the function subtrees, which takes a tree and returns a tree of trees. The result is the same shape as the original tree, but each element is replaced by its *descendents*, that is, by the subtree of the original tree rooted at that element. For example:

$$\begin{aligned} \mathsf{subtrees}(\mathsf{five}) = \; & \mathsf{br}(\mathsf{lf}(\mathsf{lf}(\mathsf{b})), \\ & \quad \mathsf{br}(\mathsf{lf}(\mathsf{b}), \mathsf{a}, \mathsf{br}(\mathsf{lf}(\mathsf{d}), \mathsf{c}, \mathsf{lf}(\mathsf{e}))), \\ & \quad \mathsf{br}(\mathsf{lf}(\mathsf{lf}(\mathsf{d})), \\ & \qquad \mathsf{br}(\mathsf{lf}(\mathsf{d}), \mathsf{c}, \mathsf{lf}(\mathsf{e})), \\ & \qquad \mathsf{lf}(\mathsf{lf}(\mathsf{e})))) \end{aligned}$$

which corresponds to the tree of trees in Figure 2. The function subtrees satisfies

$$\begin{aligned} \mathsf{subtrees}(\mathsf{lf}(\mathsf{a})) &= \mathsf{lf}(\mathsf{lf}(\mathsf{a})) \\ \mathsf{subtrees}(\mathsf{br}(\mathsf{t}, \mathsf{a}, \mathsf{u})) &= \mathsf{br}(\mathsf{subtrees}(\mathsf{t}), \mathsf{br}(\mathsf{t}, \mathsf{a}, \mathsf{u}), \mathsf{subtrees}(\mathsf{u})) \end{aligned}$$

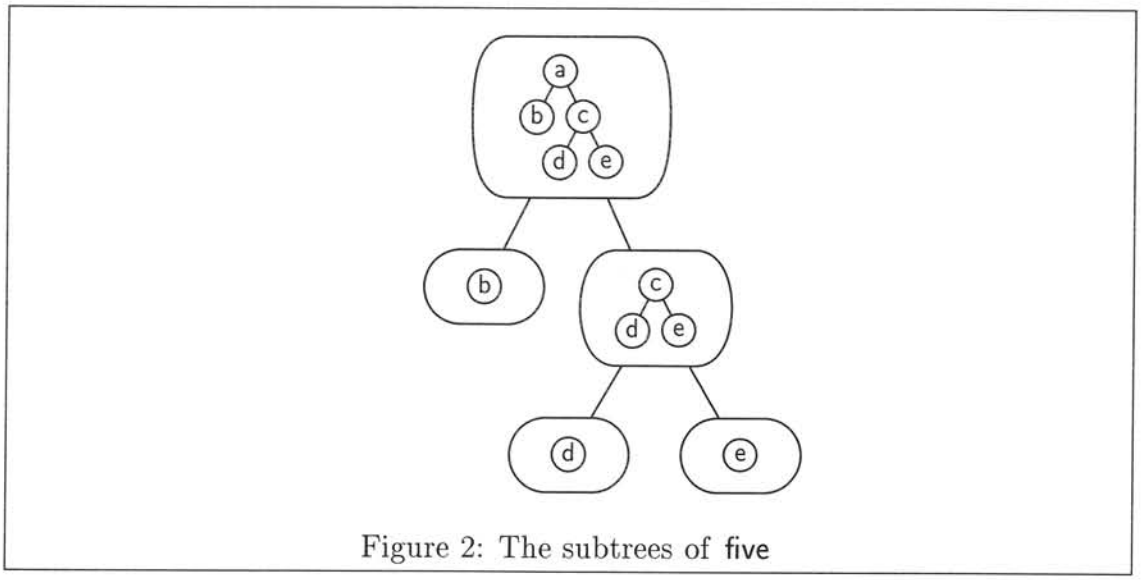

Figure 2: The subtrees of five

The *upwards accumulation* $\mathsf{up}(\mathsf{f}, \oplus)$ is obtained by mapping the tree homomorphism $\mathsf{bh}(\mathsf{f}, \oplus)$ over the subtrees of a tree:

$$\mathsf{up}(\mathsf{f}, \oplus) \;=\; \mathsf{map}(\mathsf{bh}(\mathsf{f}, \oplus)) \circ \mathsf{subtrees}$$

It can be computed in linear time (assuming that the f and $\oplus$ take constant time).

One example of an upwards accumulation is the function ndescs, which replaces every element with the number of descendents it has. Letting $\oplus$ satisfy $\mathsf{v} \oplus_{\mathsf{a}} \mathsf{w} = \mathsf{v} + 1 + \mathsf{w}$, so that $\mathsf{size} = \mathsf{bh}(\mathsf{always}(1), \oplus)$, we have

$$\begin{aligned} \mathsf{ndescs} &= \mathsf{map}(\mathsf{bh}(\mathsf{always}(1), \oplus)) \circ \mathsf{subtrees} \\ &= \mathsf{up}(\mathsf{always}(1), \oplus) \end{aligned}$$

Note that the expression involving the map takes quadratic time to compute, whereas the accumulation takes linear time.

3.2 Downwards accumulations

Upwards accumulations replace every element of a tree with some function of that element's descendents. For downwards accumulations, on the other hand, we consider an element's *ancestors*. The ancestors of an element form a *path*. For example, the ancestors of the element labelled d in five form the path in Figure 3, which could be thought of as a list with two different kinds of concatenation, 'left' and 'right', or as a tree in which each parent has exactly one child. We choose the former option. The type $\mathsf{path}(\mathsf{A})$ consists of paths of elements of type A. A path is either a single element $\langle \mathsf{a} \rangle$ or two paths x and y joined with a 'left turn', $\mathsf{x} \mathbin{+\!\!+\!\!\!\leftarrow} \mathsf{y}$, or a 'right turn', $\mathsf{x} \mathbin{+\!\!+\!\!\!\rightarrow} \mathsf{y}$. The function taking a to $\langle \mathsf{a} \rangle$ is written '$\langle \cdot \rangle$'. Just as $+\!\!+$ is associative, the operations $+\!\!+\!\!\!\leftarrow$ and $+\!\!+\!\!\!\rightarrow$ satisfy the four laws

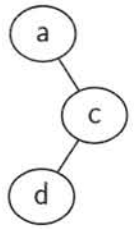

Figure 3: The path in five to the element labelled d

$$\begin{aligned} x \mathbin{+\!\!+} (y \mathbin{+\!\!+} z) &= (x \mathbin{+\!\!+} y) \mathbin{+\!\!+} z \\ x \mathbin{+\!\!+} (y \mathbin{+\!\!\Rightarrow} z) &= (x \mathbin{+\!\!+} y) \mathbin{+\!\!\Rightarrow} z \\ x \mathbin{+\!\!\Rightarrow} (y \mathbin{+\!\!+} z) &= (x \mathbin{+\!\!\Rightarrow} y) \mathbin{+\!\!+} z \\ x \mathbin{+\!\!\Rightarrow} (y \mathbin{+\!\!\Rightarrow} z) &= (x \mathbin{+\!\!\Rightarrow} y) \mathbin{+\!\!\Rightarrow} z \end{aligned}$$

We say that '$+\!\!+$ associates with $+\!\!\Rightarrow$', or '$+\!\!+$ and $+\!\!\Rightarrow$ associate with each other'. Thus, the path shown above to the element labelled d is represented by $\langle a \rangle +\!\!\Rightarrow \langle c \rangle +\!\!+ \langle d \rangle$. Because of the associativity property, brackets are unnecessary.

Path homomorphisms promote through both $+\!\!+$ and $+\!\!\Rightarrow$; if, for all a, x and y, the function h satisfies

$$\begin{aligned} h(\langle a \rangle) &= f(a) \\ h(x \mathbin{+\!\!+} y) &= h(x) \oplus h(y) \\ h(x \mathbin{+\!\!\Rightarrow} y) &= h(x) \otimes h(y) \end{aligned}$$

and $\oplus$ associates with $\otimes$, then we write $ph(f, \oplus, \otimes)$ for h.

We generalize path homomorphisms to *upwards* and *downwards* functions on paths. If, for all a, x and y, the function h satisfies

$$\begin{aligned} h(\langle a \rangle) &= f(a) \\ h(\langle a \rangle \mathbin{+\!\!+} y) &= a \oplus h(y) \\ h(\langle a \rangle \mathbin{+\!\!\Rightarrow} y) &= a \otimes h(y) \end{aligned}$$

then we say that h is *upwards*, and write it $uw(f, \oplus, \otimes)$. The operators $\oplus$ and $\otimes$ need not enjoy any associativity properties. Similarly, if, for all a, x and y,

$$\begin{aligned} h(\langle a \rangle) &= f(a) \\ h(x \mathbin{+\!\!+} \langle a \rangle) &= h(x) \oplus a \\ h(x \mathbin{+\!\!\Rightarrow} \langle a \rangle) &= h(x) \otimes a \end{aligned}$$

then we say that h is *downwards*, and write it $dw(f, \oplus, \otimes)$. Path homomorphisms are clearly both upwards and downwards; a generalization of Bird's Third Homomorphism Theorem (Gibbons, 1994) states the converse.

THEOREM (3) (Third Homomorphism Theorem for Paths (Gibbons, 1993a)) A path function that is both upwards and downwards is necessarily a path homomor-

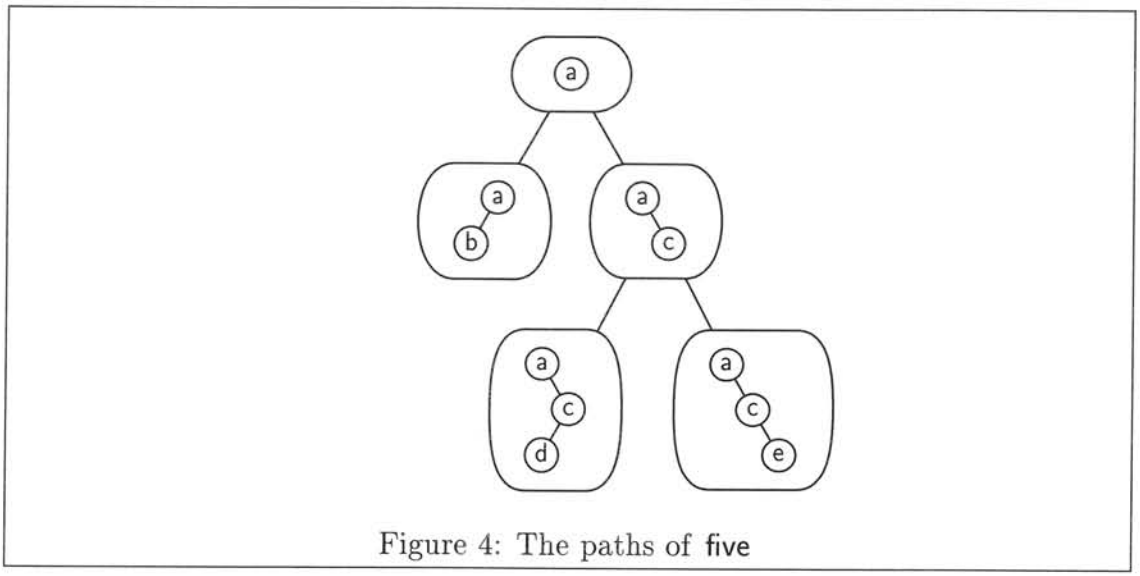

Figure 4: The paths of five

phism. ◊

The dual for downwards accumulations of the function subtrees is the function paths, which replaces each element of a tree with that element's ancestors. For example:

$$\begin{aligned}\mathsf{paths}(\mathsf{five}) \;=\; & \mathsf{br}(\mathsf{lf}(\langle \mathsf{a}\rangle \mathbin{+\!\!\!\leftarrow} \langle \mathsf{b}\rangle),\\ & \quad \langle \mathsf{a}\rangle,\\ & \quad \mathsf{br}(\mathsf{lf}(\langle \mathsf{a}\rangle \mathbin{+\!\!\!\rightarrow} \langle \mathsf{c}\rangle \mathbin{+\!\!\!\leftarrow} \langle \mathsf{d}\rangle),\\ & \qquad \langle \mathsf{a}\rangle \mathbin{+\!\!\!\rightarrow} \langle \mathsf{c}\rangle,\\ & \qquad \mathsf{lf}(\langle \mathsf{a}\rangle \mathbin{+\!\!\!\rightarrow} \langle \mathsf{c}\rangle \mathbin{+\!\!\!\rightarrow} \langle \mathsf{e}\rangle))))\end{aligned}$$

which corresponds to the tree of paths in Figure 4. The function paths is a tree homomorphism; it satisfies

$$\begin{aligned}\mathsf{paths}(\mathsf{br}(\mathsf{t},\mathsf{a},\mathsf{u})) \;=\; & \mathsf{br}(\mathsf{map}(\langle \mathsf{a}\rangle \mathbin{+\!\!\!\leftarrow})(\mathsf{paths}(\mathsf{t})),\\ & \quad \langle \mathsf{a}\rangle,\\ & \quad \mathsf{map}(\langle \mathsf{a}\rangle \mathbin{+\!\!\!\rightarrow})(\mathsf{paths}(\mathsf{u})))\end{aligned}$$

The *downwards accumulation* $\mathsf{down}(\mathsf{f},\oplus,\otimes)$ is obtained by mapping the path homomorphism $\mathsf{ph}(\mathsf{f},\oplus,\otimes)$ over the paths of a tree:

$$\mathsf{down}(\mathsf{f},\oplus,\otimes) \;=\; \mathsf{map}(\mathsf{ph}(\mathsf{f},\oplus,\otimes)) \circ \mathsf{paths}$$

Note that $\oplus$ and $\otimes$ must associate with each other for the path homomorphism to be valid.

For example, consider the function plen, which returns the length of a path. The function depths replaces every element of a tree with that element's depth in the tree, that is, with the length of its path of ancestors:

$$\mathsf{depths} \;=\; \mathsf{map}(\mathsf{plen}) \circ \mathsf{paths}$$

As it stands, it is not obvious whether depths is a homomorphism, nor whether it can be computed efficiently. However, plen is upwards,

$$\mathsf{plen} \;=\; \mathsf{uw}(\mathsf{always}(1), \oplus, \oplus) \qquad \text{where} \quad \mathsf{a} \oplus \mathsf{v} = 1 + \mathsf{v}$$

and so depths is a tree homomorphism. Moreover, plen is downwards,

$$\mathsf{plen} \;=\; \mathsf{dw}(\mathsf{always}(1), \oplus, \oplus) \qquad \text{where} \quad \mathsf{v} \oplus \mathsf{a} = \mathsf{v} + 1$$

and so depths can also be computed in linear time. Writing

$$\mathsf{depths} \;=\; \mathsf{down}(\mathsf{always}(1), +, +)$$

(since $+$ is associative, it associates with itself) shows that depths is both homomorphic and efficiently computable.

4 Drawing binary trees tidily

In this section, we define 'tidiness' and specify the function bdraw, which draws a binary tree. We make the simplifying assumption that all tree labels are the same size, because, for the purposes of positioning the elements of a tree, we can then ignore the labels altogether.

The first property that we observe of tidy drawings is that all of the elements at a given depth in a tree have the same y-coordinate in the drawing. That is, the y-coordinate is determined completely by the depth of an element, and the problem reduces to that of finding the x-coordinates. This gives us the type of bdraw, the function which draws a binary tree—its argument is of type $\mathsf{btree}(\mathsf{A})$ for some A, and its result is a binary tree labelled with x-coordinates:

$$\mathsf{bdraw} \;\in\; \mathsf{btree}(\mathsf{A}) \to \mathsf{btree}(\mathbb{D})$$

where coordinates range over $\mathbb{D}$, the type of distances. We require that $\mathbb{D}$ include the number 1, and be closed under subtraction (and hence also under addition) and halving. Sets satisfying these conditions include the reals, the rationals, and the rationals with finite binary expansions, the last being the smallest such set. We exclude discrete sets such as the integers, as Supowit and Reingold (1983) have shown that the problem is NP-hard with such coordinates.

Tidy drawings are also regular, in the sense that the drawing of a subtree is independent of the context in which it appears. Informally, this means that the drawings of children can be committed to (separate pieces of) paper before considering their parent. The drawing of the parent is then constructed by translating the drawings of the children. In symbols:

$$\mathsf{bdraw}(\mathsf{br}(\mathsf{t}, \mathsf{a}, \mathsf{u})) \;=\; \mathsf{br}(\mathsf{map}(+\mathsf{r})(\mathsf{bdraw}(\mathsf{t})), \mathsf{b}, \mathsf{map}(+\mathsf{s})(\mathsf{bdraw}(\mathsf{u})))$$

for some b, r and s.

Tidy drawings also exhibit no left-to-right bias. In particular, a parent should

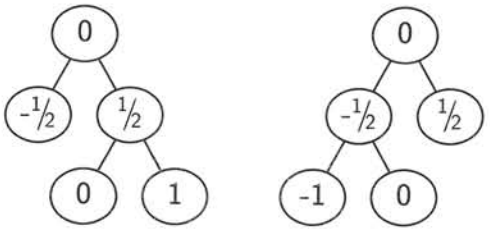

Figure 5: Drawings pic_1 and pic_2, for which $\mathsf{pic}_1 \boxplus \mathsf{pic}_2 = \text{-}2$

be centred over its children. We also specify that the root of a tree should be given x-coordinate 0. Hence, $r + s$ and b in the above equation should both be 0, as should the position given to the only element of a singleton tree:

$$\begin{aligned} \mathsf{bdraw}(\mathsf{lf}(a)) &= \mathsf{lf}(0) \\ \mathsf{bdraw}(\mathsf{br}(t, a, u)) &= \mathsf{br}(\mathsf{map}(-s)(\mathsf{bdraw}(t)), 0, \mathsf{map}(+s)(\mathsf{bdraw}(u))) \end{aligned}$$

for some s. Indeed, a tidy drawing will have the left child to the left of the right child, and so $s > 0$.

This lack-of-bias property implies that a tree and its mirror image produce drawings which are reflections of each other. That is, if we write '-' for unary negation[1], then we also require

$$\mathsf{bdraw} \circ \mathsf{brev} = \mathsf{map}(\text{-}) \circ \mathsf{brev} \circ \mathsf{bdraw}$$

The fourth criterion is that, in a tidy drawing, elements do not collide, or even get too close together. That is, pictures of children do not overlap, and no two elements on the same level are less than one unit apart.

Finally, a tidy drawing should be as narrow as possible, given the above constraints. Supowit and Reingold (1983) show that narrowness and regularity cannot be satisfied together—there are trees whose narrowest drawings can only be produced by drawing identical subtrees with different shapes—and so one of the two criteria must be made subordinate to the other. We choose to retain the regularity property, since it will lead us to a homomorphic solution.

These last two properties determine s, the distance through which children are translated. That distance should be the smallest distance that does not cause violation of the fourth criterion. Suppose the operator $\boxplus$, when given two drawings of trees, returns the width of the narrowest part of the gap between the trees. If the drawings overlap, this distance will be negative. For example, if pic_1 and pic_2 are as in Figure 5, then $\mathsf{pic}_1 \boxplus \mathsf{pic}_2 = \text{-}2$. The drawings should be moved apart or together to make this distance 1, that is,

[1]The presence of sectioning means that, strictly speaking, we should distinguish between the number 'minus one', written '-1', and the function 'minus one', written '(-1)'.

$$\mathsf{s} \;=\; (1 - (\mathsf{bdraw(t)} \boxplus \mathsf{bdraw(u)})) \div 2$$

(In the example above, s will be $1\frac{1}{2}$.)

All that remains to be done to complete the specification is to formalize this description of $\boxplus$.

4.1 Levelorder traversal

We define two different 'zip' operators, each of which takes a pair of lists and returns a single list by combining corresponding elements in some way. These two operators are 'short zip', which we write szip, and 'long zip', written lzip. These operators differ in that the length of the result of a short zip is the length of its shorter argument, whereas the length of the result of a long zip is the length of its longer argument. For example:

$$\begin{aligned} \mathsf{szip}(\oplus)([\mathsf{a},\mathsf{b}],[\mathsf{c},\mathsf{d},\mathsf{e}]) &= [\mathsf{a}\oplus\mathsf{c},\mathsf{b}\oplus\mathsf{d}] \\ \mathsf{lzip}(\oplus)([\mathsf{a},\mathsf{b}],[\mathsf{c},\mathsf{d},\mathsf{e}]) &= [\mathsf{a}\oplus\mathsf{c},\mathsf{b}\oplus\mathsf{d},\mathsf{e}] \end{aligned}$$

From the result of the long zip, we see that the $\oplus$ must have type $\mathsf{A}\times\mathsf{A}\to\mathsf{A}$. This is not necessary for short zip, but we do not use the general case.

The two zips are given formally by the equations

$$\begin{aligned} \mathsf{szip}(\oplus)([\mathsf{a}],[\mathsf{b}]) &= [\mathsf{a}\oplus\mathsf{b}] \\ \mathsf{szip}(\oplus)([\mathsf{a}],[\mathsf{b}]+\!\!+\mathsf{y}) &= [\mathsf{a}\oplus\mathsf{b}] \\ \mathsf{szip}(\oplus)([\mathsf{a}]+\!\!+\mathsf{x},[\mathsf{b}]) &= [\mathsf{a}\oplus\mathsf{b}] \\ \mathsf{szip}(\oplus)([\mathsf{a}]+\!\!+\mathsf{x},[\mathsf{b}]+\!\!+\mathsf{y}) &= [\mathsf{a}\oplus\mathsf{b}]+\!\!+\mathsf{szip}(\oplus)(\mathsf{x},\mathsf{y}) \end{aligned}$$

$$\begin{aligned} \mathsf{lzip}(\oplus)([\mathsf{a}],[\mathsf{b}]) &= [\mathsf{a}\oplus\mathsf{b}] \\ \mathsf{lzip}(\oplus)([\mathsf{a}],[\mathsf{b}]+\!\!+\mathsf{y}) &= [\mathsf{a}\oplus\mathsf{b}]+\!\!+\mathsf{y} \\ \mathsf{lzip}(\oplus)([\mathsf{a}]+\!\!+\mathsf{x},[\mathsf{b}]) &= [\mathsf{a}\oplus\mathsf{b}]+\!\!+\mathsf{x} \\ \mathsf{lzip}(\oplus)([\mathsf{a}]+\!\!+\mathsf{x},[\mathsf{b}]+\!\!+\mathsf{y}) &= [\mathsf{a}\oplus\mathsf{b}]+\!\!+\mathsf{lzip}(\oplus)(\mathsf{x},\mathsf{y}) \end{aligned}$$

Note that both $\mathsf{szip}(\oplus)(\mathsf{x},\mathsf{y})$ and $\mathsf{lzip}(\oplus)(\mathsf{x},\mathsf{y})$ can be evaluated with $\mathsf{min}(\mathsf{len}(\mathsf{x}),\mathsf{len}(\mathsf{y}))$ applications of $\oplus$.

We use long zip to define *levelorder traversal* of homogeneous binary trees. This is given by the function $\mathsf{levels}\in\mathsf{btree}(\mathsf{A})\to\mathsf{list}(\mathsf{list}(\mathsf{A}))$:

$$\mathsf{levels} \;=\; \mathsf{bh}([\cdot]\circ[\cdot],\oplus) \qquad \text{where} \quad \mathsf{x}\oplus_{\mathsf{a}}\mathsf{y} = [[\mathsf{a}]]+\!\!+\mathsf{lzip}(+\!\!+)(\mathsf{x},\mathsf{y})$$

For example, the levelorder traversals of $\mathsf{lf(b)}$ and $\mathsf{br(lf(d),c,lf(e))}$ are $[[\mathsf{b}]]$ and $[[\mathsf{c}],[\mathsf{d},\mathsf{e}]]$, respectively, and so

$$\begin{aligned} & \mathsf{levels(five)} \\ = \;& [[\mathsf{a}]]+\!\!+\mathsf{lzip}(+\!\!+)([[\mathsf{b}]],[[\mathsf{c}],[\mathsf{d},\mathsf{e}]]) \\ = \;& [[\mathsf{a}]]+\!\!+[[\mathsf{b}]+\!\!+[\mathsf{c}],[\mathsf{d},\mathsf{e}]] \\ = \;& [[\mathsf{a}],[\mathsf{b},\mathsf{c}],[\mathsf{d},\mathsf{e}]] \end{aligned}$$

We can at last define the operator $\boxplus$ on pictures, in terms of levelorder traversal. It is given by

$$\begin{aligned} \mathsf{p} \boxplus \mathsf{q} \;=\; & \mathsf{smallest}(\mathsf{szip}(\mathsf{conv}(-))(\mathsf{map}(\mathsf{largest})(\mathsf{levels}(\mathsf{p})), \\ & \qquad \mathsf{map}(\mathsf{smallest})(\mathsf{levels}(\mathsf{q})))) \end{aligned}$$

If v and w are levels at the same depth in p and q, then $\mathsf{largest}(\mathsf{v})$ and $\mathsf{smallest}(\mathsf{w})$ are the rightmost point of v and the leftmost point of w, respectively, and so $\mathsf{smallest}(\mathsf{w}) - \mathsf{largest}(\mathsf{v})$ is the width of the gap at this level. Clearly, $\mathsf{p} \boxplus \mathsf{q}$ is the minimum over all levels of these widths. For example, with pic_1 and pic_2 as in Figure 5, we have

$$\begin{aligned} \mathsf{map}(\mathsf{largest})(\mathsf{levels}(\mathsf{pic}_1)) \;&=\; [0, \tfrac{1}{2}, 1] \\ \mathsf{map}(\mathsf{smallest})(\mathsf{levels}(\mathsf{pic}_2)) \;&=\; [0, -\tfrac{1}{2}, -1] \end{aligned}$$

and so

$$\mathsf{pic}_1 \boxplus \mathsf{pic}_2 \;=\; \mathsf{smallest}([0-0, -\tfrac{1}{2}-\tfrac{1}{2}, -1-1]) \;=\; -2$$

This completes the specification of $\boxplus$, and hence of bdraw:

$$\mathsf{bdraw} \;=\; \mathsf{bh}(\mathsf{always}(\mathsf{lf}(0)), \oplus) \qquad \text{— (1)}$$

where

$$\begin{aligned} \mathsf{p} \oplus_{\mathsf{a}} \mathsf{q} \;&=\; \mathsf{br}(\mathsf{map}(-\mathsf{s})(\mathsf{p}), 0, \mathsf{map}(+\mathsf{s})(\mathsf{q})) \qquad \text{where} \quad \mathsf{s} = (1 - (\mathsf{p} \boxplus \mathsf{q})) \div 2 \\ \mathsf{p} \boxplus \mathsf{q} \;&=\; \mathsf{smallest}(\mathsf{szip}(\mathsf{conv}(-))(\mathsf{map}(\mathsf{largest})(\mathsf{levels}(\mathsf{p})), \\ & \qquad\qquad \mathsf{map}(\mathsf{smallest})(\mathsf{levels}(\mathsf{q})))) \end{aligned}$$

This specification is executable, but requires quadratic effort. We now sketch the derivation of a linear algorithm to satisfy it.

5 Drawing binary trees efficiently

A major source of inefficiency in the program we have just developed is the occurrence of the two maps in the definition of $\oplus$. Intuitively, we have to shift the drawings of two children when assembling the drawing of their parent, and then shift the whole lot once more when drawing the grandparent. This is because we are directly computing the absolute position of every element. If instead we were to compute the *relative* position of each parent with respect to its children, these repeated translations would not occur. A second pass—a downwards accumulation—can fix the absolute positions by accumulating relative positions.

Suppose the function $\mathsf{rootrel}$ on drawings of trees satisfies

$$\begin{aligned} \mathsf{rootrel}(\mathsf{lf}(\mathsf{a})) \;&=\; 0 \\ \mathsf{rootrel}(\mathsf{br}(\mathsf{t}, \mathsf{a}, \mathsf{u})) \;&=\; (\mathsf{a} - \mathsf{root}(\mathsf{t})) \odot (\mathsf{root}(\mathsf{u}) - \mathsf{a}) \end{aligned}$$

for some idempotent operator $\odot$. The idea here is that $\mathsf{rootrel}$ determines the position of a parent relative to its children, given the drawing of the parent. For example, with pic_1 as in Figure 5, we have

$$\mathsf{rootrel}(\mathsf{pic}_1) \;=\; (0 - \text{-}\tfrac{1}{2}) \odot (\tfrac{1}{2} - 0) \;=\; \tfrac{1}{2}$$

That is, if we define the function sep by

$$\mathsf{sep} \;=\; \mathsf{rootrel} \circ \mathsf{bdraw}$$

then

$$\begin{aligned} \mathsf{sep}(\mathsf{lf}(\mathsf{a})) &= 0 \\ \mathsf{sep}(\mathsf{br}(\mathsf{t},\mathsf{a},\mathsf{u})) &= (1 - (\mathsf{bdraw}(\mathsf{t}) \boxplus \mathsf{bdraw}(\mathsf{u}))) \div 2 \end{aligned}$$

For example:

$$\begin{aligned} \mathsf{sep}(\mathsf{five}) &= (1 - (\mathsf{bdraw}(\mathsf{lf}(\mathsf{b})) \boxplus \mathsf{bdraw}(\mathsf{br}(\mathsf{lf}(\mathsf{d}),\mathsf{c},\mathsf{lf}(\mathsf{e}))))) \div 2 \\ &= (1 - 0) \div 2 \\ &= \tfrac{1}{2} \end{aligned}$$

Then

$$\begin{aligned} \mathsf{bdraw}(\mathsf{br}(\mathsf{t},\mathsf{a},\mathsf{u})) \;=\; & \mathsf{br}(\mathsf{map}(-\mathsf{s})(\mathsf{bdraw}(\mathsf{t})), 0, \mathsf{map}(+\mathsf{s})(\mathsf{bdraw}(\mathsf{u}))) \\ & \text{where} \quad \mathsf{s} = \mathsf{sep}(\mathsf{br}(\mathsf{t},\mathsf{a},\mathsf{u})) \end{aligned}$$

Now, applying sep to each subtree gives the relative (to its children) position of every parent. Define the function rel by

$$\mathsf{rel} \;=\; \mathsf{map}(\mathsf{sep}) \circ \mathsf{subtrees}$$

From this, we can (and in the full paper, do) calculate that

$$\begin{aligned} \mathsf{rel}(\mathsf{lf}(\mathsf{a})) &= \mathsf{lf}(0) \\ \mathsf{rel}(\mathsf{br}(\mathsf{t},\mathsf{a},\mathsf{u})) &= \mathsf{br}(\mathsf{rel}(\mathsf{t}), \mathsf{sep}(\mathsf{br}(\mathsf{t},\mathsf{a},\mathsf{u})), \mathsf{rel}(\mathsf{u})) \end{aligned}$$

This gives us the first 'pass', computing the position of every parent relative to its children. How can we get from this to the absolute position of every element? We need a function abs satisfying the condition

$$\mathsf{abs} \circ \mathsf{rel} \;=\; \mathsf{bdraw}$$

We can (and again, in the full paper, do) calculate from this requirement a definition of abs:

$$\begin{aligned} \mathsf{abs}(\mathsf{lf}(\mathsf{a})) &= \mathsf{lf}(0) \\ \mathsf{abs}(\mathsf{br}(\mathsf{t},\mathsf{a},\mathsf{u})) &= \mathsf{br}(\mathsf{map}(-\mathsf{a})(\mathsf{abs}(\mathsf{t})), 0, \mathsf{map}(+\mathsf{a})(\mathsf{abs}(\mathsf{u}))) \end{aligned}$$

This is equivalent to

$$\mathsf{abs} \;=\; \mathsf{map}(\mathsf{uw}(\mathsf{always}(0), \mathsf{conv}(-), +)) \circ \mathsf{paths}$$

We give the upwards function $\mathsf{uw}(\mathsf{always}(0), \mathsf{conv}(-), +)$ a name, pabs ('the absolute position of the bottom of a path'), for brevity:

$$\mathsf{pabs} \;=\; \mathsf{uw}(\mathsf{always}(0), \mathsf{conv}(-), +)$$

so that

$$\mathsf{abs} \;=\; \mathsf{map}(\mathsf{pabs}) \circ \mathsf{paths}$$

Thus, we have

$$\mathsf{bdraw} \;=\; \mathsf{abs} \circ \mathsf{rel} \qquad \text{— (2)}$$

where

$$\begin{aligned} \mathsf{rel} &= \mathsf{map}(\mathsf{sep}) \circ \mathsf{subtrees} \\ \mathsf{abs} &= \mathsf{map}(\mathsf{pabs}) \circ \mathsf{paths} \end{aligned}$$

This is still inefficient, as computing rel takes quadratic time (because sep is not a tree homomorphism) and computing abs takes quadratic time (because pabs is not a downwards function on paths). We show next how to compute rel and abs quickly.

5.1 An upwards accumulation

We want to find an efficient way of computing the function rel satisfying

$$\mathsf{rel} \;=\; \mathsf{map}(\mathsf{sep}) \circ \mathsf{subtrees}$$

where

$$\begin{aligned} \mathsf{sep}(\mathsf{lf}(\mathsf{a})) &= 0 \\ \mathsf{sep}(\mathsf{br}(\mathsf{t},\mathsf{a},\mathsf{u})) &= (1 - (\mathsf{bdraw}(\mathsf{t}) \boxplus \mathsf{bdraw}(\mathsf{u}))) \div 2 \end{aligned}$$

We have already observed that rel is not an upwards accumulation, because sep is not a homomorphism—more information than the separations of the grandchildren is needed in order to compute the separation of the children. How much more information is needed? It is not hard to see that, in order to compute the separation of the children, we need to know the 'outlines' of their drawings. That is, define the function contours by

$$\begin{aligned} \mathsf{contours} \;=\; & \mathsf{fork}(\mathsf{left},\mathsf{right}) \circ \mathsf{bdraw} \\ & \quad \text{where} \quad \begin{aligned} \mathsf{left} &= \mathsf{map}(\mathsf{smallest}) \circ \mathsf{levels} \\ \mathsf{right} &= \mathsf{map}(\mathsf{largest}) \circ \mathsf{levels} \end{aligned} \end{aligned}$$

For example, bdraw(five) is pic_1 in Figure 5, and applying the function fork(left, right) to this tree produces the pair of lists $([0, -\frac{1}{2}, 0], [0, \frac{1}{2}, 1])$.

To show that these contours provide the extra information needed to make sep a homomorphism, we need to show that sep can be computed from the contours, and that computing the contours is a homomorphism.

For the first of these,

$$\mathsf{sep} \;=\; \mathsf{spread} \circ \mathsf{contours}$$

where, for some idempotent $\odot$,

$$\begin{aligned} \mathsf{spread}([0],[0]) &= 0 \\ \mathsf{spread}([0] +\!\!+ \mathsf{x}, [0] +\!\!+ \mathsf{y}) &= -\mathsf{head}(\mathsf{x}) \odot \mathsf{head}(\mathsf{y}) \end{aligned}$$

on pairs of lists, each with head 0.

Now we show that contours is a homomorphism. In the full paper, we calculate that

$$\begin{aligned} \mathsf{contours}(\mathsf{lf}(a)) &= ([0],[0]) \\ \mathsf{contours}(\mathsf{br}(t,a,u)) &= \mathsf{contours}(t) \circledast_a \mathsf{contours}(u) \end{aligned}$$

where

$$\begin{aligned} &(w,x) \circledast_a (y,z) \\ &\quad = ([0] +\!\!+ \mathsf{lzip}(\mathsf{fst})(\mathsf{map}(-s)(w), \mathsf{map}(+s)(y)), \\ &\qquad [0] +\!\!+ \mathsf{lzip}(\mathsf{snd})(\mathsf{map}(-s)(x), \mathsf{map}(+s)(z))) \\ &\qquad\quad \text{where} \quad s = (1 - (x \boxtimes y)) \div 2 \end{aligned} \qquad \text{— (3)}$$

Hence,

$$\mathsf{contours} = \mathsf{bh}(\mathsf{always}(([0],[0])), \circledast)$$

Thus,

$$\mathsf{rel} = \mathsf{map}(\mathsf{spread}) \circ \mathsf{up}(\mathsf{always}(([0],[0])), \circledast) \qquad \text{— (4)}$$

This is now an upwards accumulation, but it is still expensive to compute. The operation $\circledast$ takes at least linear effort, resulting in quadratic effort for the upwards accumulation. One further step is needed before we have an efficient algorithm for rel.

We have to find an efficient way of evaluating the operator $\circledast$ from (3):

$$\begin{aligned} (w,x) \circledast_a (y,z) &= ([0] +\!\!+ \mathsf{lzip}(\mathsf{fst})(\mathsf{map}(-s)(w), \mathsf{map}(+s)(y)), \\ &\qquad [0] +\!\!+ \mathsf{lzip}(\mathsf{snd})(\mathsf{map}(-s)(x), \mathsf{map}(+s)(z))) \\ &\qquad\quad \text{where} \quad s = (1 - (x \boxtimes y)) \div 2 \end{aligned}$$

One way of doing this is with a data refinement whereby, instead of maintaining a list of absolute distances, we maintain a list of relative distances. That is, we make a data refinement using the invertible abstraction function $\mathsf{msi} = \mathsf{map}(\mathsf{sum}) \circ \mathsf{inits}$, which computes absolute distances from relative ones. Under this refinement, the maps can be performed in constant time, since

$$\begin{aligned} \mathsf{map}(+s)(\mathsf{msi}(x)) &= \mathsf{msi}(\mathsf{mapplus}(s,x)) \\ &\qquad \text{where} \quad \mathsf{mapplus}(b,[a]) = [b+a] \\ &\qquad\qquad \mathsf{mapplus}(b,[a] +\!\!+ x) = [b+a] +\!\!+ x \end{aligned} \qquad \text{— (5)}$$

The refined $\circledast$ still takes linear effort because of the zips, but the important observation is that it now takes effort proportional to the length of its *shorter* argument (that is, to the lesser of the common lengths of w and x and the common lengths of y and z, when $\circledast$ is 'called' with arguments (w,x) and (y,z)). Reingold and Tilford (1981) show that, if evaluating $h(t) \oplus_a h(u)$ from a, $h(t)$ and $h(u)$ takes effort proportional to the lesser of the depths of the trees t and u, then the tree homomorphism $h = \mathsf{bh}(f, \oplus)$ can be evaluated with linear effort. Actually,

what they show is that if g satisfies

$$\begin{aligned} g(\mathsf{lf}(a)) &= 0 \\ g(\mathsf{br}(t,a,u)) &= g(t) + \min(\mathsf{depth}(t), \mathsf{depth}(u)) + g(u) \end{aligned}$$

then

$$g(x) = \mathsf{size}(x) - \mathsf{depth}(x)$$

which can easily be proved by induction. Intuitively, g counts the number of pairs of horizontally adjacent elements in a tree.

With this data refinement, rel can be computed in linear time.

5.2 A downwards accumulation

We now have an efficient algorithm for rel. All that remains to be done is to find an efficient algorithm for abs, where

$$\begin{aligned} \mathsf{abs} &= \mathsf{map}(\mathsf{pabs}) \circ \mathsf{paths} \\ \mathsf{pabs} &= \mathsf{uw}(\mathsf{always}(0), \mathsf{conv}(-), +) \end{aligned}$$

We note first that computing abs as it stands *is* inefficient. No operator $\oplus$ can satisfy $a + \mathsf{always}(0)(b) = \mathsf{always}(0)(a) \oplus b$ for all a and b, and so pabs can not be computed downwards, and abs is not a downwards accumulation. Intuitively, pabs starts at the bottom of a path and discards the bottom element, but we cannot do this when starting at the top of the path.

What extra information do we need in order to be able to compute pabs downwards? It turns out that the function pabsb, where

$$\mathsf{pabsb} = \mathsf{fork}(\mathsf{pabs}, \mathsf{bottom})$$

and where bottom returns the bottom element of a path

$$\mathsf{bottom} = \mathsf{uw}(\mathsf{id}, \mathsf{snd}, \mathsf{snd})$$

is a path homomorphism:

$$\begin{aligned} \mathsf{pabsb} &= \mathsf{ph}(f, \oplus, \otimes) \\ &\quad \text{where} \quad f(a) = (0, a) \\ &\qquad (v,w) \oplus (x,y) = (v - w + x, y) \\ &\qquad (v,w) \otimes (x,y) = (v + w + x, y) \end{aligned}$$

Now, $\mathsf{pabs} = \mathsf{fst} \circ \mathsf{pabsb}$, and so

$$\mathsf{abs} = \mathsf{map}(\mathsf{fst}) \circ \mathsf{down}(f, \oplus, \otimes) \qquad \text{— (6)}$$

which can be computed in linear time.

5.3 The program

To summarize, the program that we have derived is as in Figure 6.

$$
\begin{aligned}
\mathsf{bdraw} &= \mathsf{abs} \circ \mathsf{rel} \\
\\
\mathsf{rel} &= \mathsf{map}(\mathsf{spread}) \circ \mathsf{up}(\mathsf{always}(([0],[0])), \circledast) \\
(w,x) \circledast_a (y,z) &= ([0] +\!\!+ \mathsf{lzipfst}(\mathsf{mapplus}(\text{-}s,w), \mathsf{mapplus}(s,y)), \\
&\qquad [0] +\!\!+ \mathsf{lzipsnd}(\mathsf{mapplus}(\text{-}s,x), \mathsf{mapplus}(s,z))) \\
&\qquad\quad \text{where} \quad s = (1 - (x \boxtimes y)) \div 2 \\
\\
\mathsf{mapplus}(b,[a]) &= [a+b] \\
\mathsf{mapplus}(b,[a] +\!\!+ x) &= [a+b] +\!\!+ x \\
\\
\mathsf{lzipfst}(x,y) &= x, \qquad \text{if } \mathsf{nst}(x,y) \\
&= x +\!\!+ \mathsf{mapplus}(\mathsf{sum}(v) - \mathsf{sum}(x), w), \quad \text{otherwise} \\
&\qquad\quad \text{where} \quad (v,w) = \mathsf{split}(\mathsf{len}(x), y) \\
\mathsf{lzipsnd}(x,y) &= \mathsf{lzipfst}(y,x) \\
\\
\mathsf{nst}(x,[b]) &= \mathsf{true} \\
\mathsf{nst}([a],[b] +\!\!+ y) &= \mathsf{false} \\
\mathsf{nst}([a] +\!\!+ x,[b] +\!\!+ y) &= \mathsf{nst}(x,y) \\
\\
\mathsf{split}(1,[a] +\!\!+ x) &= ([a],x) \\
\mathsf{split}(n+1,[a] +\!\!+ x) &= ([a] +\!\!+ v, w) \qquad \text{where} \quad (v,w) = \mathsf{split}(n,x) \\
\\
\mathsf{spread}([0],[0]) &= 0 \\
\mathsf{spread}([0] +\!\!+ x,[0] +\!\!+ y) &= \text{-}\mathsf{head}(x) \odot \mathsf{head}(y) \qquad \text{where} \quad a \odot a = a \\
\\
v \boxtimes w &= \mathsf{lh}(\mathsf{id},\mathsf{min})(\mathsf{szip}(\mathsf{conv}(-))(v,w)) \\
\\
\mathsf{abs} &= \mathsf{map}(\mathsf{fst}) \circ \mathsf{down}(f,\oplus,\otimes) \\
&\qquad \text{where} \qquad f(a) = (0,a) \\
&\qquad\qquad (v,w) \oplus (x,y) = (v - w + x, y) \\
&\qquad\qquad (v,w) \otimes (x,y) = (v + w + x, y)
\end{aligned}
$$

Figure 6: The final program

6 Conclusion

6.1 Summary

We have presented a number of natural criteria satisfied by tidy drawings of unlabelled binary trees. From these criteria, we have sketched the derivation of an efficient algorithm for producing such drawings.

We started with an executable specification (1)—an 'obviously correct' but inefficient program. From this we needed only four inventions to yield a linear algorithm:

(i) we eliminated one source of inefficiency, by computing first the position of every parent relative to its children, and then fixing the absolute positions in a second pass (2);
(ii) we made a step towards making the first pass efficient, by turning the function computing relative positions into an upwards accumulation (4), computing not just relative positions but also the outlines of the drawings;
(iii) we made a data refinement on the outline of a drawing (5), allowing us to shift it in constant time; and
(iv) we made the second pass efficient by turning the function computing absolute positions into a downwards accumulation (6), computing not just the absolute positions but also the bottom element of every path.

The derivation showed several things:

(i) the criteria uniquely determine the drawing of a tree;
(ii) the criteria also determine the algorithm—at each stage in the derivation there was effectively only one thing to do (this claim is more defensible given the detailed derivation in the full paper);
(iii) the algorithm (due to Reingold and Tilford (1981)) is just an upwards accumulation followed by a downwards accumulation, and is further evidence of the utility of these higher-order operations;
(iv) identifying these accumulations as major components of the algorithm may lead, using known techniques for computing accumulations in parallel, to an optimal *parallel* algorithm for drawing unlabelled binary trees.

6.2 Related work

The problem of drawing trees has quite a long and interesting history. Knuth (1968, 1971) and Wirth (1976) both present simple algorithms in which the x-coordinate of an element is determined purely by its position in inorder traversal. Wetherell and Shannon (1979) first considered 'aesthetic criteria', but their algorithms all produce biased drawings. Independently of Wetherell and Shannon, Vaucher (1980) gives an algorithm which produces drawings that are simultaneously biased, irregular, and wider than necessary, despite his claims to have 'overcome the problems' of Wirth's simple algorithm. Reingold and Tilford (1981) tackle the problems in Wetherell and Shannon's and Vaucher's algorithms by proposing the criteria concerning bias and regularity. Their algorithm is the one derived for binary trees here. Supowit

and Reingold (1983) show that it is not possible to satisfy regularity and minimal width simultaneously, and that the problem is NP-hard when restricted to discrete (for example, integer) coordinates. Brüggemann-Klein and Wood (1990) implement Reingold and Tilford's algorithm as macros for the text formatting system TEX.

The problem of drawing general trees has had rather less coverage in the literature. General trees are harder to draw than binary trees, because it is not so clear what is meant by 'placing siblings as close as possible'. For example, consider a general tree with three children, `t`, `u` and `v`, in which `t` and `v` are large but `u` relatively small. It is not sufficient to consider just adjacent pairs of siblings when spacing the siblings out, because `t` may collide with `v`. Spacing the siblings out so that `t` and `v` do not collide allows some freedom in placing `u`, and care must be taken not to introduce any bias. Reingold and Tilford (1981) mention general trees in passing, but make no reference to the difficulty of producing unbiased drawings. Bloesch (1993) (who adapts Vaucher's and Reingold and Tilford's algorithms to cope with node labels of varying width and height) also appears not to attempt to produce unbiased drawings, despite his claims to the contrary. Radack (1988) effectively constructs two drawings, one packing siblings together from the left and the other from the right, and then averages the results. That algorithm is derived in (Gibbons, 1991). Walker (1990) uses a slightly different method. He positions children from left to right, but when a child touches against a left sibling other than the nearest one, the extra displacement is apportioned among the intervening siblings.

6.3 Further work

Gibbons (1991) extends this derivation to general trees. We have yet to apply the methods used here to Bloesch's algorithm (Bloesch, 1993) for drawing trees in which the labels may have different heights, but do not expect it to yield any surprises. It may also be possible to apply the techniques in (Gibbons et al., 1993) to yield an optimal *parallel* algorithm to draw a binary tree of `n` elements in $\log n$ time on $n/\log n$ processors, even when the tree is unbalanced—although this is complicated by having to pass non-constant-size contours around in computing $\circledast$.

6.4 Acknowledgements

Thanks are due to Sue Gibbons, for improving the presentation of this paper considerably.

References

Roland Backhouse (1989). *An exploration of the Bird-Meertens formalism.* In *International Summer School on Constructive Algorithmics, Hollum, Ameland.* STOP project. Also available as Technical Report CS 8810, Department of Computer Science, Groningen University, 1988.

Richard S. Bird (1987). *An introduction to the theory of lists.* In M. Broy, editor,

Logic of Programming and Calculi of Discrete Design, pages 3–42. Springer-Verlag. Also available as Technical Monograph PRG-56, from the Programming Research Group, Oxford University.

Richard S. Bird (1988). *Lectures on constructive functional programming.* In Manfred Broy, editor, *Constructive Methods in Computer Science.* Springer-Verlag. Also available as Technical Monograph PRG-69, from the Programming Research Group, Oxford University.

Anthony Bloesch (1993). *Aesthetic layout of generalized trees.* Software—Practice and Experience, 23(8):817–827.

Anne Brüggemann-Klein and Derick Wood (1990). *Drawing trees nicely with TEX.* In Malcolm Clark, editor, *TEX: Applications, Uses, Methods*, pages 185–206. Ellis Horwood.

Pierre Deransart, Martin Jourdan, and Bernard Lorho (1988). *LNCS 323: Attribute Grammars—Definitions, Systems and Bibliography.* Springer-Verlag.

Jeremy Gibbons, Wentong Cai, and David Skillicorn (1993). *Efficient parallel algorithms for tree accumulations.* Computer Science Report No. 70, Department of Computer Science, University of Auckland. Accepted for publication in Science of Computer Programming.

Jeremy Gibbons (1991). *Algebras for Tree Algorithms.* D. Phil. thesis, Programming Research Group, Oxford University. Available as Technical Monograph PRG-94.

Jeremy Gibbons (1993a). *Computing downwards accumulations on trees quickly.* In Gopal Gupta, George Mohay, and Rodney Topor, editors, *Proceedings of the 16th Australian Computer Science Conference*, pages 685–691. Available by anonymous `ftp` as `out/jeremy/papers/quickly.ps.Z` on `cs.auckland.ac.nz`.

Jeremy Gibbons (1993b). *Upwards and downwards accumulations on trees.* In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *LNCS 669: Mathematics of Program Construction*, pages 122–138. Springer-Verlag. A revised version appears in the Proceedings of the Massey Functional Programming Workshop, 1992.

Jeremy Gibbons (1994). *The Third Homomorphism Theorem.* Department of Computer Science, University of Auckland.

Donald E. Knuth (1968). *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* Addison-Wesley.

Donald E. Knuth (1971). *Optimum binary search trees.* Acta Informatica, 1:14–25.

Richard E. Ladner and Michael J. Fischer (1980). *Parallel prefix computation.* Journal of the ACM, 27(4):831–838.

Grant Malcolm (1990). *Algebraic Data Types and Program Transformation.* PhD thesis, Rijksuniversiteit Groningen.

Lambert Meertens (1986). *Algorithmics: Towards programming as a mathematical*

activity. In J. W. de Bakker, M. Hazewinkel, and J. K. Lenstra, editors, *Proc. CWI Symposium on Mathematics and Computer Science*, pages 289–334. North-Holland.

G. M. Radack (1988). *Tidy drawing of* M*-ary trees.* Technical Report CES-88-24, Department of Computer Engineering and Science, Case Western Reserve University, Cleveland, Ohio.

Edward M. Reingold and John S. Tilford (1981). *Tidier drawings of trees.* IEEE Transactions on Software Engineering, 7(2):223–228.

David B. Skillicorn (1993). *Parallel evaluation of structured queries in text.* Draft, Department of Computing and Information Sciences, Queen's University, Kingston, Ontario.

Kenneth J. Supowit and Edward M. Reingold (1983). *The complexity of drawing trees nicely.* Acta Informatica, 18(4):377–392.

Jean G. Vaucher (1980). *Pretty-printing of trees.* Software—Practice and Experience, 10:553–561.

John Q. Walker, II (1990). *A node-positioning algorithm for general trees.* Software—Practice and Experience, 20(7):685–705.

Charles Wetherell and Alfred Shannon (1979). *Tidy drawings of trees.* IEEE Transactions on Software Engineering, 5(5):514–520.

Niklaus Wirth (1976). *Algorithms + Data Structures = Programs.* Prentice Hall.