ELECTRONIC **WORKSHOPS IN COMPUTING**

Series edited by Professor C.J. van Rijsbergen

Gerard O'Regan and Sharon Flynn (Eds)

1st Irish Workshop on Formal Methods

Proceedings of the 1st Irish Workshop on Formal Methods, Dublin 3-4 July 1997

Paper:

Denotational Semantics and Refinement

S. Riddle and P. Wallis

Published in collaboration with the British Computer Society

BCS

Springer

Denotational Semantics and Refinement

Steve Riddle
Centre for Software Reliability, Department of Computing Science, University of Newcastle Upon Tyne
Newcastle Upon Tyne, UK

Peter J. L. Wallis
School of Mathematical Sciences, University of Bath
Bath, UK

Abstract

Denotational descriptions are frequently presented in terms of a sequence of descriptions of programming languages of increasing complexity. We discuss a formal approach to the study of such description sequences in terms of a refinement relation, showing how the informal use made of this approach in tutorial introductions to denotational semantics may be formally justified.

1 Introduction

Denotational semantics, originally developed by Christopher Strachey [8], has become an established branch of theoretical computer science with its own notations and conventions. It concerns the study of programming language semantics in the most abstract way, modelling programs and language constructs in terms of functions whose domain and range are related to the states of an abstract 'machine'.

Refinement is a partial ordering that holds between descriptions. It has a clear intuitive meaning: if a description B refines another description A then any customer for A would be satisfied if supplied with B. Refinement has principally been studied in connection with the formal development of programs from specification. In the present work we argue that it also applies to the way detailed descriptions can be developed stepwise from simpler ones, showing how this idea applies directly to the way in which denotational descriptions of programming languages are conventionally presented.

Our current interest in denotational semantics concerns its overall form and structure rather than its detailed technical content. In what follows, the meaning of the equations of denotational semantics is treated as peripheral to our main interest in the overall organisation of the semantic equations themselves as a vehicle for conveying detailed technical information lucidly. Semantic equations, like many of the formal descriptions of interest to computer scientists, tend to become extremely complicated when applied to examples of realistic complexity. This complexity is usually addressed by presenting semantic equations in a way intended to make them easier to understand. The purpose of the work described here is to cast some light on the methods conventionally used to do this.

Two groups of methods are of interest to us. First, the structure of each individual denotational description of a programming language follows a well-established convention according to which the Abstract Syntax, Semantic Algebras and Valuation Functions for a denotational semantics are separately described in sequence: these conventions are reviewed in Section 3.1. Secondly, the semantics of a programming language is typically presented stepwise, with the progressive addition of features to a sequence of specifications of languages of increasing complexity. This technique, originally due to Strachey, is now widely used in tutorial introductions to denotational semantics such as [7, 2, 5], and our main aim in this paper is to suggest a formal approach, using the classical notion of refinement, to justifying its use.

The notion of refinement, fundamental to this work, is explained in Section 2, which also explains how the idea of refinement relates to the idea of adding complexity to a formal description. The ideas explained in general terms in Section 2 are related specifically to denotational semantic descriptions of programming languages in Section 3. Section 4 then shows using a series of examples how this enables us to demonstrate the validity of various ways of adding complexity to a denotational description. Finally we summarise what we have achieved and relate it to other recent research.

2 Programming Languages, Description and Refinement

The denotational semantics of a programming language may be viewed simply as a description, written in a formal language, of the programming language concerned. Our present purpose is to explore the way in which denotational semantics is usually presented in tutorial introductions to the subject. The conventional way of arranging such a presentation is to give denotational semantic definitions of a sequence of model programming languages of increasing complexity, simple transformations relating successive members of this sequence.

We present our ideas in terms of refinement, which is a very general relationship holding between descriptions. In the simplest form of refinement, sufficient for present purposes, we say that B refines A (written $A \sqsubseteq B$) if a customer for A would be satisfied if supplied with B. The refinement relation $\sqsubseteq$ requires two formal properties in order to facilitate the handling of sequences of refinements: it must be reflexive

$A \sqsubseteq A$ for all descriptions A

to provide a base case for reasoning and to allow replacement of one description in a sequence by an equivalent one without invalidating the sequence, and it must also be transitive

$(A \sqsubseteq B) \wedge (B \sqsubseteq C) \Rightarrow A \sqsubseteq C.$

In Section 3 we develop a notion of refinement applicable to denotational descriptions that captures the idea that one denotational description refines another if all language constructs defined in the first language are also featured in the second with precisely the same meaning, in other words that the second language is a superset of the first. This refinement relation will then be shown to be reflexive and transitive, as required.

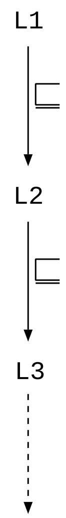

Figure 1: Refinement relations between successive languages of a sequence

Given a suitable notion of refinement that can be defined as holding between programming languages, we can then express the idea of a sequence of denotational semantic descriptions of languages of increasing complexity by saying that if we have a sequence of such languages L1, L2, L3 , . . . then we require

L1 $\sqsubseteq$ L2 $\sqsubseteq$ L3 . . .

as shown diagrammatically in Figure 1.

A more interesting structure of refinements might be obtained if it were possible to express as denotational descriptions the additional language features incorporated at each stage: a refinement structure like that of Figure 2 is then obtained. Here L1, L2, L3 ... are the languages of the sequence and E1, E2, ... the enhancements added between stages. Examples of language enhancements where this kind of treatment is possible are given in the first two of the examples considered in Section 4.

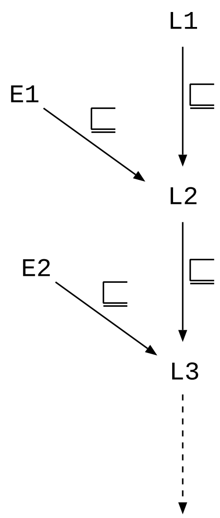

Figure 2: Refinement relations as in Figure 1 but with formalised language extensions.

Figure 2 exhibits a structure in which, for example, the description L2 is a refinement of both the descriptions L1 and E1. Such structures are of great interest in studying how complicated formal descriptions can be built from simpler parts, and have been studied in a number of contexts related to computer science. It is usual to describe such a relationship by saying that L1 and E1 are both *viewpoints* of L2; furthermore, if L2 can actually be constructed from L1 and E1, we describe this process as the *amalgamation* of the viewpoints L1 and E1 to produce the viewpoint L2.

A further feature of Figure 2 is that, as indicated by the $\sqsubseteq$ symbols, an amalgamated viewpoint is a *refinement* of each of its constituent ones. This is indeed the case for viewpoints relating to denotational descriptions, which are the only ones considered here. However, in general the relationship between viewpoints and corresponding amalgamations is a more general one called *co-refinement*[1]. We return to this point briefly in Section 5: for now note simply that for the denotational descriptions considered here refinement and co-refinement happen to be exactly equivalent.

3 Denotational Descriptions and Refinement

The preceding discussion describing incremental specification in terms of refinement is entirely general: it assumes nothing about the kind of language descriptions we are working with, and nothing about the kind of refinement relation that might hold between them. As a first step in applying this background theory to the way denotational semantics is organised, we need to relate this general discussion to the particular case of denotational semantic descriptions. We do this in two stages: first we summarise features of a denotational description, and then consider what notion of refinement might usefully be considered to apply to the relationship between denotational descriptions of different programming languages.

3.1 Form of a Denotational Description

In the rest of this paper we refer to a complete denotational semantic description of a programming language simply as a *denotational description*. Such a description provides a function corresponding to a program, to be applied to its

Abstract Syntax:

P $\in$ Program
S $\in$ Expr-Seq
E $\in$ Expr
N $\in$ Numeral

P ::= ON S
S ::= E TOTAL S | E TOTAL OFF
E ::= E_1 + E_2 | E_1 * E_2 | IF E_1,E_2,E_3 | LASTANS | (E) | N

Semantic Algebras:

Truth Values
Domain $t \in Tr = \mathbf{B}$
Operations
$true, false : Tr$
$not : Tr \rightarrow Tr$
$or : Tr \times Tr \rightarrow Tr$
$(\_ \rightarrow \_ \, [] \, \_) : Tr \times D \times D \rightarrow D$

Natural Numbers
Domain $n \in Nat = \mathbf{N}$
Operations
$zero, one, \ldots : Nat$
$plus, times : Nat \times Nat \rightarrow Nat$
$equals : Nat \times Nat \rightarrow Tr$

Valuation Functions:
$\mathbf{P}$:Program$\rightarrow Nat^*$
$\mathbf{P}[\![\texttt{ON S}]\!] = \mathbf{S}[\![\texttt{S}]\!](zero)$

$\mathbf{S}$:Expr-Seq$\rightarrow Nat \rightarrow Nat^*$
$\mathbf{S}[\![\texttt{E TOTAL S}]\!] = \lambda n.\ let\ n' = \mathbf{E}[\![\texttt{E}]\!](n) in\ n'\ \ cons\ \ \mathbf{S}[\![\texttt{S}]\!](n')$
$\mathbf{S}[\![\texttt{E TOTAL OFF}]\!] = \lambda n.\ \mathbf{E}[\![\texttt{E}]\!](n)\ \ cons\ \ nil$

$\mathbf{E}$:Expr$\rightarrow Nat \rightarrow Nat$
$\mathbf{E}[\![\texttt{E}_1\ \texttt{+}\ \texttt{E}_2]\!] = \lambda n.\ \mathbf{E}[\![\texttt{E}_1]\!](n)\ \ plus\ \ \mathbf{E}[\![\texttt{E}_2]\!](n)$
$\mathbf{E}[\![\texttt{E}_1\ \texttt{*}\ \texttt{E}_2]\!] = \lambda n.\ \mathbf{E}[\![\texttt{E}_1]\!](n)\ \ times\ \ \mathbf{E}[\![\texttt{E}_2]\!](n)$
$\mathbf{E}[\![\texttt{IF E}_1\texttt{,E}_2\texttt{,E}_3]\!] = \lambda n.\ \mathbf{E}[\![\texttt{E}_1]\!](n)\ \ equals\ \ zero \rightarrow \mathbf{E}[\![\texttt{E}_2]\!](n)\ []\ \mathbf{E}[\![\texttt{E}_3]\!](n)$
$\mathbf{E}[\![\texttt{LASTANS}]\!] = \lambda n.\ n$
$\mathbf{E}[\![\texttt{(E)}]\!] = \lambda n.\ \mathbf{E}[\![\texttt{E}]\!](n)$
$\mathbf{E}[\![\texttt{N}]\!] = \lambda n.\ \mathbf{N}[\![\texttt{N}]\!]$

$\mathbf{N}$: Numeral $\rightarrow Nat$ (definitions omitted)

Figure 3: Calculator semantics, based on Figure 4.3 of [5] with the author's permission.

initial state. This function is defined over the sub-expressions of the program. Figure 3 provides an example.

There are three parts to the description:-

1. Abstract Syntax — defined in terms of syntax domains and BNF rules. Rather than use terminal symbols as a *concrete* syntax BNF description would, each rule is expressed in terms of the tokens (or words) which are members of a syntax domain. For example in Figure 3, `P ∈ Program` indicates that `P` is an arbitrary non-terminal in the syntax domain `Program`.

2. Semantic Algebras — the domains and operations that are used to describe the semantics of the language. In Figure 3, domain *Nat* has nullary operations *zero*, *one*, . . ., and binary operations *plus*, *times*, *equals*. The domain of truth values *Tr* includes an infix *choice* operator which provides an if–then–else construct.

3. Valuation Functions — the main part of the denotational description, which defines a function for each legal

sub-expression of the program. In figure 3, the valuation function $\mathbf{P}[\![\texttt{ON S}]\!]$ has signature `Program`$\rightarrow$ *Nat* and denotes the meaning of a program as the meaning of a further valuation function S on the text of the program `S`. The brackets $[\![\ldots]\!]$ serve to separate syntax tokens from function definition.

Modelling a program's meaning as a function makes it possible to compare two statements in a language and show that they are equivalent by showing that they have the same denotation.

3.2 Refinement Relations for Denotational Descriptions

In order to understand the structure of sequences of denotational descriptions involving several programming languages, it is necessary to consider what meaning might be given to the notion of a "refinement relation" between denotational descriptions of different languages. We approach this in stages, first considering what the "correctness" of a denotational semantics might mean and then using a suitable definition of correctness to develop the applicable notion of refinement. We begin with a definition of *correctness* for denotational descriptions and programming languages.

Definition 3.1 (Correct implementation) *A programming language is a* correct *implementation of a semantic description iff all programs in that language satisfy the valuation function in the semantic description.*

In particular this definition means that a programming language which contains statements to which no valuation function applies will not be a correct implementation of the semantic description.

We can now say that for a programming language L, another language L' preserves the correctness of L if it also satisfies L's semantic description. However, this suggests that if L' has been modified to allow a wider range of program statements, correctness will not be preserved: L's semantic description will not include the extra program statements.

So our requirement is rather that, if a semantic description S is amalgamated with another to form a new description S', any programming language which implements S' should also be a possible implementation of S. The problem is that our present definition of correctness is too strong to allow any modifications to the semantics other than those which maintain equivalence. So to make further progress we weaken our definition of correctness.

Definition 3.2 (Consistent implementation) *A programming language L is a* consistent *implementation of a semantic description S iff, for all programs P in the language L ($P \in L$), either $S(P)$ is either undefined or P satisfies S.*

With this weakened definition of "correctness", we can require that, for a refinement relation to hold between a semantic description S and an extended description S', any implementation of S' must be a consistent implementation of S.

Recalling that a semantic description is an expression of a program's meaning as a function, we can finally define a refinement relation as follows:

Definition 3.3 (Refinement) *For semantic descriptions S and S', and respective implementations L and L', $S \sqsubseteq S'$ iff for all programs $P \in L$,*

$$\begin{array}{ll} & P \in L' \\ \textit{and} & S(P) \equiv S'(P) \end{array}$$

Thus refinement between semantic descriptions is defined as equivalence between meanings of a program in each of the semantic descriptions. The meaning of a program is a function too, from input to output or from initial state to final state, so $S(P) \equiv S'(P)$ exactly when their graphs are equal, that is $S(P)(x) = S'(P)(x)$ for all x.

This refinement relation is reflexive (if $S = S'$ and $L = L'$, refinement will hold) and transitive: for S, S', S'' and L, L', L'' such that $S \sqsubseteq S'$ and $S' \sqsubseteq S''$, we have for all programs P,

$(P \in L \Rightarrow P \in L') \wedge (P \in L' \Rightarrow P \in L'') \Leftrightarrow (P \in L \Rightarrow P \in L'')$

and

$(S(P) \equiv S'(P)) \wedge (S'(P) \equiv S''(P)) \Leftrightarrow S(P) \equiv S''(P)$

The relation preserves our weaker definition of correctness, *consistent implementation*: if L is a consistent implementation of S, and we derive (S', L') such that $S \sqsubseteq S'$ and L' is a consistent implementation of S', then L' is a consistent implementation of S.

The proof is as follows: if we take any program P in the language L', it may or may not be a valid program in the language L. If it is, then $P \in L$ and by the refinement relation $S'(P) \equiv S(P)$, so consistency is satisfied. Otherwise, $S(P)$ will not be defined so consistency is satisfied.

4 Examples

Having described the way adding features to a programming language may be presented in terms of refinement, it remains to show by example how this is done. We will add various features to the calculator semantics shown in Figure 3, in each case preceding the example with a discussion of the mathematical background necessary to establish that the corresponding transformation of the semantics is in fact a refinement in the sense that we have defined it.

Two examples are presented in their entirety; the third, dealing in a more general way with the conversion of a direct semantics to a continuation semantics, is given in outline form only. Our three examples together embrace typical ways in which language features are extended in tutorial presentations of denotational semantics such as [7, 2, 5]. The treatment closely follows Chapter 6 of [3].

4.1 Simple Extension of Abstract Syntax

To develop a denotational description of a programming language we need to define operations to build specifications. These can then be used in an amalgamation of two viewpoints representing denotational descriptions, and as long as we use operators that are monotonic with respect to the refinement ordering, it follows that the amalgamation refines its constituent viewpoints.

Using the notation $syn(S)$,$alg(S)$ and $val(S)$ to refer to the abstract syntax, semantic algebra and valuation function parts of semantic viewpoint S, we can identify operations to perform on a semantic viewpoint:

- Adding to the abstract syntax

  If we wish to extend the range of expressions in our programming language, we start by making an addition to the abstract syntax. If we simply do this, however, we have an inconsistent viewpoint as the valuation function is not defined over the new expression. Thus after adding to the syntax, if we wish to regain consistency (which we are only obliged to do if we wish to implement the viewpoint), the valuation function must also be extended.

- Extending the valuation function

  This can be done without also making the syntax change, but again an inconsistent viewpoint will result. We will only have a refinement if both operations are performed. For the moment we will insist that the new valuation function introduce no new semantic domain, i.e. operations used in the function are already defined in the semantic algebra. For new syntactic expression $token \texttt{::=} e$ and new valuation $f[\![e]\!]$

  $S \sqsubseteq S'$, where $S' = S[syn(S) \backslash syn(S) \cup \{token \texttt{::=} e\}, val(S) \backslash val(S) \cup \{f[\![e]\!]\}]$, provided $f[\![e]\!]$ and $token \texttt{::=} e$ are not already defined in S, and all semantic domains used in the definition of f are in S. If f is already defined in S, the signature of f must be identical.

  This is indeed a refinement, since any program for which $S(P)$ is defined will have the same meaning in S'. The operation of extending the syntax and valuation function is monotonic, since the new function will not interfere with any function already in the viewpoint.

With these basic operators we can go through a simple example, adding an elementary decision operator (`IF`) to the calculator whose semantics were given in Figure 3.

The valuation functions can be expanded on by the following commentary. A calculator program consists of a press of the ON button followed by a sequence of expressions. After the final expression has been terminated with a press of the TOTAL button, the OFF button terminates the program. The calculator also has a last-answer memory feature, hence the form of the signature for $\mathtt{S} : \mathtt{Expr\text{-}Seq} \to \mathtt{Nat} \to \mathtt{Nat}^*$. An expression sequence is evaluated as a function from the current "LASTANS" (initially `zero`) to a sequence of numbers.

A sample program would thus be
"`ON 3 + 4 TOTAL IF LASTANS, 5,6 TOTAL LASTANS + (2 * 7) OFF`".
This is evaluated as follows:

$$
\begin{aligned}
& \mathbf{P}[\![\mathtt{ON\ 3\ +\ 4\ TOTAL\ IF\ LASTANS,\ 5,6\ TOTAL\ LASTANS\ +\ (2\ *\ 7)\ OFF}]\!] \\
\widehat{=}\ & \mathbf{S}[\![\mathtt{3\ +\ 4\ TOTAL\ IF\ LASTANS,\ 5,6\ TOTAL\ LASTANS\ +\ (2\ *\ 7)\ OFF}]\!](zero) \\
\widehat{=}\ & let\ n' = \mathbf{E}[\![\mathtt{3\ +\ 4}]\!](zero)\ in \\
& \qquad n'\ cons\ \mathbf{S}[\![\mathtt{IF\ LASTANS,\ 5,6\ TOTAL\ LASTANS\ +\ (2\ *\ 7)\ OFF}]\!](n')
\end{aligned}
$$

The **E** expression evaluates to:

$$
\begin{aligned}
& \mathbf{E}[\![\mathtt{3\ +\ 4}]\!](zero) \\
\widehat{=}\ & \mathbf{E}[\![\mathtt{3}]\!](zero)\ plus\ \mathbf{E}[\![\mathtt{4}]\!](zero) \\
\widehat{=}\ & \mathbf{N}[\![\mathtt{3}]\!]\ plus\ \mathbf{N}[\![4]\!] \\
\widehat{=}\ & three\ plus\ four \\
\widehat{=}\ & seven
\end{aligned}
$$

so the **S** expression becomes:
$\mathtt{S}[\![\mathtt{IF\ LASTANS,\ 5,6\ TOTAL\ LASTANS\ +\ (2\ *\ 7)\ OFF}]\!](seven)$
and so on, leading to an output sequence of $\langle seven, six, twenty\rangle$.

4.1.1 Incremental Viewpoint

An exercise in [5] deals with adding a new button to the calculator which should test for equality, to be used in conjunction with the **IF** button; thus we would be able to say
`IF (5+4)=(3*3),2,3`
which would give 2. This suggests that the test for equality should return zero for true, non-zero for false, and the new valuation function would look like this:
$\mathbf{E}[\![\mathtt{E}_1 = \mathtt{E}_2]\!](n) = \mathbf{E}[\![\mathtt{E}_1]\!](n)\ equals\ \mathbf{E}[\![\mathtt{E}_2]\!](n) \to zero\ [\,]\ one$

A new part also needs to be added to the abstract syntax, for $\mathtt{E} ::= \mathtt{E}_1 = \mathtt{E}_2$.

We now come to the question of how a new version of the semantic description given in Figure 3 is formed with this new feature. We consider our initial viewpoint to be this semantic description.

We can express the additional information as an incremental viewpoint, as follows:

```
Abstract Syntax:

E ∈ Expr
E ::= E1 = E2
```

```
Semantic Algebras:

Truth Values
...
Natural Numbers
...
```

Valuation Functions:
$\mathbf{E} : \mathtt{Expr} \to Nat \to Nat$
$\quad \mathbf{E}[\![\mathtt{E}_1 = \mathtt{E}_2]\!](n) = \mathbf{E}[\![\mathtt{E}_1]\!](n)\ \ equals\ \ \mathbf{E}[\![\mathtt{E}_2]\!](n) \to zero\ [\,]\ one$

What we have is a valuation function only defined for expressions of the form $\mathtt{E}_1 \; \mathtt{=} \; \mathtt{E}_2$. This viewpoint is clearly quite useless on its own, being recursively defined with no base case. It is fact arguable whether we need to include the algebras for truth values and natural numbers, but we would argue that it is, since in amalgamating the viewpoint with the original we need to know whether any new semantic domains are to be added.

Having identified the new information we now need to amalgamate it with the initial viewpoint. First we identify what changes need to be made to achieve the amalgamation. To add the new piece of syntax to the viewpoint we need first check that it is unique, i.e. that $\mathtt{E} \; \mathtt{::=} \; \mathtt{E}_1 \; \mathtt{=} \; \mathtt{E}_2$ is not already in the viewpoint. We operate under the assumption that items with the same name refer to the same item, i.e. that `E` in the incremental viewpoint is the same as `E` in the original.

The result of this check is that the new syntax is not already defined, but that the syntax domain `E` is already defined. The resulting action to perform in amalgamating the viewpoints will therefore be to augment the syntax domain `E` with the new form. This check also shows that a valuation function needs to be defined for the new syntax.

The next stage is to check the semantic algebra. The algebraic domains in the incremental viewpoint are a subset of those in the original one, and have the same operations and signatures.

Finally the valuation functions. The new function **E** is already defined in the original viewpoint, but not on the expression $\mathtt{E}_1 \; \mathtt{=} \; \mathtt{E}_2$. So it can be added, after checking that **E**'s signature is the same as in the original viewpoint. This check also discharges the condition on the addition of abstract syntax, as the new valuation function defines the semantics of the new piece of syntax. Amalgamating these viewpoints then simply involves the following action:

Action: *Calc_New* is to be formed from

$$\begin{array}{l} Calc[syn(Calc)\backslash syn(Calc) \cup \mathtt{E} \; \mathtt{::=} \; \mathtt{E}_1 \; \mathtt{=} \; \mathtt{E}_2, \\ \quad val(Calc)\backslash val(Calc) \cup \\ \qquad \mathbf{E}[\![\mathtt{E}_1 = \mathtt{E}_2]\!](n) = \mathbf{E}[\![\mathtt{E}_1]\!](n) \; equals \; \mathbf{E}[\![\mathtt{E}_2]\!](n) \rightarrow zero \; [] \; one] \end{array}$$

Our amalgamated product, *Calc_New*, is a refinement of *Calc*, because the expression above defining the new viewpoint uses only the operations defined at the beginning of Section 4.1, which satisfy our refinement definition.

The feature added could not be said to be very complicated, however. We next take into account a more complex feature to be added to the calculator, involving a change to the semantic algebras of the description.

4.2 Changes to Semantic Algebras and Valuation Functions

Our next example, which adds an assignment operation to the calculator, is a more involved one that requires changes to the semantic algebras and valuation functions. Specifically, for this change the physical calculator will need a number of extra buttons for identifiers, to be called `A`,`B`,`C`,`D`, and a `:=` button (for "becomes", or "takes the value"). New expressions would be of the form:

```
A := 5 + 6
B := IF (A=4),3,2
D := (A + 4) * (A + 5)
```

Such an increment needs a number of changes:

- New syntax domain for identifiers, with BNF rule. A decision needs to be made as to whether `I := E` is a normal expression or not. Expressions of the form `(A := 5) + A` would be legal in some languages — for example in C, the statement `b = (a=5) + a;` assigns the value `5` to `a` and `10` to `b`. For the calculator this would over complicate matters, so we will identify `I := E` as a special kind of expression by altering the syntax for expression sequences *S* to include `I := E TOTAL S`.
- New semantic domain, *Store*, a function from identifiers to natural numbers, with operations to create an empty store, read and update the store
- New valuation functions for identifiers and assignments. An identifier is simply an expression, whose meaning is the natural number in the store referenced by the identifier. The meaning of an assignment is to update the store for a given identifier.

- New signatures for expression sequences S and expressions E, which will need be altered to take into account the store, rather than just the value held in the simple LASTANS store. It is natural to explain the store as an extension of the single-cell memory we already have. The valuation functions for expression sequences and expressions will change from

  $\mathbf{S}$: `Expr-Seq` $\rightarrow$ `Nat` $\rightarrow$ `Nat`*
  $\mathbf{E}$: `Expr` $\rightarrow$ `Nat` $\rightarrow$ `Nat`

  to

  $\mathbf{S}$: `Expr-Seq` $\rightarrow$ (`Nat` $\times$ `Store`) $\rightarrow$ `Nat`*
  $\mathbf{E}$: `Expr` $\rightarrow$ (`Nat` $\times$ `Store`) $\rightarrow$ `Nat`

  since both the last answer and the current store are to be maintained.

These changes will allow a program sequence such as the following:

Input	Display
`10 + 5 TOTAL`	15
`A := LASTANS TOTAL`	15
`LASTANS + 2 TOTAL`	17
`B := A * 10 TOTAL`	150

In order to make these changes we will need more composition operators than those already given. The first applies to the semantic algebra:

- A new domain can be added to the algebra by stating its signature and operations; as long as the names of the domain and its operations are not already used in the viewpoint, this operation will not affect the rest of the viewpoint (compare the addition of local variables in the refinement calculus). So for any semantic viewpoint S and new semantic domain D,

  $S \sqsubseteq S[alg(S) \backslash alg(S) \cup D]$ provided D is free in S.

  This operation has the necessary property of monotonicity, because $alg(S) \subseteq (alg(S) \cup D)$.

- Changing valuation functions. We have already seen that valuation functions can be extended. We can also transform them, by changing their ranges. This involves the use of an abstraction function from the old range to the new one: i.e. we replace a valuation function $\mathbf{F} : D \rightarrow X \rightarrow Y$ by $\mathbf{F'} : D \rightarrow X' \rightarrow Y'$ with an abstraction function $a : (X' \rightarrow Y') \rightarrow (X \rightarrow Y)$, where for an arbitrary piece of abstract syntax `d`,

  $\mathbf{F}[\![\mathtt{d}]\!] = a(\mathbf{F'}[\![\mathtt{d}]\!])$

  If a suitable abstraction a can be found, the valuation function F can be exchanged for F'. However, F is used in function composition by other valuation functions, and may use other valuation functions in its definition: these may also need to be changed if F is replaced by F'. We have to make some limitations on the form of F', as follows:

  We define the *top-level* syntactic domain of a denotational description as the (unique) domain which does not feature as an non-terminal (i.e. on the right-hand-side) in the BNF of any other syntactic domain. Thus in the calculator example `P` is the top-level syntactic domain. A top-level valuation function is the valuation function for the top-level domain.

  We require for our refinement definition that for semantic descriptions S and S', any program in the implementation of S is also a valid program, with the same meaning, in the implementation of S'. The meaning of a program P in S is defined as $S(P)$, which is the return value of the top-level valuation function of S when applied to P. Thus our definition is equivalent to $\mathbf{T}_S(P) = \mathbf{T}_{S'}(P)$ for all programs P in the implementation of S.

  The general form of $\mathbf{T}_S$ is $\lambda\, d.\Phi(\mathbf{F}_i(d_i))$, where the $\mathbf{F}_i$ are valuation functions called on sub-expressions d_i of the syntactic expression d, and Φ is the function applied to the results of these valuations. If any of the F_i are changed we require that it is possible to change Φ to keep $\mathbf{T}_S$ constant. Thus:

Abstract Syntax:

S $\in$ Expr-Seq
E $\in$ Expr
I $\in$ Identifier

S ::= I := E TOTAL S $|$ I := E TOTAL OFF
E ::= I

Semantic Algebras:

Natural Numbers
...

Identifiers
Domain $i \in Id = Identifier$

Store
Domain $s \in Store = Id \rightarrow Nat$
Operations
$init\_store : Store$
$init\_store = \lambda\, i.\ zero$
$read\_store : Id \rightarrow Store \rightarrow Nat$
$read\_store = \lambda\, i.\, \lambda\, s.\ s(i)$
$write\_store : Id \rightarrow Nat \rightarrow Store \rightarrow Store$
$write\_store = \lambda\, i.\, \lambda\, n.\, \lambda\, s.(\lambda\, j.\ j = i \rightarrow n\ []\ s)$

Valuation Functions
$\mathbf{S}$:Expr-Seq$\rightarrow (Nat \times Store) \rightarrow Nat^*$
$\mathbf{S}[\![\texttt{I := E TOTAL S}]\!] = \lambda(n,s).$
$let(n', s') = (\mathbf{E}[\![\texttt{E}]\!](n,s), write\_store[\![\texttt{I}]\!]\ \ \mathbf{E}[\![\texttt{E}]\!](n,s)\ \ s)$
$in\ \ n'\ \ cons\ \ \mathbf{S}[\![\texttt{S}]\!](n', s')$
$\mathbf{S}[\![\texttt{I := E TOTAL OFF}]\!] = \lambda(n,s).$
$let(n', s') = (\mathbf{E}[\![\texttt{E}]\!](n,s), write\_store[\![\texttt{I}]\!]\ \ \mathbf{E}[\![\texttt{E}]\!](n,s)\ \ s)$
$in\ \ n'\ \ cons\ \ nil$

$\mathbf{E}$:Expr$\rightarrow (Nat \times Store) \rightarrow Nat$
$\mathbf{E}[\![\texttt{I}]\!] = \lambda(n,s).read\_store[\![\texttt{I}]\!]s$

Figure 4: *Assign*: Incremental viewpoint for the addition of assignment

$\exists\, \Phi'.\ \Phi(\mathbf{F}_i(d_i)) = \Phi'(\mathbf{F'}_i(d_i))$

The definition of $\mathbf{T}_S$ can then be changed to $\mathbf{T'}_S = \Phi'(\mathbf{F'}_i(d_i))$, and since $T_S = T'_S$, the new semantic description refines the old one.

The effect of this restriction on F' and the creation of Φ is to reconstruct the monotonicity that is otherwise lost by the transformation.

4.2.1 Incremental Viewpoint

Figure 4 shows the incremental viewpoint *Assign* representing the changes we need to make to the abstract syntax. The subsequent amalgamation of viewpoints then requires the following actions:

- Abstract Syntax: `I` $\in$ `Identifier` is a new domain. `S` $\in$ `Expr-Seq` and `E` $\in$ `Expr` are defined in *Calc_New*, but *Assign* gives them new BNF forms. There is no conflict here so the syntax of the amalgamation will be formed by $syn(Calc\_New) \cup syn(Assign)$. New valuation functions need to be given for the new syntax.
- Semantic Algebras: The domain *Nat* is identical in both viewpoints. *Id* is new in *Assign*. The semantics can be formed by $alg(Calc\_New) \cup alg(Assign)$.

- Valuation Functions: Functions S and E have different signatures in *Assign* from those in *Calc_New*. Comparing $val(Calc\_New)$ with $val(Assign)$ shows that further changes are necessary: each of the equations for S and E in $val(Calc\_New)$ must be altered to have the same signature. We have defined those expressions that actually *use* the store; others will simply pass it along with the last answer memory.

  So the amalgamation action is twofold: first the functions from the old viewpoint must be updated to have appropriate signatures for the new one, and then the amalgamated viewpoint is formed by collecting these functions.

We change the functions for **S** and **E**, giving them new signatures:
S :Expr-Seq$\rightarrow$ $(Nat \times Store) \rightarrow Nat^*$
and
E :Expr$\rightarrow$ $(Nat \times Store) \rightarrow Nat$.
The abstraction function a_S is defined for **S** as:

$$a_S : ((Nat \times Store) \rightarrow Nat^*) \rightarrow (Nat \rightarrow Nat^*)$$
$$a_S = \lambda f.\ \lambda n.\ f(n, init\_store)$$

and a_E is defined similarly for **E**:

$$a_E : ((Nat \times Store) \rightarrow Nat) \rightarrow (Nat \rightarrow Nat)$$
$$a_E = \lambda f.\ \lambda n.\ f(n, init\_store)$$

The old versions of **E** and **S** are equivalent to $a_E(\mathbf{E'})$ and $a_S(\mathbf{S'})$, for new versions **E'** and **S'**. We now need to consider the effect on the other valuation functions in the viewpoint.

The only other valuation function that features as part of the definition of **E** is **N**. This definition will not need to be changed as the *Nat* argument was not passed to **N** in the original viewpoint.

This leaves only the top-level valuation function **P**, which must define an equivalent meaning for any program, between old and new versions. The old definition of **P** is:

$$\mathbf{P} : \text{Program} \rightarrow Nat^*$$
$$\mathbf{P}[\![\text{ON S}]\!] = \mathbf{S}[\![\text{S }]\!](zero)$$

Due to the signature change in **S'**, the right hand side of this definition must now provide a $(Nat \times Store)$ argument. The function we choose for Φ is, by no coincidence, similar to our abstraction function: it maps the `(zero)` to the corresponding value in the new domain, `(zero,` *init_store*`)`. Our new **P** is therefore

$$\mathbf{P}' : \text{Program} \rightarrow Nat^*$$
$$\mathbf{P}'[\![\text{ON S}]\!] = \mathbf{S}'[\![\text{S}]\!](zero)$$

The amalgamation step then consists simply of assembling the new functions, and removing the $'$ decorations.

4.3 Conversion from Direct to Continuation Semantics

Finally we outline how a more radical change to the whole structure of the semantic description might be shown to be valid using our overall approach. Up to now we have been using *direct* semantics. Such semantics emphasise the structure of a language; an equation like

$$\mathbf{E}[\![E_1 + E_2]\!](n) = \mathbf{E}[\![E_1]\!](n) \text{ plus } \mathbf{E}[\![E_2]\!](n)$$

makes no attempt to prescribe the intended order of evaluation. In some applications however the concept of concept of *control* is necessary to express order of evaluation; it can be modelled using a semantic construct called a *continuation*.

This allows modelling of control flow as presented in "goto" jumps, returns from procedures, errors or interrupt handling. *Continuation* semantics incorporates this idea, representing the "rest of the program" as a continuation (a function from machine-state to machine-state, taking the current state to the state at the termination of the program); this provides a way to override the "normal continuation" for jumps, error handling and so on.

Standard denotational semantics texts [2, 5] deal with the complexity of the continuations issue by first explaining direct semantics and then motivating the addition of continuations, in a manner similar to the discussion in the previous paragraph. This is clearly an incremental change such as we have been discussing.

The general process followed in texts is to present the semantics of some command in direct semantics and introduce a *command continuation*, using it to replace the direct semantics version with one in continuation style; in the case of [5] this is via a direct semantics which involves a command stack, which is replaced by the continuation function. Similarly expression semantics are altered to include the order of evaluation.

The question of the relationship between direct and continuation semantics is quite complex — proving that two definitions describe the same program is made difficult by the different structures of the definitions. A better way, coinciding with our approach to explanations here, would be to derive a continuation semantics definition from a direct one.

As an example, suppose we have a valuation function

$$\mathbf{C}_D : \mathit{Command} \to \mathit{Store}_\perp \to \mathit{Store}_\perp$$

in direct semantics. Adding continuations to this valuation function we would expect to result in

$$\mathbf{C}_C : \mathit{Command} \to \mathit{Cmdcont} \to \mathit{Cmdcont}$$

where the command continuation is $\mathit{Cmdcont} = \mathit{Store}_\perp \to \mathit{Store}_\perp$.

Sethi and Tang [6] proposed a method for constructing continuation semantics from a direct semantics. Their method is to define, for each construct $[\![\mathrm{C}]\!]$, the continuation semantics from the direct semantics. An overview of the method follows.

If F and F' are domains of function values in direct and continuation semantics respectively, it is possible to define transformations for each term T in F to derive a corresponding T' in F'. The derivation is done in small stages by using an auxiliary operator δ as a syntactic place-holder, starting with δT, which represents the (as yet undefined) continuation semantics version of T in F', and 'pushing' δ onto smaller sub-expressions of T using transformation rules.

To show that T and T' represent related values, all intermediate expressions in the derivation have meanings in either the direct or continuation domain. Representing a term as a tree, any sub-term below δ represents values from the direct domain and the rest represent values from the continuation domain. Thus δ is used as a bridge between the direct and continuation worlds to enable the derivation to be made a step at a time. We can make a comparison between this and the progression from specification to program; the refinement calculus uses the specification statement as a bridge for a similar reason.

A predicate is defined to test the equivalence of values from each of the domains. For direct value $v \in V$ and continuation value $w \in K \to A$, the values satisfy the equivalence predicate if either both are $\perp$, both $\top$ or, for v' equivalent to v, $w = \lambda k.k(v')$ for expression continuation k. This predicate is written $\mathbf{pk}(v, w)$.

In each step of the transformation from a $v \in V$ to a $w \in K \to A$ from $v \in V$, the equivalence predicate is maintained. The method is an incremental development, of the same nature as we have been modelling in this chapter. We can use the stipulations we make about refinement to verify that the method is acceptable for generating a correct continuation term T' from a direct term T.

- **Correctness**. We define correctness as follows: for two terms T, T', T' is correct with respect to T if the equivalence predicate defined above is satisfied ($\mathbf{pk}(T, T')$).

- **Monotonicity**. The δ function has the following property:

  For term T consisting of subterms T_1, T_2, we write their composition as $T = T_1 \oplus T_2$ for monotonic operation $\oplus$. Then $\delta T = \delta(T_1 \oplus T_2) = \delta(T_1) \oplus \delta(T_2)$.

  This property is necessary for the "pushing" method of term derivation, and also gives us the property of preservation of monotonicity we need.

- **Refinement relation**. We can define a refinement relation as follows:

  For some terms v in the direct domain and w in the continuation domain such that $\mathbf{pk}(v,w)$, if w' is obtained from w by transformation operations and $\mathbf{pk}(v,w')$, then w is refined by w'.

  This refinement relation maintains correctness and is reflexive (w is obtained from w by the identity transformation) and transitive (since each transformation maintains the predicate).

So the process of transformations outlined above is valid as a refinement method, as it has the necessary properties we have been discussing.

5 Conclusions and Related Work

Viewing denotational descriptions as formal descriptions of programming languages, we defined a notion of refinement between denotational descriptions that captures the idea that one language is a refinement of another exactly when the constructs of the first language are featured in the second with equivalent abstract syntax and valuation functions. This relation was shown to be reflexive and transitive, giving it the required formal properties for a refinement relation.

Given our refinement relation, we then indicated using examples how we could show that the kinds of transformation typically used to derive one language description from another in an introductory presentation of denotational semantics preserve the refinement relation. This allows us to conclude that every denotational description in such a sequence is a refinement of all the previous ones. In other words, we have shown that a whole sequence of denotational descriptions connected by transformations of the kind we have have been considering has the property that each constituent description of the sequence refines all those that precede it. The work applies not only to simple sequences of consecutive descriptions as shown in Figure 1 but also embraces more elaborate tree-like structures of descriptions like that shown in Figure 2.

Structures relating descriptions in these ways are not peculiar to denotational descriptions, and indeed other recent work [4] applies the same basic ideas to sequence of specifications of software systems. Apart from the obvious difference that it concerns software specifications rather than denotational descriptions, that work differs in two technical respects from that described here.

1. The relationship between successive specifications in that sequence is not exactly a refinement relation but a closely related kind of relation called co-refinement [1]. Co-refinement, unlike refinement, models the effect of interactions or restrictions between the specifications that are amalgamated to produce the combined product: restrictions are crucial to the example discussed in [4] but are not relevant to denotational semantic descriptions, where refinement suffices. In [4] we introduce the term “integrated” to describe specification sequences in which every specification is co-refined by all those that follow it[1].

2. The co-refinements established between different descriptions in [4] are verified individually whereas in the present work they hold as a consequence of the operations used to combine denotational descriptions.

Both these projects–the one described here and the one that forms the example in [4]–form parts of a larger, ongoing research project whose overall aim is to gain enhanced theoretical understanding of the way complicated systems are described in simple terms. This is an important general problem in many different branches of computer science, since computer-based systems tend by their nature to be extremely complicated and hard to comprehend in simple terms. Formal studies of the modularisation capabilities of formal languages such as those for programming and software specification are planned as future components of this work.

6 Acknowledgements

The comments of the referees are gratefully acknowledged. Peter Wallis is grateful to Wolfson College, Oxford for continuing hospitality. This work of Steve Riddle was largely supported by an ESPRC Research Studentship held at the University of Bath.

[1] Since co-refinement and refinement are equivalent in cases like those of denotational descriptions where there is no interference, this permits us to describe as “integrated” the sequences of denotational descriptions described in the present paper.

References

[1] M Ainsworth and PJL Wallis. Co-refinement. In D Till, editor, *Proc. 6th BCS-FACS Refinement Workshop*, pages 151–166, City University, London, 5th–7th January 1994. Springer-Verlag.

[2] Michael J C Gordon. *The denotational description of programming languages.* Springer-Verlag, 1979.

[3] S Riddle. *Viewpoints in Practice*. PhD thesis, University of Bath, 1997.

[4] S Riddle and PJL Wallis. Checking the integrity of an incremental specification. 1997. Submitted for publication.

[5] DA Schmidt. *Denotational semantics: a methodology for language development*. Allyn and Bacon, 1986.

[6] R Sethi and A Tang. Constructing call-by-value continuation semantics. *Journal of the ACM*, 27(3):580–597, July 1980.

[7] JE Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[8] C Strachey. Towards a formal semantics. In T Steel, editor, *Formal language description languages*, pages 198–218, Amsterdam, 1966. North-Holland Publishing Company.