Vrije Universiteit Brussel
Faculteit Wetenschappen

An Informal Tour on Denotational Semantics

De Meuter Wolfgang
Niels Boyen

Techreport vub-prog-tr-94-08

Programming Technology Lab
PROG(WE)
VUB
Pleinlaan 2
1050 Brussel
BELGIUM

Fax: (+32) 2-629-3495
Tel: (+32) 2-629-3308
Anon. FTP: progftp.vub.ac.be
WWW: progwww.vub.ac.be

An Informal Tour On Denotational Semantics

Wolfgang De Meuter Niels Boyen

Brussels Free University
Department of Computer Science
Programming Technology Lab
Pleinlaan 2
1050 Brussels
e-mail
wdmeuter@vnet3.vub.ac.be
nboyen@vnet3.vub.ac.be

ABSTRACT

This paper presents a semi-formal overview of the most important topics from the theory of denotational semantics. The emphasis has been laid on the natural introduction of most concepts of the semantic realm. Since this theory is usually hard to understand and accept in a seemingly intuitive discipline as computer science, we try to introduce them from a computer scientist point of view. By this, we mean that most mathematical constructions will be presented in a manner as if they were trivially connected to reasoning about real computer programs. One of the most important intentions of this paper is to give the reader a brief overview of the existing results, while laying adequate pointers to relevant literature. We hope that the way in which the material is presented, will contribute to the popularization of semantics theory[1].

1. Introduction

Usually, a programming language is characterized by three main descriptions: its syntax, its semantics and its pragmatics. The syntax of the language describes the appearance and the structure of its sentences. Its semantics on the other hand gives a meaning to those sentences. Finally, the pragmatics of the language describes the usability of the sentences. Topics involving pragmatics are the ease of implementation of the language, the domains of its applicability and so on. For short, one could state that the syntax describes how to *form* right sentences, the semantics outlines how to *assign a meaning* to those sentences and the pragmatics describes how to *use* those sentences. Because this is a paper about formal semantics, the first and last topic will not be further discussed. Pragmatics is by no means formalisable, and for a formal or informal treatment on syntax, we refer to a standard textbook about formal grammars such as [Hopcroft79] or [Rayward-Smidt84].

Why should anybody be concerned with a formal study of the semantics of programming languages? In fact, several reasons can be quoted by different people, each having a different view on the meaning of programs. For example: a programmer might desire a formal specification with which he can construct correctness proofs about the programs he has written. The language designer, on the other hand, might want a notion of semantics to be able to determine whether the main constructions of his language behave in an orthogonal manner. And last, the implementor of the language could require a formal specification in order to ensure the correctness of his compiler. A bit more taxonomically, one could list the arguments in favor of formal semantics as follows:

[1] It is important to keep in mind that this paper is about formal semantics of *deterministic sequential* program constructs. The introduction of parallellism and nondeterminism requires some important changes to the theory. The resulting theory is somewhat harder to comprehend. An extensive treatment of both concurrency and nondeterminism can be found in [Hennessy88].

• A formal specification of the behaviour of programming language constructs allows correctness proofs of the language's implementation, which could enable a compiler designer to state that his compiler formally corresponds to the definition of the language.
• Formal semantics might, in a declarative way, support a formal reasoning system about the mechanisms a language contains. These reasoning systems consist of so called partial correctness proofs, and a bunch of inference rules allow the programmer to combine them into correctness proofs for his entire program.
• Another point at which formal semantics might help a user of a language is its definition. Up until now, the semantics of most languages is still defined by informal statements, and experience has shown that almost every language ever described by means of natural language, contains at least a few holes in the specification of its semantics. Apart from the troubles in implementing such a language, as a consequence, programmers often tend to create subtle bugs in their minds about some of the languages features. The only way to avoid this is to formally specify the meaning of those features.
• The last good reason to study formal semantics is the generation of compilers and interpreters. By analogy to the generation of syntax checkers, it is not difficult to see that a formal specification of semantics could easily lead to construction tools that translate semantic constructs into executable code which is able to do the semantic checking stage of a compiler.

However, at the time of writing(1994), the semantics area is not as well developed as the syntax area. Standard methods for describing formal semantics of programming languages are still evolving. Meanwhile, we hope that this paper will contribute to the popularization of the currently established results.

The structure of the paper is as follows. In the following section, we will give a general introduction and taxonomy of the realm of semantics. In section 3, we will try to provide a quick overview of the denotational approach. One of the central research issues in the seventies was the characterization of the so called semantic domains. Once found, they seemed to have many artificial properties. In section 4 we try to introduce them from a computer scientist point of view. In section five we formalize the concepts introduced in section 4 and give methods for the modularization of semantic definitions. Section 6 applies these mathematical definitions to some important language constructions. The final section briefly discusses the relations between the several approaches.

2. An Overview of the Semantic Landscape

As already mentioned in section 1, not everyone has the same idea as to what a formal description of programming languages should be. As a consequence, different researchers started to design their formalisms with different intentions, and therefore several approaches to formal semantics have been proposed. For the moment, these can be classified into three different tendencies: operational, denotational and axiomatic semantics. It is important to know that these approaches should by no means be viewed as competitors. They were one by one designed to serve different purposes and their theoretical equivalence has been shown more than ten years ago. However, from a 'practical' point of view, it is meaningless to compare them to each other, and arguing about which of them should be seen as the best, can be safely called an idiotism. This should be clear by the following brief overview.

• Operational semantics
Operational semantics is based on an evolutionary model. One defines a semantic function $\mathfrak{F}$ and a class of abstract machines $\mathfrak{M}$ such that every element of the set of valid syntactic constructions $\mathcal{S}$, can be mapped by $\mathfrak{F}$ to some abstract machine in $\mathfrak{M}$, that is $\mathfrak{F} : \mathcal{S} \to \mathfrak{M}$. According to the operational approach, the meaning of the elements in $\mathcal{S}$ is therefore specified in terms of the computations or operations performed by the machine when executing the construct.
The machines from $\mathfrak{M}$ are usually defined as couples (S,P) where P represents the part of the program remaining to be executed and where S represents the internal state of the machine. The execution of the abstract machines is formalized by a set of reduction rules which transform couples. The meaning of the initial program is then seen as the complete reduction; i.e. the evaluation history of the program.
The main benefit coming from this approach is the preciseness with which the execution of the program is specified. It is therefore strongly advocated by people who write compilers and interpreters for languages. The correctness of a compiler can thus be shown by defining a one-to-one correspondence between the abstract machine and the language to which the program is transformed. The most important disadvantage of the operational paradigm is that for nontrivial languages, the abstract interpreter usually gets very large and complex, and thus becomes difficult to understand. This is why

the paradigm is not very well suited for programmers and language designers. A beautiful introduction to this method is presented in [Plotkin].

• Denotational semantics
Denotational semantics is based on a denotationary model. In this paradigm, a phrase is mapped directly onto its meaning, called its denotation. To accomplish this, one defines a semantic function $\mathfrak{F}$ and a semantic domain $\mathfrak{D}$ such that every syntactic construct in $\mathcal{S}$ is mapped by $\mathfrak{F}$ onto some element of $\mathfrak{D}$, which is a structured set of abstract values (truth values, natural numbers, natural number functions,...); i.e. $\mathfrak{F} : \mathcal{S} \to \mathfrak{D}$. Hence, one tries to *denote* the meaning of a program by means of some mathematical object without going into the details of how it is actually executed. A widely used approach is to simply define the denotational semantics of a program as a *function* relating input and output values without stating its implementation details.
This paradigm should be used as a tool for designing languages. The elegance of a language is often measured by the length of its denotational definition.
Denotational semantics can be useful to users of a language as well, because it clearly shows the essence of a language feature, without specifying how these features are physically realized.
The denotational paradigm is sometimes referred to as the Scott-Strachey-approach, named after the inventors of the method. An extensive treatment of it can be found in [Schmidt88].

• Axiomatic semantics
Axiomatic semantics is based on an inference model. One defines a set of predicates which can contain the free variables of the program phrase. These predicates can thus serve as statements saying something about the contents of those variables before and after the execution of the program segment. Axiomatic semantics therefore consists of a proof system which contains axioms and inference rules. The axioms usually are triples of the form {Pre}P{Post} where P is a program segment and Pre and Post are logic formulae serving as pre and post conditions that hold for P. The inference rules depend on the kind of construct P belongs to. For example, {}while E do C{E=false} could be a simple axiom about the Pascal-while-statement. Evidently, it is this kind of semantics that is useful for a programmer to construct partial correctness proofs for his programs.
The main objective of an axiomatic semantics is to lay down such a rule for every possible program construct of a language, and to prove that this set of rules is a sound and complete reasoning system.
Axiomatic semantics was mainly advocated by Hoare. An excellent introduction can be found in [deBakker80].

Recently, a generalization of two of the above approaches has been proposed. This is sometimes referred to as algebraic semantics. Algebraic semantics introduces category theory into denotational semantics. By comparing different denotational domains, it allows us to construct models that can serve as a basis for a sound and complete axiomatic semantics. Algebraic semantics thus acts as a bridge between the denotational and the axiomatic approach. We like to see this as a part of the denotational approach with some nice side effects in the axiomatic paradigm.

3. The Denotational Approach

As explained in the previous section, the general idea of the denotational approach is to denote each syntactic construct by some abstract value chosen from a domain of denotations. This will be accomplished by a function that maps language sentences to their meanings. This function will be called the *semantic valuation function*.
We assume that the syntax of our languages is defined by a context free grammar. Whenever we want to replace a subphrase of a sentence by a phrase which has a meaning identical to the meaning of the original subphrase, we want the meaning of the entire phrase to be preserved. As a consequence, the meaning of a phrase should only depend on the meaning of its constituents. This is called *compositionality*. Compositionality will most easily be reached by structuring the semantic valuation function according to the structure of the BNF definition of the language. In fact, it is argued in [Goguen77] that structuring the semantic valuation function according to the context free grammar is crucial to denotational semantics. An important consequence of compositionality is that it allows properties of semantic valuations to be proved by *structural induction*.
We keep in mind that the semantic valuation function is based on BNF and compositionality. A denotational definition consists of a syntax definition, a semantic valuation and a *semantic algebra*. The semantic algebra is a domain of semantic values together with some operations on it.
The question now arises of which kind of mathematical structures are appropriate to be used as semantic domains. For the moment, we will let our intuition do its job, and take sets as the first

candidates for being semantic domains. For example, let us create a simple language to denote additive expressions.

```
<digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<number> → <digit> | <number><digit>
<term> → (<expression>) | number
<expression> → <term> | <term><add-op><expression>
<add-op> → plus | minus
```

The semantics of this language could simply be the tuple ($\mathbb{N}$, {+,-,*} , $\mathbb{N}$) where $\mathbb{N}$ is the usual set of natural numbers, +, -, * are the usual operators on natural numbers and $\mathbb{N}$ is defined as follows[2]:

$\mathbb{N}[\![0]\!] = 0$, $\mathbb{N}[\![1]\!] = 1$, ... , $\mathbb{N}[\![9]\!] = 9$

$\mathbb{N}[\![\text{<number><digit>}]\!] = \mathbb{N}[\![\text{<number>}]\!] * 10 + \mathbb{N}[\![\text{<digit>}]\!]$

$\mathbb{N}[\![\text{ (<expression>) }]\!] = \mathbb{N}[\![\text{<expression>}]\!]$

$\mathbb{N}[\![\text{<term>}\textit{plus}\text{<expression>}]\!] = \mathbb{N}[\![\text{<term>}]\!] + \mathbb{N}[\![\text{<expression>}]\!]$

$\mathbb{N}[\![\text{<term>}\textit{minus}\text{<expression>}]\!] = \mathbb{N}[\![\text{<term>}]\!] - \mathbb{N}[\![\text{<expression>}]\!]$

Hence, every additive expression is denoted by some natural number. For example, both the expressions *7plus4* and *((3plus14)minus6)* are denoted by the natural number 11 $\in \mathbb{N}$.
The semantic algebra that is used in this example consists of the set of natural numbers together with the usual operators +, - and *. The conversion from ((3*plus*14)*minus*6) to ((3+14)-6) is done by the semantic valuation function. The latter expression is further simplified by the rules of the underlying semantic algebra. The purpose of simplification is calculating the *actual semantic value* of the phrase. For example, the actual semantic value of *((3plus14)minus6)* is not ((3+14)-6) but 11.

No trouble emerged from this example and therefore we conclude that sets can be perfectly suitable to play the role of being semantic domains. However, this is not always the case. This will be discussed in the following section.

4. Semantic domains: some requirements

Let us start thinking about some pathological cases that may occur while designing a denotational semantics.
First, a problem of having a meaning at all arises. For example, we would not want the expression 3+true to have any meaning. Therefore, a notion of 'has no meaning' should be introduced. This can be done in two different ways. First, we could require that the semantic function ($\mathbb{N}$ in the example above) is undefined in some cases. Second, we could include the meaning 'no meaning' to our semantic domain. Although both approaches yield more or less the same theory, most authors prefer to work with an explicit 'no meaning'-denotation. This denotation is given by $\perp$ and is pronounced *'bottom'*.
Second, it turns out that sets are not enough for designing semantic domains for general purpose programming languages. The most important problem that forces us to leave the nice and clean world of sets is recursion or iteration. It should be emphasized that by recursion[3], we actually mean *variable recursion* and not primitive recursion. To illustrate the problems arising from recursion, let us look at the following procedure definition in some Algol-like language

```
function f(n:natural): natural;
begin
        if n=0 then f(n):=1 else f(n):=f(n+1);
end;
```

Clearly, we would like the semantics of this construct to be a function g: $\mathbb{N} \to \mathbb{N}$ which satisfies g(0)=1 and g(x) = g(x+1) for each x≥1. A solution for this problem could be the function h: $\mathbb{N} \to \mathbb{N}$ such that h(x) = 1 for all x in $\mathbb{N}$. However, every function satisfying g(0) = 1 and g(x) = a for x≥1 and some constant a is a valid meaning of the above construct. How should we solve this one ? It should be clear that the situation forces us to compare every function with one another. After the comparison is made, we have to start looking for the one which seems to be the best among all functions.

[2]The use of $[\![$ and $]\!]$ is historically grown. These double brackets separate the realm of syntax from the realm of semantics.
[3]From now on, we will treat recursion and iteration identically. This is a commonly used technique when modeling computer programs.

To be able to compare the functions, we will install some kind of order $\sqsubseteq$ on the set of functions. This order will be the formalization of the notion "contains more information". Hence, f $\sqsubseteq$ g whenever g contains more information than f. This order can never be total since there will always be functions containing exactly the same amount of information (consider e.g. f(x) =1 and g(x) = 2)[4]. Therefore the order will be a *partial order*. Now let S be the set of all solutions of the problem stated above. S will be the set of all functions satisfying f(0)=1 and f(x)=f(x+1) for x≥0. Since a computer will never be able to invent meaning itself, the meaning of a program will be the meaning which contains a minimum of information. Therefore, the meaning of the above construct, will be the *least element* of S. It is clear that this is the function g: $\mathbb{N} \to \mathbb{N}$ satisfying g(0) = 1 and g(x) = $\perp$ for x≥1, because this literally states that g has no meaning for every x≥1. Compatible with ordinary computability theory, $\perp$ therefore denotes the concept of undefinedness or nontermination, which is what we like because it is clear that if we run the code segment stated above, the resulting procedure will be nonterminating, for each input x≥1.
So far, we conclude that our semantic domains should at least be partial orders.
Because it is also possible to write a procedure that has no meaning at all, for example,

```
function f(n:natural): natural;
begin
        f(n):= f(n+1);
end;
```

another argument is found to indicate that our partial orders should also contain a bottom element, which stands for an undefined piece of code. However, it will become clear that partial orders containing $\perp$ will still not satisfy our needs. To illustrate this, let us consider the following example. Let

```
function fac(n:natural): natural;
begin
       if n=0 then fac(n):=1 else fac(n):= n*fac(n-1);
end.
```

be an expression for the well-known factorial function. The denotation for this phrase is the function

$$f: \mathbb{N} \to \mathbb{N} \text{ given by } f(n) = n!$$

It is common to think of this function using its graph,

$$\text{graph}(f) = \{(0,1),\ (1,1),\ (2,2),\ \dots,\ (i,i!),\ \dots\}.$$

This graph is an infinite set. The Cartesian methodology for understanding an infinite object is to define finite objects which are its finite subparts, and then building up, step by step toward the object. Let us therefore take functions $\text{graph}_{i+1} = \{(0,1),\ (1,1),\ (2,2),\dots,\ (i,i!)\}$. Such a function can be considered as one that is capable of unfolding the factorial definition at most i+1 times, and returning the appropriate result. When a number larger than i is tried, the function is undefined. It is clear that $\text{graph}_i \subseteq \text{graph}_{i+1}$ and $\text{graph}_0 = \emptyset$.
Further, for all i, $\text{graph}_i \subseteq \text{graph}(f)$. This implies that $\bigcup_{i=0}^{\infty} \text{graph}_i \subseteq \text{graph}(f)$.
Conversely, when $(a,b) \in \text{graph}(f)$, there will be some i such that $(a,b) \in \text{graph}_i$.
Therefore, $\text{graph}(f) \subseteq \bigcup_{i=0}^{\infty} \text{graph}_i$ and hence $\text{graph}(f) = \bigcup_{i=0}^{\infty} \text{graph}_i$.
What we found so far is that if we want the set of graphs defined here to be a semantic domain, it should in some way contain its limits. Expressed as in ordinary calculus, we say that our domains will have to be *complete* in some way. As usual, completeness means that the upper bound of every row of elements should be included. Hence, our domains should be complete partial orders (cpos) with a bottom element. Notice that the problems addressed here would not arise if it were not for variable iteration. Notice also that this example is an instance of our general methodology developed so far. The semantic domain used here is the set of (graphs of) functions[5] from $\mathbb{N}$ to $\mathbb{N}$. The partial order $\sqsubseteq$

[4]This is a good example to show that information value is fundamentally different from information content.
[5]Later, we will see that the set of all functions from $\mathbb{N}$ to $\mathbb{N}$ is not a valid domain, because there are too many functions in this set.

discussed above is given by the inclusion relation $\subseteq$. This relation meets the requirements because $graph_{i+1}$ contains more information than $graph_i$. The upper bound of a sequence of graphs is the infinite union of that sequence. Moreover, another argument for the presence of $\perp$ has been found (this is Ø in our example, because Ø contains no information, i.e. $Ø \subseteq graph_i$ for all $i \geq 1$).

An important topic which was also addressed by this example, is a general construction methodology for:

Semantics of recursive program constructs

We already explained that $graph_i$ is the factorial function unfolded for i times. Hence,

$$graph_{i+1} = \lambda n.\ Cond_{\mathbb{N}}\ (n\ ,\ 1\ ,\ n*graph_i(n-1)\ ) \qquad \text{for all } i \geq 0. \qquad [*]$$

where $Cond_A : \mathbb{Z} \times A \times A \rightarrow A$ is the conditional, returning the first element if the condition is zero and the second otherwise[6], i.e.

$$\begin{cases} Cond_A\ (0,a,a') = a \\ Cond_A\ (x,a,a') = a' \text{ for all } x \neq 0 \end{cases}$$

If we define[7] $F_S = \lambda f.\lambda n.\ Cond_{\mathbb{N}}\ (n=0,\ 1,\ n * f(n-1)\ )$, then equation [*] can be rewritten as

$$graph_{i+1} = F_S(graph_i) = ... = F_S{}^i(graph_0) = F_S{}^i(Ø)$$

F_S is a function taking a function as an argument and returning a function as result. Such a function is called a *higher order function* or a *functional*.

Now $graph(f) = \bigcup_{i=0}^{\infty} graph_i$ can be rewritten as $graph(f) = \bigcup_{i=0}^{\infty} F_S{}^i(Ø)$

More general, we can state that if r is a recursive specification such that r = F(r) then we define the semantics of r as the least upper bound of the semantics of r's finite approximations

$Semantics(r) = \bigcup_{i=0}^{\infty} Semantics(F)^i(Ø))$. In the case of subsets, the least upper bound is known as an infinite union. In general, we will denote it by $\sqcup$. The bottom element (Ø in this example) is used as a starting point for the unfoldings. The entire concept of the meaning of recursion is based upon it.
Notice that it can be shown that

$$Semantics(r) = F_S(\ Semantics(r)\ )$$

F_S is the semantic version of the functional F used to define r. The semantics of r is a fixed point of the functional F_S. Furthermore, we already stated that a computer can not invent a meaning on its own. Therefore, Semantics(r) should be the *least* fixed point of F_S.

For pure denotational semantics, two questions remain unanswered. First, we could ask ourselves whether it is always possible to find an appropriate domain given a syntactic phrase. Second, we might wonder whether the least fixed point (i.e. the infinite union) used in the example always exists. We know that when working in a cpo, this limit will be defined, if the sequence $F_S{}^0(\perp)$, $F_S{}^1(\perp)$,... is a 'well defined sequence. What should these sequences be like? Does every functional F gives us these kind of sequences?

5. The Construction of Domains

5.1 Semantic domains for general purpose programming languages.

[6]Notice that use of integers to hide boolean expressions does not contribute to a well designed language, but design is not our main concern in this paper. The problem can easily by solved by adding a syntactic set of boolean expressions and writing semantic declarations for them.
[7]Notice that F_S is the semantic version of the functional $F = \lambda f \lambda n$. if n=0 then 1 else n*f(n-1) which is used to denote the factorial function at a *syntactic* level.

In this section, we will present an overview of the techniques that are available for constructing semantic domains. However, first, we need some formal definitions.

Definition:
A *partial order* (po) over a set D is a relation $R \subseteq D \times D$ such that
(1) reflexivity: for all x in D: $(x,x) \in R$.
(2) transitivity: for all $(x,y), (y,z) \in R$, $(x,z) \in R$.
(3) antisymmetry: $(x,y), (y,x) \in R$ implies x=y.

The sequence that we have been constructing in the factorial example will be called a *chain* or a *total order*.

Definition:
For a partially ordered set $(D, \sqsubseteq)$, a subset X of D is a *chain* iff X is non empty and for all $a,b \in X$, either $a \sqsubseteq b$ or $b \sqsubseteq a$.

An example of a partial order is $\mathcal{P}(A), \subseteq$ where $\mathcal{P}(A)$ is the powerset of some arbitrary set A. The rational numbers $\mathbb{Q}$ together with the $\leq$ relation define a total order.
In our factorial example, the relation $\sqsubseteq$ was taken to be $\subseteq$ and D = { $graph_i$ | $i \geq 0$ }.

The limit of a chain will be formalized by the least upper bound notion.

Definition:
Let D be a partially ordered set and let X be a subset of D. $d \in D$ is called an *upper bound* of X iff for all $x \in X$, $x \sqsubseteq d$. An upper bound d is called a l*east upper bound (lub)* if for every other upper bound d' of X, $d \sqsubseteq d'$. The least upper bound of a set X is denoted $\sqcup X$.

Definition:
A partially ordered set D is a *complete partial order* iff every chain in D has a least upper bound in D.
A partially ordered set D is a *pointed or lifted complete partial* order if it is a complete partial order and it has a least element, i.e. an element l such that for all $x \in D$, $l \sqsubseteq x$.

An example of a complete partial order is $\mathbb{R},\leq$ (whereas $\mathbb{Q},\leq$ is not complete), but it is not lifted. To change it into a lifted complete partial order, a notion of $\perp$ should be included. This can be done by introducing the element $-\infty$ or by restricting the set to $\mathbb{R}^+,\leq$ (i.e. all reals starting from zero).

Definition:
A *semantic domain* is a lifted complete partial order.

5.2 The semantics of recursive programs

At the end of the section 4 we asked ourselves which sequences will have limits. Now we know that the sequences must be chains. We also wondered if every functional F will result in the sequence $F^0(\perp)$, $F^1(\perp)$, $F^2(\perp)$,... being a good one. In other words, will every functional F yield a chain when applied successively to $\perp$? It turns out that this is not the case. In order to get chains, F must be a monotonic functional.

Definition:
Let A and B be cpo's (or just po's). A function $f: A \rightarrow B$ is *monotonic* iff $x \sqsubseteq_A y$ implies $f(x) \sqsubseteq_B f(y)$

Monoticity is a very natural concept for computer science. It is quite trivial that the more information is offered to a functional, the more information it can produce. In fact, it turns out that every computable functional is monotonic. A nice example of the converse of this statement is to show that the Halting Problem is not computable, by showing that it is not monotonic.

It can be shown that every monotonic functional F will result in a chain $F^0(\perp)$, $F^1(\perp)$, $F^2(\perp)$,... but this is not enough for the upper bound of this chain to satisfy the fixed point property. It can be shown (see [Tarski55]) that every monotonic function over a cpo has a cpo of fixed points. But what we need is the least among those fixed points, because we already concluded that a computer can not add meaning by itself. Moreover, what we require is a unique least fixed point because we do not want a recursive program to have more than one meaning.

Of every possible meaning, *the* one which contains the least amount of information is the semantics we are looking for. In order to accomplish this, we will need a generalization of monoticity towards infinite objects (i.e. the lub's). This generalized concept is called continuity.

Definition:
For cpos A and B, a monotonic function $f: A \to B$ is *continuous* iff for any chain $X \sqsubseteq A$,
$f(\sqcup X) = \sqcup \{ f(x) \mid x \in X \}$.

Thus, continuous functions preserve limits of chains. As explained before, continuous functions are exactly what we are looking for:

Theorem:
If the domain D is a lifted cpo, then the unique least fixed point of a continuous functional $F: D \to D$ exists and is defined to be $fix(F) = \sqcup \{ F^i(\bot) \mid i \geq 0 \}$.

As suggested by the factorial example, we state:

Definition:
The meaning of a recursive specification $f = F(f)$ is taken to be $fix(F_S)$, the least fixed point of the functional F_S which is the denotation for F.

5.3 Constructing semantic domains

When designing a denotational semantics, one will use a bunch of primitive domains and use domain operators to combine them into more complicated domains. For example, the semantics of a Pascal integer variable might be given by an ordinary integer number from $\mathbb{Z}$. Because every integer number contains an equal amount of information, the partial order induced by $\mathbb{Z}$ is empty! These are the kinds of domains we will use as our primitive domains. They are called *flat or discrete partial orders*. We will now discuss the operators domain-theory offers us to combine domains into more complex domains, the so called *compound domains*. First, we need an operator to convert a flat po into a semantic domain.

Definition:
For a partially ordered set A, its *lifting* $A_\bot$ is the set $A \cup \{\bot\}$, partially ordered by the relation $d \sqsubseteq_{A_\bot} d'$ iff $d = \bot$ or $d, d' \in A$ and $d \sqsubseteq_A d'$.

The elements of A are called proper elements while $\bot$ is an improper element. Some authors prefer to use the bottom element $\bot$ to denote error situations. This is a bad idea because an error situation is fundamentally different from an unrecoverable situation. Errors can be propagated 'upward' and can be given to the user as a result of a computation, while this is certainly not the case with nontermination. This is because the Halting Problem is undecidable. The best thing to do is to include a special element ? in our domains, which will stand for an error situation.

Definition:
For partially ordered sets A and B, their *product* $A \times B$ is the set $\{ (a,b) \mid a \in A \text{ and } b \in B\}$, partially ordered by the relation $(a,b) \sqsubseteq_{A \times B} (a',b')$ iff $a \sqsubseteq_A a'$ and $b \sqsubseteq_B b'$.

The usefulness of products is obvious. They can be used to serve as semantic domains for lists, records, arrays, pairs of functions, etc. A construction which is a bit more exotic is the disjoint union or sum of domains.

Definition:
For partially ordered sets A and B, their *sum* $A + B$ is the set $\{ (0,a) \mid a \in A\} \cup \{ (1,b) \mid b \in B\}$, partially ordered by the relation $d \sqsubseteq_{A+B} d'$ iff ($d = (0,a)$, $d' = (0,a')$, and $a \sqsubseteq_A a'$) or

($d = (1,b)$, $d' = (1, b')$ and $b \sqsubseteq_B b'$).

Disjoint unions are used to specify choices between semantic elements. For example, defining domain C as A + B (i.e. C = A + B) is used to express that a value of C can be either a value of A or one of B.

Until now, the domain constructors have been strongly biased towards modeling of data. But we also need a domain which contains denotations that are able to process something. In theoretical computer science, this is usually modeled by a function which transforms some value into another one. Such a domain of transformers is actually a function space. But according to the arguments given above, we are only interested in continuous functions. This will also turn out handy when solving recursive domain equations.

Definition:
For partially ordered sets A and B, their *function space* $A \to B$ is the set of all continuous functions with domain A and range B, partially ordered by the relation $f \sqsubseteq_{A \to B} g$ if for all $a \in A$, $f(a) \sqsubseteq_B g(a)$.

There are lots of theorems about complete partial orders and lifted complete partial orders. In fact, the study of these structures is an entire subfield of mathematics. The most important of them are summarized in the following properties. They resemble the closure properties of (lifted) complete partial orders.

Properties:
A flat domain P is never lifted.
If A and B are lifted, then $A \times B$ is also lifted.
$A + B$ is never lifted.
If B is lifted, so is $A \to B$.
$A_\bot$ is by definition always lifted.
If A and B are complete, then $A \times B$, $A + B$ as well as $A \to B$ are complete.

Another important theorem is that every operator over $A \times B$, $A + B$ and $A_\bot$ such as projection functions, selecting $\bot$ from $A_\bot$, ... is continuous. Furthermore, we know that $A \to B$ only contains continuous functions if it is defined with the order given in the above definition. These are all very important results because they ensure that every functional expression over lifted cpos that follows the given definitions (including functional expressions involving selectors of the compound domains) has a least fixed point (because they are continuous) and therefore can be used to give the semantics of a general purpose language.

The final problem that remains to be solved is the recursively defined domain equations. The theory resumed so far only allows us to write nonrecursive equations such as Value = $(\text{Boolean} + \text{Integer})_\bot$ but when one wants to give a semantics to things as lists or strings, the need for recursively defined domains arises quickly.

5.4 Recursive domain equations

Let us, for example, try to define a domain for expressing the meaning of ordinary lists. Let FL be the domain of finite lists. A finite list over a set A is either the empty list (which will be denoted by *nil*) or an element from A followed by another finite list. A list can also be undefined.
Hence,

$$FL = (\{nil\} + (A \times FL))_\bot$$

Does such an equation make any sense? It is possible to solve such equations (which we will call *recursive domain equations*) in an effective way? But is it really necessary to solve them? From a mathematicians point of view, this question should be answered positive, but for a computer scientist, the equation itself contains all the information he needs to know.

Caution is needed when interpreting such equations. For example, a semantic domain for the λ-calculus is given by the equation $D = D \to D$ because λ-terms such as $\lambda x.xx$ tell us that objects from λ-calculus can serve both as a function and as an argument. Hence, a domain for the λ-calculus should be one that contains its own function space. From a mathematical point of view, it should be clear that this is pure nonsense. No set can contain its own function space. Therefore, the equation sign should actually be an isomorphism sign. This makes sense because a function from $D \to D$ can always be represented by an element of D using its Gödel encoding, and vice versa[8].

In practice, the denotational descriptions are generally defined as solutions (up to an isomorphism) of domain equations involving standard primitive domains and the standard domain constructors $\perp$, $\times$, $+$ and $\to$. The solution of such equations is due to Scott and has become known as the inverse limit construction. The inverse limit method works like the limit construction used for giving semantics to a recursive function. We will build the domain by constructing a sequence of approximating domains. The elements in each approximation will be contained in the solution domain, and each approximation D_i will be a subdomain of D_{i+1}. If the equation was $D = F(D)$ where F is a functional expression build from using $\perp$, $+$, $\times$ and $\to$, we define $D_{i+1} := F(D_i)$ and $D_0 = \{\perp\}$. By some special mathematical constructions (retraction pairs), D_i is properly embedded in D_{i+j} such that $D_i \sqsubseteq D_{i+j}$, for all $j \in \mathbb{N}$. Then the solution D_∞ is defined $D_\infty = \cup\, D_i$ and it is easily shown that $D_\infty = F(D_\infty)$ and that D_∞ is the *least* domain satisfying the equation. We will not give further details about Scott's construction. The only important question remaining to be answered is a characterization of F that gives us precise conditions to guarantee a solution. It is shown in [Schmidt88] that any domain expression F which maps a lifted cpo E to a lifted cpo F(E) results in a solution D_∞ for the equation $D = F(D)$. Hence every *domain* equation can be solved. Many examples of this can be found in the literature (see [Schmidt88] and [Tennent91]). As a counter example consider the equation $D = B + (D \to D)$. This equation has no solution because a disjoint union is never lifted (see the rules above). The problem is removed if we change the equation to $D = (B + (D \to D))_\perp$.

6. Some Standard Techniques

Since the development of denotational semantics by Dana Scott and Christopher Strachey, many people used their ideas when trying to lay down semantic domains and valuation functions for their languages. From this, a series of 'standard' techniques has become available. In this section, we will discuss the most relevant among them. An excellent treatment of them can be found in [Gordon79].

6.1 The semantics of arithmetic expressions

In this section, a semantics will be given for an expression language in which ordinary integer expressions can be written. We assume that expressions consist of literals, variables and the ordinary operators +, -, * and div. The mechanism used to associate variables with a value, is given by a function mapping variable names to values. Such a function is called a state. Hence, we will be using three domains for denoting integer expressions. The first domain, is the flat domain $\mathbb{Z}$ of integer numbers. Another flat domain is 'Identifier' which will be used to denote the mapping that models the state. Finally a state is a function associating identifier names with integer numbers. In terms of the previous paragraphs, a state is therefore an element of the space of continous functions Identifier $\to$ Integer.

$$\begin{aligned}&\text{Integer} = \mathbb{Z}\\ &\text{State} = \text{Identifier} \to \text{Integer}\end{aligned}$$

The value of an integer expression containing variables depends on the values of those variables. Hence, a state (by which the value of each identifier is determined) will be given to our semantic evaluation function. The result of evaluating an expression will therefore be a continuous function associating states to values.

$$\mathbb{E} : \text{Expression} \to \text{State} \to \text{Integer}$$[9]

As is often done in denotational semantics, this is the *curried* version of $\mathbb{E}$ (after Mr. Curry). The definition of $\mathbb{E}$ can also be written as $\mathbb{E}$: (Expression $\times$ State) $\to$ Integer, which is its uncurried version. Let us now discuss this function in detail.

The semantics of syntactic numerals are their corresponding values from $\mathbb{Z}$:

$$\mathbb{E}\llbracket \text{ zero } \rrbracket = \lambda s \,.\, 0$$

[8]How is this possible? Isn't it always the case that a function space is much bigger that an ordinary set? The trick that enables us to say that the equation $D = D \to D$ has a solution, is the fact that $D \to D$ only contains continuous functions. There are much less continuous functions than functions.

[9]Notice that $\to$ associates to the right. By $A \to B \to C$, we mean $A \to (B \to C)$ and not $(A \to B) \to C$.

$$\mathbb{E}\llbracket \text{ one } \rrbracket = \lambda s \,.\, 1$$

…

The semantics of a variable consists of looking it up in the provided state s and returning its value:

$$\mathbb{E}\llbracket\, i \,\rrbracket = \lambda s \,.\, s(i) \text{ where } i \in \text{Identifier}$$

The semantics of other numerical expression is defined according to the syntactic definition:

$$\begin{aligned}&\mathbb{E}\llbracket E_1 + E_2 \rrbracket = \lambda s \,.\, \mathbb{E}\llbracket E_1 \rrbracket s +_{\mathbb{Z}} \mathbb{E}\llbracket E_2 \rrbracket s\\ &\mathbb{E}\llbracket E_1 * E_2 \rrbracket = \lambda s \,.\, \mathbb{E}\llbracket E_1 \rrbracket s *_{\mathbb{Z}} \mathbb{E}\llbracket E_2 \rrbracket s\\ &\mathbb{E}\llbracket E_1 - E_2 \rrbracket = \lambda s \,.\, \mathbb{E}\llbracket E_1 \rrbracket s -_{\mathbb{Z}} \mathbb{E}\llbracket E_2 \rrbracket s\\ &\mathbb{E}\llbracket E_1 \text{ div } E_2 \rrbracket = \lambda s \,.\, \mathbb{E}\llbracket E_1 \rrbracket s \text{ div}_{\mathbb{Z}}\, \mathbb{E}\llbracket E_2 \rrbracket s\end{aligned}$$

where $+_{\mathbb{Z}}$, $*_{\mathbb{Z}}$, $-_{\mathbb{Z}}$ and $\text{div}_{\mathbb{Z}}$ are the usual operators from the semantic algebra $\mathbb{Z}$. Notice that the last clause (for div) is not quite right. Indeed, what happens if E_2 evaluates to zero? This can easily be solved by including the value ? in the 'Integer' domain, and returning ? every time an error has occured.

It is not difficult to see that the same techniques can be used for determining the semantics of other kinds of expressions, as long as the expressions cause no side-effects. The essence of these techniques is that, for an expression-like language, one needs to structure the semantic function recursively based on the formal grammar of the language (in order to get compositionality). The recursion is stopped at the level of the terminal symbols.

6.2 The semantics of common commands

By common commands, we mean commands that do not change control flow 'in an unnatural manner'. This will become clear in the following sections. As usual, a computation process consists of transforming states through time. Thus, ordinary commands in procedural languages take a state and transform it into another state.

$$\mathbb{C}\colon \text{Command} \to \text{State} \to \text{State}$$

A few examples of commands are an assignment, a conditional expression and a while loop.

The semantics of an assignment reflects the adjustment of the provided state s, with the evaluation of the right hand side of the command. Hence,

$$\mathbb{C}\llbracket\, i{:=}E \,\rrbracket = \lambda s \,.\, s \Big[\ \mathbb{E}\llbracket\, E \,\rrbracket s \ / \ i \Big]$$

where s[v / i] is the same state as s, except that it maps the identifier i to the value v.
The meaning of a conditional is

$$\mathbb{C}\llbracket \text{ if } E \text{ then } C_1 \text{ else } C_2 \rrbracket = \lambda s \,.\, \text{Cond}_{\text{State}}(\ \mathbb{E}\llbracket E \rrbracket s\ ,\ \mathbb{C}\llbracket C_1 \rrbracket s\ ,\ \mathbb{C}\llbracket C_2 \rrbracket s\ )$$

As explained in section 4, no trouble would have emerged from the use of ordinary sets, for the constructs treated so far. Our next construction is a while loop. We already stated several times that treating variable iteration or recursion forces us to use domains instead of sets. The semantics of a while loop is given as follows:

$$\mathbb{C}\llbracket \text{ while } E \text{ do } C \rrbracket = \lambda s \,.\, \text{Cond}_{\text{State}}(\ \mathbb{E}\llbracket E \rrbracket s\ ,\ \mathbb{C}\llbracket \text{ while } E \text{ do } C \rrbracket\ (\mathbb{C}\llbracket C \rrbracket\ s)\ ,\ s\ )$$

Clearly, $\mathbb{C}\llbracket$ while E do C $\rrbracket$ is defined in function of itself. But we saw that this is solved by taking the fixed point of its defining functional, i.e.

$$\begin{aligned}&\mathbb{C}\llbracket \text{ while } E \text{ do } C \rrbracket = \lambda s \,.\, \text{Cond}_{\text{State}}(\ \mathbb{E}\llbracket E \rrbracket s\ ,\ \mathbb{C}\llbracket \text{ while } E \text{ do } C \rrbracket\ (\mathbb{C}\llbracket C \rrbracket\ s)\ ,\ s\ )\\ &\Leftrightarrow\\ &\mathbb{C}\llbracket \text{ while } E \text{ do } C \rrbracket = (\lambda d \lambda s.\text{Cond}_{\text{State}}(\ \mathbb{E}\llbracket E \rrbracket s\ ,\ d\ (\mathbb{C}\llbracket C \rrbracket\ s)\ ,\ s\ )\ )\ \mathbb{C}\llbracket \text{ while } E \text{ do } C \rrbracket\\ &\Leftrightarrow\\ &\mathbb{C}\llbracket \text{ while } E \text{ do } C \rrbracket = \text{fix}\ (\ F\ ) \text{ where } F = \lambda d \lambda s.\text{Cond}_{\text{State}}(\ \mathbb{E}\llbracket E \rrbracket s\ ,\ d\ (\mathbb{C}\llbracket C \rrbracket\ s)\ ,\ s\ )\end{aligned}$$

What remains, after applying fix to F, is a function transforming states to states, which is exactly what we wanted according to the definition of $\mathbb{C}$.

As these examples show, there is nothing mysterious or spooky about denotational semantics. Most natural programming language constructs are modeled by it in a very trivial way. However, it will soon become clear that a lot of research still needs to be done because the semantics of higher level programming constructs such as procedures, types and block structuring is not so easy to comprehend.

6.3 The semantics of blockstructured languages

The use of blockstructured languages requires the notion of a ‘scope’. When can we say whether a variable is visible or not, and which variable is actually meant when two of them with the same name have been declared. The main problem with blocks is that at some point, they introduce the need for sharing of variables, i.e. two different identifiers that share the same content. This is very much the case when we talk about call-by-reference and when we introduce pointer structures.

To solve this, a pointer mechanism is introduced. Instead of immediately associating the identifiers to a value (with a state), we will map identifiers to locations and locations to values. This makes it possible for two different identifiers to be mapped onto the same location (and thus also to the same value). Once the first identifier is updated by an assignment command, the second will also be adjusted. To model the association of locations with values, we introduce the concept of a *store*. The domain of stores is defined by:

$$\text{Store} = \text{Location} \to (\text{Value} + \{\text{unused}\})$$

where Value is the set of denotable values (Integer in our case) and where Location is an arbitrary flat domain.

The association of identifiers to stores is done using an environment. An environment maps a set of identifiers to their corresponding location, and hence serves as a way to look at the identifiers. Stated somewhat more technically, they define a scope.

$$\text{Environment} = \text{Identifier} \to (\text{Location} + \{\text{unbound}\})$$

A request for the value of an identifier is then solved in two stages. First, the identifier is looked up in the provided environment, which will return a location. Next, the contents of this location should be determined by looking up the location in the current store. Therefore, the semantic evaluation for expressions requires two parameters e and s, representing the current environment and store. In particular, the semantics of an identifier is given by[10]:

$$\mathbb{E}\,\llbracket\, i \,\rrbracket = \lambda\, e\, s.\ \text{Cond}_{\text{Value}}\big(\, e(i) = \text{unbound} \,,\, \text{error} \,,\, s(e(i))\,\big)$$

Hence, assignment commands will only change the contents of a location, but not the location of an identifier itself. The only way in which an identifier’s location can be modified is by changing the environment and this is exactly what is done to express the semantics of block-entering or procedure calling. An extensive treatment of these topics can be found in [Gordon79].

6.4 The semantics of control commands

Using the techniques of the previous section, it is very difficult to denote statements in which the control flow of the program is explicitly changed, such as procedure-calls, goto’s, exception handling, etc.

To avoid these problems, Scott proposed a set of techniques which has become known as *continuation semantics*, in contrast to *direct semantics*, which is a term used to denote the techniques presented above. In continuation semantics, an explicit description of control is used. Such a description is called a *continuation*. A continuation can be seen as a description of the future computation. It models ‘the rest of the program’ or ‘what remains to be done’. The idea is that each construct decides for itself where to pass its result. Usually, this will simply be the command following the construct, but in some cases such as a ‘goto’ this scheme can be altered, and the result is passed to another continuation. Other examples where continuations are useful is error handling, exception handling, escapes, etc.

10 From now on, we will assume that $\mathbb{E}$: Expression $\to$ Environment $\to$ Store $\to$ (Value + {error})

We argued that a computational process is simply a transformation of memory through time. Hence, ‘the rest of the program’ should also be such a transformation. In other words, a continuation is nothing more but a continuous function that converts a store into another store, i.e.

$$\text{Continuation} = \text{Store} \to \text{Store}$$

Every command should be evaluated within a provided environment. Furthermore, a continuation must be given which tells what to do after the command has been executed. As explained above, the command will transform a store into another store through time. Taken all together, the semantics of a command specifies how a provided environment, a continuation and a store are transformed into another store, i.e.

$$\mathbb{C} : \text{Command} \to \text{Environment} \to \text{Continuation} \to \text{Store} \to \text{Store}$$

But Store $\to$ Store = Continuation, such that

$$\mathbb{C} : \text{Command} \to \text{Environment} \to \text{Continuation} \to \text{Continuation}$$

Hence, every command gets a continuation and evaluates to another continuation. In order to explain the denotation of a command, we proceed as follows.
According to the above definition, the semantics of a command is a continuous function from

$$\text{Environment} \to \text{Continuation} \to \text{Continuation}$$

Provided with an environment e and a continuation c, we get $\mathbb{C}\llbracket\, C \,\rrbracket\, e\ c$ which is a function from

$$\text{Store} \to \text{Store}$$

The meaning of this function $\mathbb{C}\llbracket\, C \,\rrbracket\, e\ c : \text{Store} \to \text{Store}$ is the following. The command C is executed within environment e in store s. The result of this execution is s’ which is then passed to c, i.e.

$$\mathbb{C}\,\llbracket\, C \,\rrbracket\, e\ c\ s = c\ s'$$

For expressions, the situation is analogous. An expression receives an expression-continuation and transforms it to a normal continuation.

$$\mathbb{E} : \text{Expression} \to \text{Environment} \to \text{EContinuation} \to \text{Continuation}$$

where

$$\text{Econtinuation} = \text{Value} \to \text{Continuation} = \text{Value} \to \text{Store} \to \text{Store}$$

Once the value v of an expression is known (in a certain environment e), it will transform a store s into another store s’, which is then passed to the expression continuation k which resembles the rest of the expression evaluation process, i.e.

$$\mathbb{E}\,\llbracket\, E \,\rrbracket\, e\ k\ s = k\ v\ s'$$

As a concrete example of continuation semantics, we redefine the denotation of a while construct.

$$\mathbb{C}\llbracket \text{while } E \text{ do } C \,\rrbracket = \lambda e\ c\ s.\ \mathbb{E}\llbracket E \,\rrbracket\, e\ (\,\lambda v.\ \lambda\ s'.\text{Cond}_{\text{State}}(v,\ \mathbb{C}\llbracket C \,\rrbracket\, e\ (\,\mathbb{C}\llbracket \text{while } E \text{ do } C \,\rrbracket\, e\ c\,)\ s',\ s'))\ s$$

These seem to be unnatural and very complicated constructs. However, continuation semantics also has some advantages over direct semantics. One of them is the ease with which a denotation for unconditional branches can be given.

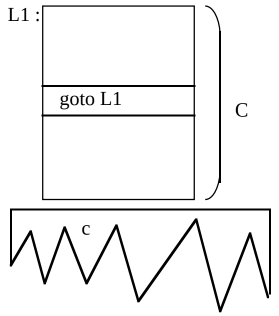
$$\mathbb{C}\llbracket\ \text{goto } L\ \rrbracket = \lambda\ e\ c\ .\ \mathbb{E}\llbracket\ L\ \rrbracket\ e\ k \text{ where } k : \text{Value} \to \text{Continuation}\text{: } \lambda c'.c'$$

This equation is interpreted as follows. First, the label L is evaluated in the current environment e. The value of L will be some continuation c, which represents ‘the rest of the program’ after the label declaration L. This continuation will be offered to λc’.c’. Hence, the overall result will be the continuation bound to L. Notice that this requires continuations to be storable values. This should be no surprise because, in order to model jumps, it is essential that labels can be bound to program positions which actually resemble the rest of the program.

The only problem that remains to be solved is extending the environment with the label L bound to its corresponding continuation.

```
begin
        …
        L_1 : C_1;
        …
        L_k : C_k;
        …
        goto L_1;
        …
end;
```

We will assume that a label is declared by a statement L:C where L is a label and C is a (possibly compound) command. The scope of L is defined to be the smallest surrounding block of C which means that it is not possible to jump out of blocks in which a label was declared. When entering a block, aside from extending the environment with any new declarations, the environment also has to be extended with every label that is declared inside the block. This is much more difficult as it may seem. Consider the following piece of code.



The semantics of this code is $\mathbb{C}[\![L_1 : C]\!]$ e c where c represents the computation after block C has been executed. Suppose that we would extend our environment at this point by simply binding the label L_1 to $\mathbb{C}[\![C]\!]$ e c. Hence, $\mathbb{C}[\![L_1 : C]\!] e\ c = \mathbb{C}[\![C]\!] e_{extended}\ c$ where $e_{extended}$ is the environment e upgraded with a binding of L_1 to the continuation representing C.
But this is exactly $\mathbb{C}[\![C]\!] e_{extended}\ c$ because C has to be evaluated in its new environment in order to look up L_1 when the denotation of the goto statement has to be calculated. Again, this circularity can be removed by solving a fixed point equation over the definition of the new environment[11]. This equation is given by

$$c' = \mathbb{C}[\![L{:}C]\!]\ e\ c = \mathbb{C}[\![C]\!]\ e[\ c' / L\ ]\ c$$

Thus,

$$\mathbb{C}[\![L{:}C]\!]\ e\ c = \text{fix}(\ \lambda\underline{c}.\ \mathbb{C}[\![C]\!]\ e[\ \underline{c} / L\ ]\ c\ )$$

Now consider the situation in which the goto statement is written before L has been declared. Then, the approach presented here will not work because we are binding labels at the time when they are met. We will therefore install an evaluation function $\mathbb{J}$ which will collect every binding of a block before the block is entered. This function is recursively defined for every command. Hence, $\mathbb{J}[\![C]\!]$ takes the original environment together with the rest of the program, i.e. the statements following the block. The result of $\mathbb{J}$ is a new environment extended with every label associated to its corresponding continuation.

$$\mathbb{J}\text{: Command} \rightarrow \text{Environment} \rightarrow \text{Continuation} \rightarrow \text{Environment}$$

[11]This is only normal, because otherwise, we would be able to construct a variable loop without fixed point semantics.

Of course, the definition of $\mathbb{J}$ has to be combined with the fixed point method presented above.

When $\mathbb{J}$ is applied to ‘normal’ commands, nothing noteworthy happens. $\mathbb{J}$ is only correctly distributed over the command’s constituents. For example, in a command $C_1;C_2$, we assume that the labels occurring in C_2 override those occurring in C_1. This is just a matter of convention to solve name clashes. The only important situation in which $\mathbb{J}$ is worth considering, is the actual declaration of a label, i.e. in a command of the form L : C. In such a situation, the continuation of C should be bound to L. This is done by structuring $\mathbb{J}$ as follows:

$$\mathbb{J}[\![L : C]\!]\ e\ c = \Big(\mathbb{J}[\![C]\!]\ e\ c \Big) \Big[\ (\mathbb{C}[\![C]\!]\ e\ c\ ) / L\ \Big]$$

Hence, L is bound to the continuation corresponding to doing C followed by the rest of the program c. The function is explained in great detail in [Gordon79].

To combine this technique with the fixed point technique presented above, the part of $\mathbb{C}$ used to denote label declarations, has to be redefined.

$$\mathbb{C}[\![L : C]\!] = \mathbb{C}[\![C]\!]$$

This is because the label L was already integrated (by $\mathbb{J}$ of course) in the environment of the block surrounding the command L : C.

Last, the denotation of entire blocks is given as follows.

$$\mathbb{C}[\![\text{begin C end}]\!]\ e\ c = \mathbb{C}[\![C]\!]\ e[e']\ c \text{ where } e' = \mathbb{J}[\![C]\!]\ e[e']\ c$$ [12]

7. Relation to Other Approaches

In section 2 it was explained that none of the three approaches should be seen as a competitor of the others. The semantic trinity can be seen as a tool for language development. Given the task of developing a new language, it should first be clarified how the concepts of the language are related to each other and to the user of the language. An axiomatic description of the language is highly suited for this task. Then a denotational description of the language should be given. With this, the meaning of the language becomes fixed and a formal proof can be constructed to establish the desired properties of the constructs (i.e. to show that the denotational description can be seen as a model for the axiomatic description). Finally, the denotational semantics should be implemented using an operational definition. Together, these orthogonal language definitions support a systematic design, development and implementation of the language.
To make this scheme possible in practice, some formal relations between the paradigms must be investigated. These can be found in [Tennent91]. One of the greatest break throughs in this field was [Goguen77]. For concurrent programs, we refer to [Hennessy88].

8. Acknowledgments

This paper would have never come into existence without the numerous discussions we had with Patrick Steyaert. We also wish to thank Kim and Tom Mens for our endless/useful discussions about formal systems and everyone who has spent time on reading the earlier draft versions.

9. References

[deBakker80]

Bakker, de, Jaco ; Mathematical theory of program correctness; Prentice-Hall (London), 1980

[12]The notation e[e’] is used to denote the environment e extended (or overriden) with environment e’. For formalizations of these issues, see [Gordon79].

[Goguen77]

Goguen, J.A., Thatcher, J.W., Wagner, E.G and Wright, J.B. ; Initial algebra Semantics and Continuous Algebras; JACM Vol. 24 No. 1 January 1977, pp 68-95.

[Gordon79]

Gordon, Micheal J.C. ; The denotational description of programming languages: An introduction; Springer Verlag (New-York, Heidelberg, Berlin), 1979.

[Hennessy88]

Hennessy, Mathew ; Algebraic Theory of Processes;.MIT-Press (Cambridge), 1988.

[Hopcroft79]

Hopcroft, John E. and Ullman, Jeffrey D. ; Introduction to automata theory, languages and computation; Addison-Wesley (Amsterdam), 1979.

[Plotkin]

Plotkin, G. D. ; A structural approach to operational semantics; Technical Report, University of Aarchus, Denmark.

[Rayward-Smith84]

Rayward-Smith, V.J. ; A first course in formal language theory; Blackwell (Oxford), 1984.

[Schmidt88]

Schmidt, David A. ; Denotational Semantics: A methodology for Language Development; Wm. C. Brown Publishers (Dubuque, Iowa), 1988.

[Tarski55]

Tarski, A. ; A Lattice-Theoretical Fixpoint Theorem and its Applications; Pacific Journal of Mathematics 5, pp. 285-309, 1955.

[Tennent91]

Tennent, R.D. ; Semantics of Programming Languages; Prentice Hall (NY, London, Toronto, Sydney, Tokyo, Singapore), 1991.