An Application of Algebra to the Semantics of Programming Languages

by

C.A.Baker-Finch, B.Sc. Dip.Ed.

submitted in fulfilment
of the requirements for the degree of

Doctor of Philosophy

UNIVERSITY OF TASMANIA
HOBART

August 1985

Except as stated herein, this thesis contains no material which has been accepted for the award of any other degree or diploma in any university. To the best of my knowledge and belief, this thesis contains no copy or paraphrase of material previously published or written by another person, except where due reference has been made in the text of the thesis.

[illegible]

C A Baker-Finch

Acknowledgements

I would like to express my gratitude to my supervisors Charles Lakos and Richard Watkins for their valuable time and helpful advice and especially Phil Collier for his many suggestions and healthy scepticism. Thanks also to Toni Hickey for careful and accurate typing of a difficult manuscript. Last but certainly not least I thank my wife Sue for her patience and constant support.

Abstract.

This dissertation investigates the use of the algebraic style of abstract data type specifications for the definition of programming language semantics. The choice of appropriate mathematical models for such presentations is an important aspect of this work largely because the semantics of programming languages will generally be defined in terms of domains that are more complex than those required for dealing with more elementary data types. The relationship between initial algebra semantics and the proposed style of specification is explored.

From this foundation, the intuitive notion of the congruence of a pair of semantic definitions can be inspected and formalised against an algebraic background. Using the formal definition so developed and the simple but powerful notion of initiality, proofs of congruence are possible for semantics that are not amenable to the more traditional techniques of structural and fixed-point induction.

Finally the problem of establishing the correctness of a compiler is investigated, reworking the traditional "commuting square" approach for the style of semantic presentation developed in this thesis rather than the usual initial algebra style. This allows a clearer focus on some of the shortcomings of the commuting square notion.

Table of Contents

ctd.

Chapter 1
Introduction.

The connection between universal algebra and the specification of abstract data types has become well established since the seminal works of Guttag (1975) and Liskov & Zilles (1975). While these early works were rather informal with respect to the precise mathematical object being associated with each particular specification (i.e. the "semantics" of the specification language), much has been published since then (Kutzer & Lichtenberger, 1983) especially relating to the modularisation of the specifications. Naturally these two facets are intermingled in the literature since each new modularisation tool that is put forward requires separate treatment of its formal semantics. Despite the volume of literature that has appeared in the past decade there is still no real consensus on the basic issue of what mathematical object (algebraic theory, signed theory constrained theory, equational variety of algebras, initial algebra etc) is the most appropriate choice as the semantics of an abstract data type specification.

Burstall & Landin (1969) suggested a connection between universal algebra and programming, particularly programming language semantics and compiler correctness, but progress in this area of research seems to have been much slower. It would appear that the use of equations by Wand (1980b) to specify programming language semantics was the first indication that the work on abstract data type specifications could be extended to apply to programming language semantics. However, the better understanding of abstract data type specifications has not really impacted upon this area and informal and simplistic semantics dominate in the literature.

In Chapter 3 we investigate the use of algebraic presentations for specifying programming language semantics. We adopt a particular mathematical object as being the most appropriate one to associate with such presentations, thus defining a semantics of the specification language. The adequacy of such a choice is then examined and its relationship to initial algebra semantics is detailed.

One of the main purposes of precise and well-founded specifications is to make it possible to state and prove formal properties in a rigorous way. Hence, in Chapter 4 we look at the rather intuitive notion of *congruence* between semantic models and produce a clear and thorough algebraic formulation of the concept. The machinery of the algebraic foundations makes possible some proofs of congruence that are not amenable to the traditional inductive approach used by workers in denotational semantics.

The techniques and concepts developed in Chapter 4 are immediately applicable to the classic problem of establishing the correctness of a compiler or translation algorithm. Chapter 5 looks at a variation on Morris's approach that is suited to our style of semantic definition and allows us to focus more clearly on some of the shortcomings of the (perhaps too simplistic) commuting square notion.

To put this dissertation in a proper perspective it is appropriate to briefly review the major related applications of algebra to the specification of programming language semantics that have appeared in the literature.

Goguen, Thatcher & Wagner (1977) exploit the implicitly algebraic structure of denotational semantics in their work on initial algebra semantics. By relating context-free grammars to signatures, abstract syntax becomes identified with an initial algebra. Hence every other algebra with the same signature provides a "semantics" for that language via the unique homomorphism property. We will look more closely at this approach, particularly in §3.2.

Wand (1980b) suggests the possibility of semantic definitions consisting of a signature and a set of equations. As such, his work is closely related to our approach and provided considerable inspiration, but was in terms of single-sorted algebras and did not address the problem of restricting the class of models of a presentation that provide an acceptable semantics for the particular language. Further, while Wand identifies denotational semantics with the initial model, his operational semantics is reducible to the same object, this view being supported by Goguen (1980) and Goguen & Parsaye-Ghomi (1981). Mosses (1983) raises the criticism that Wand's semantic functions are "... just operations of abstract data types that combine syntax and semantics", yet he is willing to be much more flexible in his consideration of denotational semantics valuations. Our work in Chapter 3 suggests this criticism is not justified.

Goguen & Parsaye-Ghomi (1981) built on Wand's work by treating a larger language, using many-sorted algebras and by modularising the definition, though we feel that the choice of modules does little to assist the reader. Further, since they insist on the initial model their semantics is *not* denotational, as the semantics of procedures are given by closures consisting partly of unevaluated program text.

The work of Gaudel (1978, 1980) and Pair (1982) is based on the style of abstract data type specifications. They appear to be specifying a particular algebra (based on their definition of states or "informations") though their work is rather informal. They make considerable use of pre- and post-conditions and "modifiable operators" (though they can be considered only as notational sugar and are easily factored out) so that their definitions are clearly oriented toward compiler generation (Bjorner, 1983). We have misgivings about their use of notation without full consideration of its semantics and hence their rather arbitrary technique of establishing the correctness of implementations.

The "abstract semantics algebras" of Mosses (1980, 1981, 1983) are based on a combined use of initial algebra semantics and abstract data type specifications. One of the major aims would seem to be to further the author's intention to make semantic definitions more modular, so that extensions to the language being treated do not necessarily require a major rewrite of the existing clauses of the definition. Some confusion arises in (Moses, 1983) as to the meaning of an abstract semantic algebra specification, at first claiming:

> "There are, in general, many different possible models of the axioms of an ASA specification, including the (discrete) initial one, which is taken as its meaning."

However, the following contradictory statement is made later:

> "One may obtain a standard denotational description from one based on ASAs by choosing a model (using Scott domains) for the axioms of the ASA specifications."

Finally, he expresses some misgivings that his use of discrete algebras only provides for finite unfoldings (eg. of loops), yet this is only the case if attention is restricted to the initial algebra since otherwise we may simply choose a model constructed from appropriate Scott domains.

Broy & Wirsing (1980), Broy, Dosch, Moller & Wirsing (1981) and Wirsing & Broy (1982) take a rather different approach using partial algebras, with "definedness" predicates explicitly included in the specifications and maintained under an appropriate version of the notion of "homomorphism". This rather neatly allows the class of *all* models of a presentation to be considered. Though we have not considered the problem in detail, there would appear to be some difficulties related to the sufficiency and generality of an explicit "definedness" specification.

A somewhat different connection between algebra and programming language semantics is the notion of order-algebraic semantics (eg. Elgot , 1973, Wagner, Thatcher & Wright, 1978, Guessarian, 1981, 1983) where the central concepts are that of a rational algebraic theory or an ordered algebra. Work in this area is not directly relevant to the application of algebra propounded in this thesis, since it is concerned less with specifying semantics than with modelling the fundamental machinery of computation.

Chapter 2
Foundations.

Our aim in this chapter is to briefly survey the mathematical concepts and results that form the background against which the work of this dissertation is set. As such, we tread what we see as a middle ground between two extremes.

Firstly, we do not consider it appropriate here to give a full introductory coverage of the field and hence the discussion and examples given will most likely be insufficient for the complete algebraic novice. The following references are among those that combine to provide an accessible and computer science oriented introduction to those aspects of universal algebra and category theory that are relevant to this thesis: Goguen, Thatcher & Wagner (1978), Burstall & Goguen (1979, 1982), Goguen & Burstall (1984), Goguen & Meseguer (1983), Cohn (1981).

On the other hand we consider it equally inappropriate to aim this chapter only at the mathematical sophisticate. Thus in many instances where there is a choice among various expressions of the same result, or formulations of the same concept, we invariably choose the most prosaic or intuitively pleasing one, frequently at the cost of elegance. Also in this vein, we consider it inappropriate to include any proofs in a survey such as this. Perhaps the best justification for our approach is that we assume our audience to be computer scientists, and as such we feel that the best approach is the one that gives the best intuitions and insights without compromising the accuracy or rigour of the presentation.

The references given in this chapter are not in general to papers containing the original results but are chosen according to the criteria that they be reasonably recent and fairly standard; that their notation is similar to ours, where possible; and that they are preferably oriented toward computer science, at least with respect to the examples used.

Generally, surveys like this chapter are rather dry and lack relevance in isolation. We therefore feel that the material herein should be given only cursory attention to get the flavour of the mathematics used in the body of this dissertation. The reader may then refer back to this chapter when necessary.

2.1 Signatures and Algebras

One of the central concepts used in this work is that of a *many-sorted algebra* (Goguen, Thatcher & Wagner, 1978) which is basically a reformulation of the earlier notion of a *heterogeneous algebra* (Birkhoff & Lipson, 1970) which is in turn a generalization of an *algebra* (Birkhoff, 1935).

Definition. A *heterogeneous algebra A* consists of

1. a family $\{S_i\}$ indexed by some set I where each S_i is a non-void set called a *phylum* of *A*;
2. a set $\{f_\alpha\}$ of finitary total functions indexed by some set Ω where each f_α is a mapping
   $f_\alpha : S_{i(1,\alpha)} \times S_{i(2,\alpha)} \times \dots \times S_{i(n(\alpha),\alpha)} \rightarrow S_{r(\alpha)}$
   for some non-negative integer $n(\alpha)$, function $i_\alpha : k \rightarrow i(k,\alpha)$ from $\{1,2,\dots,n(\alpha)\}$ to I, and $r(\alpha) \in I$.

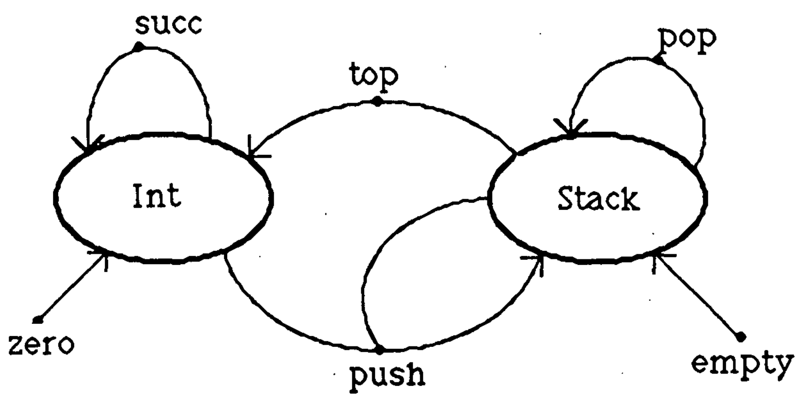
Simply put, a heterogeneous algebra is a family of non-void sets together with some functions among those sets. As such, it is a generalisation of the earlier notion of a (homogeneous) algebra that consists of a single set S together with some functions on that set.

It is convenient when dealing with algebras together with *algebraic theories* (another central concept in this dissertation) to give a different formulation of the same structure in terms of a *signature* or *operator domain* so that we may precisely characterise "species" of algebras. Again, a *many-sorted algebra* is essentially a family of sets (called the *carriers* of the algebra) with a collection of *operations* (total functions) among them. The index set for the carriers is called the *sort* set. (Note that the "non-void" restriction has been dropped from the original definition.) Since we deal exclusively with many-sorted algebras (rather than single-sorted) in this thesis, we will feel free to shorten the name to *algebra* without confusion.

Definition. Given a set S of *sorts*, an S-sorted *signature* or *operator domain* Ω is a family $\Omega_{w,s}$ of sets, for $s \in S$ and $w \in S^*$ (where S^* is the set of all finite strings from S, including the empty string λ). $F \in \Omega_{w,s}$ is an *operation symbol* of *rank* w,s, of *arity* w and of *sort* s.

Definition. Given an S-sorted signature Ω, an ***Ω-algebra A*** consists of a set A_s for each $s \in S$ and a function $f_A : A_{s1} \times A_{s2} \times \ldots \times A_{sn} \to A_s$ for each $f \in \Omega_{w,s}$ with $w=s1s2\ldots sn$. For $f \in \Omega_{\lambda,s}$, $f_A \in A_s$.

Thus, the purpose of signatures or operator domains is to identify the ***shape*** of many-sorted algebras, essentially providing names for the carriers and operators. Following this notion of shape a little further, signatures may be specified diagramatically with considerable clarity. Consider the following representation for a Stack-of-Integers signature.



The names in the ovals are the sorts, while the names on the edges connecting them are the operator symbols with arities given by the source of those edges and sorts by the targets. While this representation may be quite clear, we will generally employ the more widely used and compact notation given below for the same signature.

sort Int
- zero : $\to$ Int
- succ : Int $\to$ Int

sort Stack
- empty : $\to$ Stack
- push : Stack $\times$ Int $\to$ Stack
- pop : Stack $\to$ Stack
- top : Stack $\to$ Int

Calling this signature Σ, a possible Σ-algebra $\mathcal{S}$ would have carriers $\mathcal{S}_{Int} = \mathbf{N}$ and $\mathcal{S}_{Stack} = \mathbf{N}^*$. The operators could be chosen as follows:

$\text{zero}_{\mathcal{S}} = 0$
$\text{succ}_{\mathcal{S}} =$ the successor function

empty$_{\mathcal{S}}$= $\langle\rangle$, the empty list
push$_{\mathcal{S}}$= concatenate
pop$_{\mathcal{S}}$= tail of the list
top$_{\mathcal{S}}$= head of the list

Another useful generalization from the single-sorted case is that of the concept of homomorphism of many-sorted algebras.

Definition. If A and B are both Ω-algebras, an *Ω-homomorphism* $h: A \rightarrow B$ is a family of functions $\{h_s : A_s \rightarrow B_s \mid s \in S\}$ that satisfy
1. if $f \in \Omega_{\lambda,s}$ then $h_s(f_A) = f_B$
2. if $f \in \Omega_{s1s2...sn,s}$ and $\langle a_1,a_2,...,a_n\rangle \in A_{s1} \times A_{s2} \times ... \times A_{sn}$, then $h_s(f_A(a_1,a_2,...,a_n))=f_B(h_{s1}(a_1),h_{s2}(a_2),...,h_{sn}(a_n))$

Thus, in the same sense that group homomorphisms "preserve" the group operations, Ω-homomorphisms "preserve" the operations named in Ω.

It is convenient for us to couch some of our discussion in terms of the language of category theory, though it is not in fact necessary to do so. We use only the most basic notions of category, initial and final object. However it would be quite a straightforward matter to adjust our standpoint such that categories were the most central mathematical structure, rather than algebras. Informally, a *category* consists of a collection of *objects* together with some arrows (*morphisms*) between them such that identities are included and end-to-end composition is associative. For more details in a computer science vein see Goguen, Thatcher, Wagner & Wright (1973, 1975,1976). Arbib & Manes (1975) provide a very accessible introduction, while MacLane (1972) is the standard reference.

Result. A class of Ω-algebra together with *all* the Ω-homomorphisms between the algebras form a *category of Ω-algebras*, say **C**. The *objects* of **C** are the algebras and the *morphisms* of **C** are the Ω-homomorphisms.

This is the only type of category with which we will be having many direct dealings.

Definition. An object is *initial* in a category if and only if there is a

unique morphism from that object to every object in the category. The *dual* notion is the following:
An object is *final* (terminal) in a category if and only if there is a unique morphism to that object from every object in the category.

Clearly, in terms of categories of Ω-algebras this concept translates into: A is *initial* in a category **C** of Ω-algebras if and only if for every algebra B in **C** there is a *unique* homomorphism $h: A \rightarrow B$.

This notion is widely used both in the study of abstract data types (Goguen et al, 1978 and Zilles, 1979 among many others) and in some work on the semantics of programming languages (Goguen, Thatcher, Wagner & Wright, 1977, Mosses, 1983), including the present endeavour. A very straightforward but important result is the following:

Result. Given an algebra A, initial in a category **C** of Ω-algebras, an algebra B (in **C**) is initial in **C** if and only if B is isomorphic to A.

At this point it should be noted that it is the standard practice in algebra not to distinguish between isomorphic objects. Thus we will generally speak of *the* initial algebra rather than the isomorphism class of initial algebras.

Given a signature Ω, we denote the category of *all* Ω-algebras, Alg_Ω. Alg_Ω always has an initial algebra and the following construction provides us with a technique of directly deriving such an algebra from the signature Ω. The algebra is called an *Ω-word algebra* and is denoted T_Ω.

Definition. (Goguen et al., 1978). Let Ω (ambiguously) denote the set of all operator symbols in the S-sorted signature Ω, ie, $\bigcup_{w \in S^*, s \in S}(\Omega_{w,s})$. Let $\langle T_{\Omega,s}\rangle_{s \in S}$ be the family of the smallest sets of strings contained in $(\Omega \cup \{\underline{(}\,,\underline{)}\})^*$ satisfying the following two conditions (here $\{\underline{(}\,,\underline{)}\}$ is a two element set disjoint from Ω, though except for this definition we shall omit the underlines):

1. $\Omega_{\lambda,s} \subseteq T_{\Omega,s}$;
2. If $\sigma \in \Omega_{w,s}$, $w = s_1 \ldots s_n$ and $t_i \in T_{\Omega,si}$, $i = 1..n$, then $\sigma\underline{(}t_1 \ldots t_n\underline{)} \in T_{\Omega,s}$.

Such strings are usually called *Ω-words*. The family $\langle T_{\Omega,s}\rangle$ can be made into an Ω algebra by defining the operations:

1. For $\sigma \in \Omega_{\lambda,s}$, $\sigma_{T_\Omega} = \sigma \in T_{\Omega,s}$;
2. For $\sigma \in \Omega_{w,s}$, $w=s_1...s_n$ and $t_i \in T_{\Omega,si}$, $i=1...n$, $\sigma_{T_\Omega}(t_1,...,t_n) = \sigma(t_1...t_n) \in T_{\Omega,s}$

Such a T_Ω is initial in Alg_Ω, the category of all Ω-algebras.

We occasionally make use of the notion of the *final* algebra in Alg_Ω and hence a method for constructing such an algebra is desirable. It is much simpler than the initial algebra construction and requires us simply to choose the carrier of each sort to be a *singleton* set and define the operators accordingly. We sometimes refer to this algebra as the *degenerate* Ω-algebra.

It is important to see (we make frequent use of the fact) that although the structure of an Ω-algebra $\mathcal{A}$ is specified by a certain subset of the set of all finitary operations among the carriers of $\mathcal{A}$, the important role is played not merely by the set of operations defined by Ω, but by the set of all operations obtainable from them by composition. The single-sorted case is treated in Cohn (1981) and Manes (1976) in terms of the notion of a *clone* (closed set of operations) on a set of M which briefly is a set of operations on M that is closed under composition and contains the projection functions (selecting the ith element of a tuple). The *clone of action* of a single-sorted signature Σ on M is the clone generated by the operators defined by Σ.

The (formal) extension of the notion to the many-sorted case is straightforward but a little tedious. We prefer to approach it from the much more intuitively (and notationally) pleasing idea of *derived operators*. It is straightforward to extend the notion of Ω-words to include "variables" as follows. First fix an S-indexed family $\langle X_s \rangle_{s \in S}$ of sets of variables. It serves no purpose here to consider the effect of limiting the number of variables, so we assume all the X_s are infinite. Clearly, we can construct a new signature $\Omega(X)$ which is derived from Ω by adding each $x \in X_s$ to $\Omega_{\lambda,s}$ for all $s \in S$, thus considering (temporarily) each variable to be a constant of the appropriate source. Now, by simply generating $T_{\Omega(X)}$ we have an algebra whose carriers consist of Ω-words with variables. $T_{\Omega(X)}$ when considered as an Ω-algebra rather than an $\Omega(X)$-algebra is usually denoted $T_\Omega(X)$ and called the *free Ω-algebra generated by* X. Now any word of sort s containing variables $x_1,...,x_n$ respectively of sorts $s_1,...,s_n$, say $t(x_1...x_n)$ defines a *derived operator* t of arity $s_1...s_n$ and sort s for any

Ω-algebra. The set of all such derived operators of Ω is denoted $\overline{\overline{\Omega}}$ and coincides with the clone of action of Ω for any particular Ω-algebra.

2.2 Equational Presentations

In this section we introduce the notion of an *equational presentation* and treat it only in terms of many-sorted algebras, leaving the concept of an *algebraic theory* to §2.3.

In §2.1 we introduced signatures as a means by which "species" of algebras may be characterised by their "shape". To allow further restriction we introduce the notion of a set of axioms in the form of first-order, universally-quantified equations, that a particular algebra may or may not satisfy. In this way we may characterise species of algebras with a certain signature which also satisfy a certain set of equations. We begin by developing the *algebraic* concepts of *equation* and *satisfy*.

Definition. Given an S-sorted Ω-algebra A and an S-indexed family of sets of variables $X=\langle X_s\rangle$. Any function $\alpha: X \to A$ (actually a family of functions $\langle \alpha_s: X_s \to A_s \rangle_{s \in S}$) is called an *assignment* of values of sort s in A to variables of sort s in X.

Using this idea we can formalise the notion of evaluating an expression (ie a term of $T_\Omega(X)$) given values for the variables.

Result. Let A be an Ω-algebra and $\alpha: X \to A$ an assignment. Then there is a unique Ω-homomorphism $\bar{\alpha}: T_\Omega(X) \to A$ that extends α in the sense that $\bar{\alpha}_s(x)=\alpha_s(x)$ for all $s \in S$ and $x \in X_s$.

Despite the notation and abstract formulation, what we are doing here is quite familiar. Any $t \in T_\Omega(X)$ is an expression involving some variables from X and α is an assignment of values from A to those variables. Further, since A is an Ω-algebra the symbols from Ω appearing in t already have some corresponding meaning in A. Hence $\bar{\alpha}(t)$ *evaluates* t to get a unique value in A, so $\bar{\alpha}$ can be seen as the process of evaluation of expressions with the values of the variables given by α.

Definition. An *Ω-equation* is a triple $\langle X,t_1,t_2\rangle$ where X is an S-indexed set (of variables) and $t_1,t_2 \in T_\Omega(X)_s$ for some s. A more suggestive notation is $\forall X, t_1=t_2$ though since a suitable X can be deduced from t_1 and t_2, we generally write $t_1=t_2$.

Definition. An Ω-algebra A *satisfies* an Ω-equation $\langle X,t_1,t_2\rangle$ if and only if $\alpha(t_1) = \alpha(t_2)$ in A for *all* assignments $\alpha: X \to A$. A satisfies a set E of Ω-equations iff A satisfies each $e \in E$.

Definition. An *equational presentation* (or just *presentation*) P is pair $\langle\Omega,E\rangle$ where Ω is a signature and E is a set of Ω-equations. An Ω-algebra that satisfies E is called a *P-algebra*.

If we continue our Stack-of-Integers example begun in §2.1, we may give an equational presentation Stk consisting of the signature Σ together with the following set of Σ-equations.

1. pop(push(s,n)) = s
2. pop(empty) = empty
3. top(push(s,n)) = n

The Σ-algebra S defined in §2.1 is a Stk-algebra. We will leave the proof that this is the case until later when we have developed a proof-theoretic notion of an equation being satisfied to complement the model-theoretic one above.

Generally in universal algebra, given a presentation $P=\langle\Omega,E\rangle$, the class of all P-algebras is termed the *E-variety of Ω-algebras* and studied as a class of objects. We, however, prefer to add a little more structure by constructing the category of P-algebras, denoted Alg_P, from the class of all P-algebras together with all Ω-homomorphisms between them. As for Alg_Ω (§2.1), Alg_P always has an initial algebra and we now proceed with a method for the construction of T_P, the E-quotient of T_Ω, that is always initial in Alg_P. Firstly we need a little more machinery.

Definition. An *Ω-congruence* $\equiv$ on an Ω-algebra A is a family $\langle\equiv_s\rangle_{s\in S}$ of equivalence relations $\equiv_s$ on A_s for each $s \in S$, such that if $\sigma \in \Omega_{s1\ldots sn,s}$, $a_i,a'_i \in A_{si}$ and $a_i \equiv_{si} a'_i$ for $i=1..n$, then $\sigma_A(a_1,\ldots,a_n) \equiv_s \sigma_A(a'_1,\ldots,a'_n)$.

We now need to discuss the notion of taking the quotient of an algebra for a congruence defined on that algebra. If A is an Ω-algebra and $\equiv$ is an Ω-congruence on A, we define a new Ω-algebra called the *quotient* of A by $\equiv$, denoted $A/\equiv$ as follows. For each $s \in S$ let $(A/\equiv)_s$ be $A_s/\equiv_s$, the set of

$\equiv_s$-equivalence classes of A_s. If we denote the equivalence class of a by [a], we may now make the S-indexed family $A/\equiv$ into an Ω-algebra by defining the operations as follows:

1. If $\sigma \in \Omega_{\lambda,s}$ then $\sigma_{A/\equiv} = [\sigma_A]$
2. If $\sigma \in \Omega_{s1...sn,s}$ and $[a_i] \in (A/\equiv)_{si}$ for i=1..n, then $\sigma_{A/\equiv}([a_1],...,[a_n])=[\sigma_A(a_1,...,a_n)]$

The final step in the development involves determining a congruence from the equations of a presentation. A set of equations E determines a relation E(A) on any Ω-algebra A consisting of the family of sets of all pairs $\langle\alpha_s(t_1),\alpha_s(t_2)\rangle$ where $\langle X,t_1,t_2\rangle \in E_s$ and α is an assignment $X \to A$. There is a least Ω-congruence on A containing E(A), referred to as the ***Ω congruence generated by*** E(A) on A. At last we can define T_P and give the initiality result.

Result. Let P = $\langle\Omega,E\rangle$ be a presentation and let $\equiv_E$ be the Ω-congruence on T_Ω generated by E(T_Ω). Then $T_\Omega/\equiv_E$, the quotient of T_Ω by $\equiv_E$, which we shall denote T_P, is the initial algebra of Alg_P.

Continuing our Stack-of-Integers example, the carrier of sort Stack in T_{Stk} has elements

[empty],
[push(empty,n)], $\forall$ n $\in T_{Stk,Int}$
[push(push(empty,n),m)], $\forall$m,n $\in T_{Stk,Int}$
and so on.

The functions are defined along the lines of
$push_{T_{Stk}}$([s],[n]) = [push(s,n)].

It should be noted in passing that since Stk includes no equation involving "top(empty)", the carrier of sort Int in T_P includes the elements

[top(empty)],
[succ(top(empty))],
[succ(succ(top(empty)))]

and so on, implying that such elements can validly be "pushed" onto a stack. This may or may not be seen as undesirable depending on the application. If such a situation is considered unacceptable, error terms may be introduced, but this too is fraught with danger and may require the equations be "error-conditioned" (Goguen, 1978).

Finally we note that we can construct an algebra by distinguishing a single element from each of the equivalence classes of the carriers of T_P. Clearly such an algebra is isomorphic to T_P and is hence initial in Alg_P. Such an algebra is generally called a *canonical term algebra*.

We complete this section with a brief discussion of *equational deduction* based largely on the work of Goguen & Meseguer (1982). As mentioned above in the context of showing *S* to be a Stk-algebra, a proof-theoretic notion that coincides with the model-theoretic definition of an equation being satisfied is desirable, especially given that our central concern is the development of proof techniques. Thus we need to define an *equational logic* (deduction system) that is *sound* in the sense that new equations that are deduced are always satisfied by any algebra satisfying the given equations, and that it is *complete* in the sense that *every* equation satisfied by all the algebras satisfying the given equations can be deduced using the rules of the system.

Unfortunately the usual rules of equational deduction, reflexivity, symmetry, transitivity and substitutivity, while they may be sound and complete for the single-sorted case, are not sound when generalised to the many-sorted case. We demonstrate this with an example taken from Goguen & Meseguer (1981) based on the following presentation, B.

Signature
sort Bool
  tt : $\rightarrow$ Bool
  ff : $\rightarrow$ Bool
  $\neg$: Bool $\rightarrow$ Bool
  $\wedge$: Bool $\times$ Bool $\rightarrow$ Bool
  $\vee$: Bool $\times$ Bool $\rightarrow$ Bool
sort A
  foo : A $\rightarrow$ Bool

Equations
(we use the more suggestive infix form for the operator symbols)

1. $\neg(tt) = ff$
2. $\neg(ff) = tt$
3. $b \vee \neg b = tt$
4. $b \wedge \neg b = ff$
5. $b \vee b = b$
6. $b \wedge b = b$
7. $foo(a) = \neg foo(a)$

Now using the usual system of equational deduction we may show:

$$\begin{aligned} tt &= foo(a) \vee \neg foo(a) && (3)\\ &= foo(a) \vee foo(a) && (7)\\ &= foo(a) && (5)\\ &= foo(a) \wedge foo(a) && (6)\\ &= foo(a) \wedge \neg foo(a) && (7)\\ &= ff && (4) \end{aligned}$$

If such rules of deduction were sound then we would expect $tt = ff$ to hold in *every* B-algebra, but this is not the case. There is a B-algebra *Bar* where Bar_{Bool} = {true,false}, Bar_A = {}, foo is the empty function and all the Bool functions are the usual ones. Clearly true ≠ false in *Bar* and thus the rules are not sound.

Goguen & Meseguer (1981) and (1983) provide quantified versions of reflexivity, symmetry, transitivity and substitutivity that are sound for many-sorted logic. This is why the set of variables used is explicitly included in the structure of equations. Two more rules are required to make the deduction system complete: *abstraction* and *concretion* that basically provide a means for adding and removing variables from that part of the equation.

We do not give the details of this deduction system here since (fortuitously) for the examples we consider in this thesis, as well as many other applications, the ordinary rules are indeed sound and complete. to make this precise we need the following notions (Huet & Oppen, 1980).

Definition. If Ω is an S-sorted signature, we say that $s \in S$ is *strict* in Ω if and only if there is either
1. some $\sigma \in \Omega_{\lambda,s}$, or
2. some $\sigma \in \Omega_{s1\ldots sn,s}$ where s_i is strict in Ω, i=1..n.

The signature Ω is *sensible* if and only if for every $\sigma \in \Omega_{s1\ldots sn,s}$, if s is strict then so are all the s_i, i=1..n.

We prefer this definition to the concept of *non-void* sorts in Goguen & Meseguer (1983) since it is slightly more general. The final result follows:

Result. The ordinary rules of equational deduction are sound and complete for a signature Ω if and only if Ω is sensible.

For the presentation B above, the signature is not sensible since Bool is strict (tt : $\rightarrow$ Bool) but A is not, and foo : A $\rightarrow$ Bool. We make a blanket appeal to this result, claiming all the signatures of later chapters to be sensible.

Further contributions to many-sorted equational logic have recently been put forward by Padawitz & Wirsing (1984) and MacQueen & Sanella (1984).

2.3 Algebraic Theories

It is fairly natural for computer scientists to consider equational presentations as a *specification language* as witnessed by the bulk of the work using algebra for the definition of data abstractions. In the previous section we used presentations to specify a class of algebras; in this section we wish to explore an alternative "semantics" for such a "language".

The object we now claim to be specified by a presentation is a *(many-sorted) algebraic theory*, which can be seen as one possible formalisation of the loose, intuitive mathematical notion of a "theory". There are many alternative definitions of algebraic theories, depending on the background against which they are being developed. The most common, including the original definition (Lawvere, 1963) is in terms of a particular category whose objects are the natural numbers, though even then there is considerable variation. Goguen et al. (1975) use theory congruences, Elgot (1973) takes a more axiomatic approach, while Kamin (1979) avoids using category theory explicitly yet still constructs the same object. Other formulations are based on triples (or monads) as in Manes (1976) and Cohn (1981), or functors (Goguen et al., 1975), or even an algebra (Goguen, 1975), (Fasel, 1980). A more accessible definition, though perhaps less amenable to mathematical discourse, is that of the related notion of *Ω-theory* by Burstall & Goguen (1979) based on *signed theories* (Goguen & Burstall, 1984a, 1984b). Our definition of a many-sorted algebraic theory is similar in style to that of an Ω-theory. Unfortunately we pay a price for using such an intuitively appealing definition by making some of the related definitions slightly more difficult and hence a little indirect at times. A more rigorous development of (most of) the same concepts introduced here is given by Goguen et al. (1975) in terms of a more traditional definition of algebraic theory.

We will give a model-theoretic definition of an algebraic theory in terms of the following notions.

Definition. Given a set E of Ω-equations, let E^* denote the set of all Ω-algebras which satisfy every equation in E.

Given a set M of Ω-algebras, let M^* denote the set of Ω-equations satisfied by every algebra in M.

Given an Ω-algebra A, let $\overline{A}$ denote A considered as an

$\overline{\Omega}$-algebra. $\overline{A}$ is the largest $\overline{\Omega}$-subalgebra of A.

Given a set M of Ω-algebras, let M^+ denote the set of $\overline{\Omega}$-algebras such that $\overline{A} \in M^+$ iff $A \in M$.

Given a set E of Ω-equations, the *clone-closure* of E is the set E^{*+*}, denoted $\tilde{E}$.

We may now give the central definitions of this section.

Definition. An *algebraic theory* is a presentation $\langle\Omega,E\rangle$ such that Ω is closed and E is clone-closed.
Thus $\Omega = \overline{\Omega}$ and $E = \tilde{E}$.

Definition. The *algebraic theory presented by* $P = \langle\Omega,E\rangle$ is $\langle\overline{\Omega},\tilde{E}\rangle$ and is denoted Th_P.

Given this notion, we call the algebras of Alg_P the *models* of the theory presented by P. Hence we may view algebraic theories as a "higher level" of semantics for presentations, fitting in between the presentation and the class of algebras specified.

Our definition differs from the notion of an Ω-theory which retains the original signature and only takes the closure of the equations. It is more in the spirit of the original work (Lawvere, 1963) to abstract away from the signature as well as the equations. The notion of an Ω-theory is slightly simpler and adequate for Burstall & Goguen's (1979) semantics of the specification language, Clear. The main advantage of using our more abstract notion is that we can avoid treating theory morphisms and derivors separately. In contrast, it was clearly more convenient for Burstall & Goguen to maintain such a separation since Clear has a specific *derive* operator. Much of the remaining material in this section consists of reworking the material of part 2 of Burstall & Goguen (1979) in terms of our notion of algebraic theory.

Definition. A *signature morphism* from an S-sorted signature Ω to an S'-sorted signature Ω' is a pair $\langle f,g\rangle$ consisting of a map $f: S \rightarrow S'$ and a family of maps $g_{w,s}: \Omega_{w,s} \rightarrow \Omega'_{f^*(w),f(s)}$, where f^* is the pointwise extension of f to strings.

Thus a signature morphism is a map that takes sorts to sorts and operators to operators, preserving their arities and sorts.

Definition. Given two theories, T and T', say $\langle\Omega,E\rangle$ and $\langle\Omega',E'\rangle$. A *theory morphism* from T to T', is a signature morphism $\sigma: \Omega \to \Omega'$ such that $\sigma(e) \in E'$ for each $e \in E$.

We leave the notion of σ being extended to equations as intuitively understood here, and refer the reader to Burstall & Goguen(1979), part 2.3 for a rigorous definition. A theory morphism is therefore a signature morphism that preserves the axioms. It is worth noting here that the above two definitions are identical to those for Ω-theories. It should be borne in mind that our theories only differ from Ω-theories by insisting that the signature be closed, so we should expect some overlap.

Result (Presentation Lemma). Given two presentations $\langle\Omega,E\rangle$ and $\langle\Omega',E'\rangle$ of theories T and T' respectively. If $\sigma : \Omega \to \overline{\overline{\Omega}}'$ is a signature morphism then σ can be uniquely extended to $\overline{\sigma}: \overline{\Omega} \to \overline{\Omega}'$. (This closely follows the idea of σ being extended to equations; see Burstall & Goguen, 1979, part 2.3). Now $\overline{\sigma} : T \to T'$ is a theory morphism if and only if $\sigma(e) \in \tilde{\tilde{E}}'$ for each $e \in E$.

Thus, if we can define $\sigma : \Omega \to \overline{\overline{\Omega}}'$ such that the equations of E are still satisfied, we may deduce $\overline{\sigma} : \overline{\Omega} \to \overline{\Omega}'$ and be sure that all the equations of $\tilde{\tilde{E}}$ are satisfied. This is of considerable importance for our work since it provides a proof technique for establishing whether a given signature morphism is a theory morphism in terms of the presentations alone.

Finally we may ask how a theory morphism can be reflected in the models of the source and target theories. In short, it provides a means for deriving a model of the source theory from any model of the target theory; in the opposite direction to the theory morphism, so to speak. Though we are more interested in theory morphisms in this dissertation the concept is also applicable to signature morphisms, as reflected by the following definition.

Definition. Given an S-sorted signature Ω and an S'-sorted signature Ω' together with a signature morphism $\sigma : \Omega \to \Omega'$. If $\mathcal{A}$ is any Ω'-algebra then there is an Ω-algebra, denoted $U_\sigma(\mathcal{A})$ where the carriers and operators correspond as follows:

1. $(U_\sigma(A))_s = A_{\sigma(s)}$ for all $s \in S$
2. $\tau_{U_\sigma(A)} = \sigma(\tau_A)$ for all operator symbols $\tau \in \Omega$.

The notion of a U_σ-derivor can be extended to theory morphisms in a straightforward manner.

Definition. Given two theories T and T' presented by $\langle\Omega,E\rangle$ and $\langle\Omega',E'\rangle$ respectively, together with a signature morphism $\sigma: \Omega \to \overline{\Omega}'$ that is (or can be extended to) a theory morphism $T \to T'$. If A is any $\langle\Omega',E'\rangle$-algebra (model of T') then $U_\sigma(A)$ is a model of T, constructed as above.

A special case we shall occasionally find useful is where σ is an *inclusion* morphism; that is the sorts of Ω are a subset of the sorts of Ω', similarly for the operators, and $\sigma(s) = s$.

Definition. Given an S-sorted signature Ω and an S'-sorted signature Ω', where $S \subseteq S'$ and $\Omega_{w,s} \subseteq \Omega'_{w,s}$ for all $w \in S^*$, $s \in S$. If σ is a signature morphism $\Omega \to \Omega'$ such that $\sigma(s) = s$ for all $s \in S$ and $\sigma(\tau) = \tau$ for all $\tau \in \Omega$ (ambiguously denoting all the operator symbols in Ω), then for any Ω'-algebra A, we call the Ω-algebra $U_\sigma(A)$ the *Ω-reduct* of A.

In essence, taking the Ω-reduct of an Ω'-algebra is achieved by "forgetting" the sorts and operators of Ω' that are not also in Ω.

It is possible to avoid all this extra machinery (for our applications at least) by dealing with derivors on individual algebras rather than morphisms between theories. We prefer not to do this for a number of reasons. First, it is usually advantageous to work at the highest available level of abstraction and generality. Second, by finding a theory morphism $T \to T'$ we have a means of deriving a T-algebra from *any* T'-algebra, whereas we would need to repeat the proof for each algebra were we to use the derivor machinery. Finally, in later chapters a pleasing and convenient split of semantic congruences and compiler definitions into two-stage connections is reflected by the separation of models from their theories.

Note that we will occasionally allow an abuse of notation (when no confusion can arise) that involves denoting theories by their presentations. Thus, given a presentation P, we may speak of "the models of P" rather than "the models of Th_P" and so on.

Chapter 3
Specification of Programming Language Semantics

In this chapter we describe the technique we have adopted for specifying the semantics of programming languages. It has been influenced by the Oxford style of denotational semantics in that we provide a set of semantic domains and semantic functions from an abstract syntax for the language. It has also been influenced by the early work on algebraic specification of abstract data types (eg Liskov & Zilles, 1975). As such it is relatively unsophisticated in that our specifications consist only of a signature and a set of first-order equations, thus relinquishing the expressive power of parameterized modules, conditional axioms and structured "theory - building" operations.

The advantage of such a plain specification language is that it retains very simple semantics and in fact it directly reflects those semantics. Further, it seems unwise at this stage to settle on a choice between Clear, OBJ etc. especially when none is completely suited (semantically) to the present endeavour.

The disadvantage of such unstructured specifications as ours is quite obvious, however. As P. Lucas points out in the first session discussion in (Bjorner, 1983),

> "... it was clear almost from the beginning that having a big language definition on the one hand and a big implementation on the other hand and then asking whether the implementation obeys the rules of the definition is not really a viable question to ask.... So it was clear that it was necessary to decompose this gigantic task into smaller sub-tasks that are manageable. In other words, we are looking for modularity. Modularity of the definition as well as the proofs of correctness."

Thus for a full-scale venture such as a large language or frequent use, our specification language absolutely requires to be structured. However, since in this dissertation we are investigating proof techniques that are based on the semantics of our specification language we intend to persevere with the simple notation we have adopted. Development of a more modular style must take a high priority in any further development of this work.

3.1 Semantic Presentations and Models

We now proceed to discuss the semantics of our specification language, largely in terms of an example using the lambda calculus. Though our presentations look like a simplified version of OBJ (Goguen & Tardo, 1979) they have different semantics. While an OBJ object represents a particular algebra (the initial one), our presentations represent algebraic theories. However, we are not directly interested in the theories themselves, but rather their classes of models (algebras). It is each of these models that give a concrete semantics whereas the presentation can be considered as a kind of "semantic schema". It may seem at this point that we are interested in an equational variety of algebras rather than an algebraic theory. While this is partly true, we will later find it convenient to define relationships between the entire classes of models of two presentations, rather than individual algebras and more elegant machinery exists for doing this in terms of theories rather than varieties.

3.1.1 A Presentation for the Lambda Calculus

Rather than dealing with the pure lambda calculus (Church, 1941), we extend it by including constant valued atoms. While this is strictly unnecessary, it does make the operational semantics considered in §4.2 somewhat more tangible. Thus, a lambda expression is either a constant or an identifier; or an abstraction in which case it consists of a bound variable which is an identifier and a body which is a lambda expression; or an application being an operator-operand pair of lambda expressions. We stress again that the presentation that follows represents an algebraic theory and that the semantics of the lambda calculus are given by the models of that theory rather than by the theory itself.

<u>Signature (Σ)</u>

sort Lambda.

- constant : B $\rightarrow$ Lambda
- var : Ide $\rightarrow$ Lambda
- abstraction : Ide $\times$ Lambda $\rightarrow$ Lambda
- application : Lambda $\times$ Lambda $\rightarrow$ Lambda

sort env.

arid : $\rightarrow$ env
bind : env $\times$ Ide $\times$ W $\rightarrow$ env
find : env $\times$ Ide $\rightarrow$ W
sort W.
injB : B $\rightarrow$ W
injA : Abstr $\rightarrow$ W
err : $\rightarrow$ W
appl : W $\times$ W $\rightarrow$ W
sort Abstr.
A : Lambda $\times$ Ide $\times$ env $\rightarrow$ Abstr
M : Lambda $\times$ env $\rightarrow$ W

Equations (ε)
1. find(arid,x) = err
2. find(bind(e,x,w),y) = *if* x=y *then* w *else* find(e,y)
3. M(constant(b),e) = injB(b)
4. M(var(x),e) = find(e,x)
5. M(abstraction(x,η),e) = injA(A(η,x,e))
6. M(application(α,β),e) = appl(M(α,e), M(β,e))
7. appl(injB(b),w) = err
8. appl(injA(A(η,x,e)),w) = M(η,bind(e,x,w))
9. appl(err,w) = err

LC - lambda calculus presentation

Note that although the abstract syntax was included as part of the signature as sort Lambda, it is generally more convenient to employ the usual strings of the language. Note also that the presentation LC is somewhat incomplete with sorts B and Ide being left unspecified. We will frequently make use of such deliberately loose notation especially with respect to Ide since we are generally not interested in the set of identifiers save for the fact that there are enough of them and that we can test for their equality. The following presentation is sufficient provided we wish to have an unlimited pool of identifiers.
sort Ide.
first : $\rightarrow$ Ide
next : Ide $\rightarrow$ Ide
equal : Ide $\times$ Ide $\rightarrow$ Bool
equal (first, first) = tt

equal (first, next(y)) = ff
equal(next(x),first) = ff
equal(next(x),next(y)) = equal(x,y).

We will also in general assume a sort Bool with two constants tt and ff to be available and will only explicitly include it in presentations when further operators are required. Similarly we will feel free to use the apparently generic (mixfix) operator *if* _ *then* _ *else* _ in the equations of presentations in the knowledge that such an operator is easily defined for any given target sort X as follows:

$\textit{if}\ \_\ \textit{then}\ \_\ \textit{else}_X\ \_ : \text{Bool} \times X \times X \rightarrow X$
$\textit{if}\ \text{tt}\ \textit{then}\ x_1\ \textit{else}_X\ x_2 = x_1$
$\textit{if}\ \text{ff}\ \textit{then}\ x_1\ \textit{else}_X\ x_2 = x_2$

Again, in some circumstances such as when the presentation of Bool consists of more than just two constant operators (eg §4.6), such specifications of *if then else* operators may be inappropriate, so in those cases we will explicitly include them in the presentation. However, for most of the examples treated in this thesis the style of definition given above is sufficient and we therefore assume an *if then else* operator of every sort to be available without actually writing down the details.

In a slightly different vein sort B, representing basic values, is left unspecified since any particular choice of model for B will not impinge on our discussion of LC and its models, so it could be seen as a primitive form of parameterization. Also it may seem a little unusual to have a semantic domain (sort B) forming part of the syntax as described by "constant : B → Lambda" rather than the more usual complete separation of syntactic and semantic domains as in standard denotational semantics. There, typically, numerals and numbers are distinguished and an obvious semantic function say $\mathcal{N} : \text{Num} \rightarrow \mathbb{N}$, is said to exist but is left unspecified. (See Stoy, 1977 for example). In our algebraic framework we may easily mix syntactic and semantic sorts and we often choose to do so in an effort to avoid such vagueness.

As an aid to the reader an intuitive interpretation of the lambda calculus presentation follows. As already discussed, sort B represents a domain of basic values, Ide a domain of identifiers and Lambda the parse trees of the language of λ-expressions and sort W represents the underlying domain of "values" of λ-expressions and is basically intended to look like the sum of B and Abstr. Hence the two operators injB and injA depict the usual injection functions. Sort W is also provided with an operator appl for "applying" elements of W to each other, and a distinguished error term in case such applications go wrong. Equations 7, 8 and 9 axiomatise the behaviour of appl. Sort env represents environments where information regarding the binding of values of sort W to identifiers is kept and the intention of the operators bind and find and the constant arid is clear from equations 1 and 2. The operator A and the sort Abstr may at first appear somewhat mysterious and incompletely specified. However, if we keep in mind our intention that the presentation is a "semantic schema", and that the algebras provide the actual semantics, it is clear that possible models for Abstr may include $[V \to V]$ where V is a suitable model of W, or "closures" (Landin, 1964) consisting of the information required to represent an abstraction (ie its body, its bound variable and the current environment) so that its application to another expression may be simulated. These two cases will be dealt with in detail in §3.1.2 and §4.2 respectively. Finally, the operator M represents the semantic function as defined by equations 3 to 6 and is isolated from the other operators in the signature to emphasise its distinguished role.

It seems most natural to view the presentation LC as describing an operational semantics. In other words it is seen as specifying a set of rewrite rules on terms rather than a set of operator symbols and axioms. It is interesting to note that this intuitive interpretation of a presentation like LC is (always) the initial model of the theory presented. As pointed out earlier in this chapter (indeed, we have possibly laboured the point), we intend that no particular model is "the" semantics. Rather, we claim that there may be many models of LC other than the initial one that provide satisfactory semantics of the lambda calculus. Unfortunately not every model gives an acceptable semantics, an obvious case being the algebra in which the carrier of W is a singleton set. Further since M is described only by recursive equations any fixed-point satisfies the equations and is thus a legitimate choice for modelling M. The issue of characterizing those algebras that are satisfactory semantic models is taken up in §3.3.

3.1.2 A Model of the Lambda Calculus Presentation

Our discussion in this section centres around the (by now standard) denotational semantics of the lambda calculus given below. A very similar definition is given in Stoy (1977) along with detailed discussion of the domains involved.

<u>Domains</u>

$V = B + FUN$

$FUN = V \to V$

$ENV = Ide \to V$

<u>Semantic Function</u>

$Val : Lambda \times ENV \to V$

(V1) $Val[\![c{:}constant]\!]\rho = c$ *in* V

(V2) $Val[\![x{:}Ide]\!]\rho = \rho(x)$

(V3) $Val[\![\lambda x.\eta]\!]\rho = \lambda a.Val[\![\eta]\!]\rho\,[x/a]$ *in* V

(V4) $Val[\![\alpha(\beta)]\!]\rho = Val[\![\alpha]\!]\rho\,|FUN(Val[\![\beta]\!]\rho\,)$

where

$$v|FUN = \begin{cases} v & \text{if } v \in FUN \\ \bot & \text{otherwise} \end{cases}$$

and

$$\rho[x/a] = \lambda z. \begin{cases} \rho(z) & \text{if } z \neq x \\ a & \text{if } z = x \end{cases}$$

<u>denotational semantics of lambda calculus</u>

To proceed, we make the observation that the denotational semantics *is* a many-sorted algebra. It fits the definition very neatly since in the final analysis it consists of nothing more than a family of sets (the domains) and some functions among them. Note however, that in viewing such semantics as an algebra requires that we be more explicit about identifying exactly which functions are being used, a detail that is not an issue for the usual view of denotational definitions. Put another way, having decided to view the denotational semantics as an algebra, we must then decide on a signature for that algebra. It will be seen below that we have some degree of choice in this matter.

Our overall aim in this section is to show that the algebra (call it Den) associated with the denotational definition above is a model of LC. To do this we need only to show that Den has the signature Σ as given in LC (or at least *arrange* for it be be so) and further to show that Den satisfies the equations ε given in LC.

It is easy to demonstrate that Den has the same signature as LC by firstly defining a correspondence between the sorts of LC and the domains (carriers) of Den and then listing the operator symbols of Σ each paired with a function from Den of corresponding arity and sort. Note that we are not concerned that the denotational definition may involve other functions which have no associated operator symbol in Σ.
Instantiate the sorts as follows:

sort env :	ENV
sort W :	V
sort Abstr :	FUN

Then the operator symbols and functions correspond as follows:

Σ	Den
injB:B → W	_ *in* V
injA: Abstr → W	_ *in* V
err: → W	⊥
arid: → env	λx.⊥
bind: env × Ide × W → env	_[_/_]
find: env × Ide → W	_(_)
A: Lambda × Ide × env → Abstr	λa.Val⟦_⟧_[_/a]
appl: W × W → W	(_ | FUN)(_)
M: Lambda × env → W	Val

Some of the functions listed under Den may look a little strange at first glance. The underline notation has been used where the operation has been written in mixfix notation (Mosses, 1980). Thus _[_/_] : env × Ide × V → env, eg ρ[x/a]. Further, some of the functions have been derived from simpler ones by means of composition. For example, (_ | FUN)(_) uses projection and function application. It is in this sense that we have "arranged" for Den to have the signature Σ rather than some other signature containing say, an operator for each of the primitive functions involved in the denotational semantics. In fact, it is not at all clear that any agreement could be reached as to the identification of these primitive functions. The central point is that the abstract syntax, semantic domains and valuations are the nucleus of any denotational definition. We are therefore at liberty to install them in any algebra that suits our purpose and the other sorts and operator symbols (carriers and operators) present in that algebra may be chosen arbitrarily without affecting the intended semantics.

To establish that Den satisfies the equations of LC, the equations may be translated into expressions of Den using the signature correspondence described above and then verified using the definitions of the operations of Den. In fact, having once recognised the signature correspondence, this part of the proof goes through very easily. Note that since Σ is a *sensible* signature (for any reasonable B) as discussed in chapter 2, we have no problems applying the ordinary rules of equational deduction.

1. $\lambda z.\bot(x) = \bot$
   {find(arid,x) = err}
   Immediately true.
2. $\rho[y/v](x) =$ if $x=y$ then v else $\rho(x)$
   {find(bind(e,y,w),x) = *if* x=y *then* w *else* find (e,x)}
   Can easily be shown from the definition of $\rho[y/v]$ by considering the cases $x=y$ and $x \neq y$.
3. $\mathrm{Val}[\![b{:}\mathrm{constant}]\!]\rho = b$ *in* V
   {M(constant(b),e) = injB(b)}
   Immediate by V1 of Den.
4. $\mathrm{Val}[\![x{:}\mathrm{Ide}]\!]\rho = \rho(x)$
   {M(var(x),e) = find(e,x)}
   Immediate by V2.
5. $\mathrm{Val}[\![\lambda x.\eta]\!]\rho = \lambda a.\mathrm{Val}[\![\eta]\!]\rho\,[x/a]$ *in* V
   {M(abstraction(x,η),e) = injA(A(η,x,e))}
   Immediate by V3.
6. $\mathrm{Val}[\![\alpha(\beta)]\!]\rho = (\mathrm{Val}[\![\alpha]\!]\rho \mid \mathrm{FUN})(\mathrm{Val}[\![\beta]\!]\rho)$
   {M(application(α,β),e) = appl(M(α,e), M(β,e))}
   Immediate by V4.
7. $((b$ *in* $V) \mid \mathrm{FUN})(v) = \bot$
   {appl(injB(b),w) = err}
   Follows from definition of | FUN.
8. $((\lambda a.\mathrm{Val}[\![\eta]\!]\rho\,[x/a]$ *in* $V) \mid \mathrm{FUN})(v) = \mathrm{Val}[\![\eta]\!]\rho\,[x/v]$
   {appl(injA(A(η,x,e)),w) = M(η, bind(e,x,w))}
   Follows from definition of | FUN and lambda-substitution.

9. $(\bot \mid \mathrm{FUN})(v) = \bot$
{appl(err,w) = err}
Follows from definition of $\mid$ FUN and $\bot$.

Through this proof we have demonstrated that there are interesting models of algebraic theories other than the initial one, thus justifying our decision to view presentations as denoting theories rather than a single particular algebra.

Before leaving this example for the time being (we return to it in §4.2), some further points of clarification need to be discussed. First, it may not have escaped the readers attention that the functionality of Val in the denotational semantics was written Val: Lambda $\times$ ENV $\rightarrow$ V rather than the more usual Val: Lambda $\rightarrow$ ENV $\rightarrow$ V. Had we wished our presentation to reflect this curried version, a new sort representing [env $\rightarrow$ W] would have to be added to the signature and appropriate changes made to the equations as follows.

sort envtoW.
apply: envtoW $\times$ env $\rightarrow$ W
M': Lambda $\rightarrow$ envtoW
3'. apply (M'(constant(b)),e) = injB(b)
4'. apply(M'(var(x)),e) = find(e,x)
5'. apply(M'(abstraction(x,η,)),e) = injA(A(η,x,e))
6'. apply(M'(application(α,β)),e) =
appl(apply(M'(α),e),apply(M'(β),e))

Of course the fact that we are persisting with parenthesised prefix function notation makes the above changes look worse than they otherwise could, but there is no denying that treating the semantic functions of more complex cases in this way will quickly become unwieldy. For example, to reflect the curried nature of the single valuation (taken from Tennent, 1977) $\mathcal{A}$: Exp $\rightarrow$ Us $\rightarrow$ Md $\rightarrow$ U $\rightarrow$ K $\rightarrow$ C in a theory presentation would require 4 more sorts and 4 more *apply*-like operators than allowing $\mathcal{A}'$: Exp $\times$ Us $\times$ Md $\times$ U $\times$ K $\rightarrow$ C. When one considers the number of valuations required for a realistic language such additions to the signature would quickly become tedious and would certainly reduce the readability of the presentation.

It should not be surprising however, that a denotational semantics with the curried version of Val is also a model of LC. After all, Lambda $\rightarrow$ ENV $\rightarrow$ V and Lambda $\times$ ENV $\rightarrow$ V are isomorphic domains. The only change to the proof above that is necessary is a clarification of the correspondence between M and the new Val, for example Val⟦_⟧_ using the same underline notation as previously or perhaps more explicitly $\lambda\langle\eta,\rho\rangle.\mathrm{Val}[\![\eta]\!]\rho$. Effectively, we un-curry the model to suit the signature rather than currying the signature to suit the model. Clearly then, the rather non-standard functionality of the original Val was not strictly necessary, but was considered desirable at this initial expository stage.

As a further variation on the denotational semantics and its relation to LC, it is enlightening to consider changing applications to call by value rather than the call by name used in the current model. Basically, the difference is that in this mode of evaluation both the operator and operand are evaluated *before* the application itself is performed. Thus, under call by name an expression like $(\lambda y.0)((\lambda x.xx)(\lambda xx.xx))$ evaluates to 0 whereas it fails to terminate under call by value. The change required to the denotational semantics to reflect this alteration is to replace (V3) by the following equation

$$\mathrm{Val}[\![\lambda x.\eta]\!]\rho = (\mathit{strict}\,(\lambda a.\mathrm{Val}[\![\eta]\!]\rho\,[x/a]))\ \mathit{in}\ V.$$

A fuller discussion may be found in Stoy (1977). The function *strict* : FUN $\rightarrow$ FUN is defined such that for all $v \in V$,

$$\mathit{strict}(f)(x) = \begin{cases} \bot, \top \text{ or } ? & \text{if } x \text{ is respectively } \bot, \top \text{ or } ? \\ f(x) & \text{otherwise} \end{cases}$$

It is clear that this new algebra (call it Den') is not a model of LC, since it fails to satisfy equation 8 -

$$((\mathit{strict}\,(\lambda a.\mathrm{Val}[\![\eta]\!]\rho\,[x/a])\ \mathit{in}\ V)|\mathrm{FUN})(v) = \mathrm{Val}[\![\eta]\!]\rho\,[x/v].$$

The naive addition of another equation

10. appl(w,err) = err

is insufficient and compounds our problems rather than solving them. If we consider the expression appl(injA(A(0,y,e)),err), then by equation 10 this reduces to err. However by equations 9,4 and 2 it reduces to 0, thus "collapsing" the carrier of W to a single value. To overcome the problem we need to condition the equations on certain arguments not being error terms so that for example equation 8 only holds when $w \neq$ err. This conditioning can be carried out quite systematically as described in Goguen (1978) and in fact the specification language OBJ (Goguen & Tardo, 1979) has a syntax that explicitly reflects this technique of handling error terms.

3.2 Initial Algebra Semantics

Perhaps one of the most influential and successful applications of algebra to programming language semantics to date is the so-called *initial algebra semantics* reported in (Goguen, Thatcher, Wagner & Wright, 1977) and heavily used in (Thatcher, Wagner & Wright, 1979). Their aim was to unify some apparently diverse approaches to semantic definitions using the single but powerful concept of initiality. By observing that for any context-free grammar G there is a signature Σ such that T_Σ corresponds exactly to the parse trees of G, it is clear that any other Σ-algebra provides a semantics for the language of G through the unique homomorphism assigning "meanings" to all the terms of T_Σ. After filling in some more details, we investigate the ways in which denotational semantics fits into this approach and then generalize this relationship to cater for our "semantic model" concept.

To derive a signature corresponding to a context-free grammar we proceed as follows. Associate a sort with each non-terminal and an operator symbol with each production $N_0 \rightarrow \alpha_0 N_1 \alpha_1 \ldots N_k \alpha_k$ whose sort is N_0 and arity $N_1 N_2 .. N_k$. The signature we require is just that set of sorts and operator symbols. As a demonstration, consider the lambda calculus example of §3.1. In BNF, the syntax is written as follows with each production named to simplify expression of the signature.

(constant)	‹Lambda› ::= ‹B›
(var)	‹Lambda› ::= ‹Ide›
(abstraction)	‹Lambda› ::= λ‹Ide›.‹Lambda›
(application)	‹Lambda› ::= ‹Lambda›(‹Lambda›)

Choose sorts Lambda, B and Ide corresponding in the obvious way to the non-terminals. The operator symbol associated with the production (constant) has sort Lambda and arity B, and so forth. Using our standard notation to express the signature (call it Ω) we get:

constant: B → Lambda
var: Ide → Lambda
abstraction: Ide × Lambda → Lambda
application: Lambda × Lambda → Lambda

which is precisely the abstract syntax part of the signature Σ given in §3.1.1.

According to the principles of initial algebra semantics we now need only choose a suitable Ω-algebra S_Ω to define the meanings of λ-expressions since we automatically get a unique homomorphism from T_Ω to S_Ω. So choosing a carrier for each sort of Ω and defining a function for each operator symbol of Ω is all that is required. For this example we allow ourselves to be guided by the denotational definition given in §3.1.2 and we assume exactly the same domain definitions here. Associate carriers with the sorts in the following way

$$S_{\Omega,\mathrm{Lambda}} = [\mathrm{Env} \to \mathrm{V}]$$
$$S_{\Omega,\mathrm{Ide}} = \mathrm{Ide}$$
$$S_{\Omega,\mathrm{B}} = \mathrm{B}$$

and define the operators as follows (using lambda notation)

$$\mathrm{constant}_{S\Omega}(b) = \lambda\rho.(b \textit{ in } \mathrm{V})$$
$$\mathrm{var}_{S\Omega}(x) = \lambda\rho.\rho(x)$$
$$\mathrm{abstraction}_{S\Omega}(x,\eta) = \lambda\rho.(\lambda a.\eta(\rho[x/a])) \textit{ in } \mathrm{V}$$
$$\mathrm{application}_{S\Omega}(\alpha,\beta) = \lambda\rho.(\alpha(\rho) \mid \mathrm{FUN})(\beta(\rho)).$$

The underlying idea of initial algebra semantics that there is a unique homomorphism from the abstract syntax in the form of T_Ω to any S_Ω chosen to be the semantics of that language can be seen as an attempt to formalize what *constitutes* a semantics. Since we subscribe to such a view of semantics there will frequently be places in this dissertation where we will attempt to relate our style of semantics to the initial algebra style. As a precursor to the first and most detailed of these, we will now examine the connection between denotational and initial algebra semantics.

3.2.1 Denotational and Initial Algebra Semantics

The following quote from Goguen et al (1977) contains the general thrust of their claim that denotational semantics fits the initial algebra semantics concept.

"In general, the 'semantic equations' define the meaning of a syntactic construct C as a function F_C of the meanings of the

> components to that construct, and in so doing the semantic equations describe an algebra (the function F_C is the operation corresponding to the syntactic construct C) *and* say that semantics is a homomorphism."

While the general intent may be clear, the connection is discussed only briefly and rather informally and consequently leaves some questions of detail unanswered. Clearly if the semantic functions are to be homomorphisms then their definitions, the semantic equations must be constrained in some way, yet no explicit mention is made of such in the fundamental denotational semantics literature. However, in the "folklore" surrounding denotational semantics much emphasis is placed on the concept of *referential transparency* (Stoy, 1977), (Milne & Strachey, 1976), though again no hard-and-fast definition is given. One of the implications is that the meaning of a particular syntactic construct depends *only* on the meanings of its constituents. If we adhere to this principle then we immediately satisfy the requirements for the semantic functions to be homomorphisms. Nevertheless, inspection of some typical denotational definitions reveals several cases where the semantic equations do not appear to be entirely homomorphic, three classes of which were identified by Mosses (1983) and are treated below.

First the equations are sometimes non-compositional and therefore appear to deviate from the principle of referential transparency, eg.

$\mathcal{E}[\![e_1 - e_2]\!] = \mathcal{E}[\![e_1 + (-e_2)]\!]$

$\mathcal{C}[\![\text{repeat } c]\!] = \ldots\mathcal{C}[\![c]\!]\ldots\mathcal{C}[\![\text{repeat } c]\!]\ldots$

Such equations, usually seen as harmless shorthand, can be dealt with in two ways; either reject them as unacceptable and dismiss the semantic definition as non-denotational, or replace them with more acceptable versions. The first example can be rewritten by an expansion of $\mathcal{E}[\![e_1 + (-e_2)]\!]$ using the clauses for $\mathcal{E}[\![e_1 + e_2]\!]$ and $\mathcal{E}[\![-e]\!]$, and the second one can be rewritten with explicit use of the fixed-point operator.

Second, the creation of environments may directly involve identifiers rather than their denotations as in

$\mathcal{D}[\![\text{const } i = e]\!] = \ldots\rho[\varepsilon/i]\ldots.$

The way around this has already been mentioned in our lambda calculus example. We take Ide to be a semantic domain (as well as syntactic one) and leave the semantic function Ide → Ide implicit as an identity. Basically these problems only arises due to a lack of complete formality in the definitions.

Third, there are often several semantic functions for the same syntactic domain, such as $\mathcal{E}$, $\mathcal{L}$, $\mathcal{R}$: Exp → ... . We feel that it may have been exactly this problem that led Wand (1982) to suggest that the homomorphic nature of semantics was lost in continuation semantics where he claims"... the notion of the value of a subexpression is meaningless". The classic case for continuation semantics is a language with labels and gotos, and this requires at least two semantic functions operating on commands, one to collect up label values, the other to evaluate the commands themselves. The solution is straightforward if we consider the several functions as components of a single compound semantic function and use explicit projection and tripling to manipulate that function. A detailed example is given in §3.2.2.

Thus it appears that given a denotational semantics (within certain guidelines) we can re-express it directly as an initial algebra semantics, so denotational definitions are indeed a possible expression of our fundamental concept of "semantics". However Mosses' (1983) claim that we can "go the other way" needs to be tempered somewhat. Certainly any initial algebra semantics can be expressed in the *notation* of denotational semantics, but it will not necessarily be denotational. For example we can easily give an initial algebra semantics where the semantic algebra consists of the strings of the language or where procedure declarations are handled syntactically as in Goguen & Parsaye-Ghomi (1981), yet such domains are not acceptable in denotational semantics.

3.2.2 Semantic Models and Initial Algebra Semantics

Since our semantic presentations bear some resemblance to denotational definitions, at least in style, it would seem hopeful that our semantic models also characterize an initial algebra semantics. We intend to demonstrate that such is indeed the case and also to develop a little more formality about any requirements we wish to place on the form of semantic equations. The example we intend to use is a stripped-down language with only gotos, labels and another statement, whose action is undefined. Although it is not strictly necessary to do so, we give the presentation as well as the model with which we choose to work.

<u>Signature</u>

sort Program
    prog: Stmt → Program
sort Stmt
    seq: Stmt × Stmt → Stmt
    goto: Ide → Stmt
    labelled: Ide × Stmt → Stmt
    other: → Stmt
sort Env
    arid: → Env
    bind: Env × Ide × C → Env
    find: Env × Ide → C
    bindall: Env × Idlist × Clist → Env
sort C   {continuations}
    - depends on semantics of 'other' statement -
    err: → C
sort Idlist
    emptyi: → Idlist
    cati: Ide × Idlist → Idlist
    headi: Idlist → Ide
    taili: Idlist → Idlist
    appi: Idlist × Idlist → Idlist
sort Clist
    emptyc: → Clist
    catc: C × Clist → Clist
    headc: Clist → C
    tailc: Clist → Clist
    appc: Clist × Clist → Clist

$\mathcal{P}$: Program × Env × C → C
$\mathcal{C}$: Stmt × Env × C → C
$\mathcal{L}$: Stmt × Env × C → Env
$\mathcal{I}$: Stmt → Idlist
$\mathcal{M}$: Stmt × Env × C → Clist

<u>Equations</u>

1. find(arid,x) = err
2. find(bind(ρ,x,θ),y) = *if* x=y *then* θ *else* find(ρ,y)

3. bindall(ρ,xℓ,cℓ) = *if* xℓ = emptyi *then* ρ *else*
   bindall(bind(ρ,headi(xℓ),headc(cℓ)),
   taili(xℓ),tailc(cℓ))
4. headi(cati(x,xℓ)) = x
5. taili(cati(x,xℓ)) = xℓ
6. appi(xℓ,yℓ) = *if* xℓ = emptyi *then* yℓ
   else cati(headi(xℓ), appi(taili(xℓ),yℓ))
7. headc(catc(θ,cℓ)) = θ
8. tailc(catc(θ,cℓ)) = cℓ
9. appc(cℓ,dℓ) = *if* cℓ = emptyc *then* dℓ
   else catc(headc(cℓ),appc(tailc(cℓ),dℓ))
10. $\mathcal{P}$(prog(s),ρ,θ) = $\mathcal{C}$(s,$\mathcal{L}$(s,ρ,θ),θ)
11. $\mathcal{L}$(s,ρ,θ) = bindall(ρ,$\mathcal{I}$(s), $\mathcal{M}$(s,$\mathcal{L}$(s,ρ,θ),θ))
12. $\mathcal{C}$(seq(s1,s2),ρ,θ) = $\mathcal{C}$(s1,ρ,$\mathcal{C}$(s2,ρ,θ))
13. $\mathcal{C}$(goto(ℓ),ρ,θ) = find(ρ,ℓ)
14. $\mathcal{C}$(labelled(ℓ,s),ρ,θ) = $\mathcal{C}$(s,ρ,θ)
15. $\mathcal{C}$(other,ρ,θ) = ... (not specified)
16. $\mathcal{I}$(seq(s1,s2)) = appi($\mathcal{I}$(s1), $\mathcal{I}$(s2))
17. $\mathcal{I}$(goto(ℓ)) = emptyi
18. $\mathcal{I}$(labelled(ℓ,s)) = cati(ℓ,$\mathcal{I}$(s))
19. $\mathcal{I}$(other) = emptyi
20. $\mathcal{M}$(seq(s1,s2),ρ,θ) = appc($\mathcal{M}$(s1,ρ,$\mathcal{C}$(s2,ρ,θ)), $\mathcal{M}$(s2,ρ,θ))
21. $\mathcal{M}$(goto(ℓ),ρ,θ) = emptyc
22. $\mathcal{M}$(labelled(ℓ,s),ρ,θ) = catc($\mathcal{C}$(s,ρ,θ), $\mathcal{M}$(s,ρ,θ))
23. $\mathcal{M}$(other,ρ,θ) = emptyc

<u>GL - simple goto-language presentation</u>

Note that we have been rather meticulous with our notation in these equations. In future examples for the sake of readability we shall use more "normal" abstract syntax and where there can be no misunderstanding, implicit parentheses. The model (offered without proof) of GL we intend to work with is the denotational one given below.

$U = Ide \rightarrow C$

$\mathfrak{P}: Program \rightarrow U \rightarrow C \rightarrow C$

$\mathfrak{C}: Stmt \rightarrow U \rightarrow C \rightarrow C$

$\mathfrak{I}: Stmt \rightarrow Ide^*$

$\mathcal{M}: Stmt \rightarrow U \rightarrow C \rightarrow C^*$

$\mathfrak{P}[\![\textbf{begin } s \textbf{ end}]\!]\rho\theta = \mathfrak{C}[\![s]\!]\ \mathit{fix}\ (\lambda\rho'.\rho[\mathfrak{J}[\![s]\!]/\mathcal{M}[\![s]\!]\rho'\theta])\theta$
$\mathfrak{C}[\![s_1;s_2]\!]\rho\theta = \mathfrak{C}[\![s_1]\!]\rho(\mathfrak{C}[\![s_2]\!]\rho\theta)$
$\mathfrak{C}[\![\textbf{goto } \ell]\!]\rho\theta = \rho(\ell)$
$\mathfrak{C}[\![\ell{:}s]\!]\rho\theta = \mathfrak{C}[\![s]\!]\rho\theta$
$\mathfrak{C}[\![\text{other}]\!]\rho\theta = \ldots$
$\mathfrak{J}[\![s_1;s_2]\!] = \mathfrak{J}[\![s_1]\!]\ \mathit{app}\ \mathfrak{J}[\![s_2]\!]$
$\mathfrak{J}[\![\textbf{goto } \ell]\!] = \langle\rangle$
$\mathfrak{J}[\![\ell{:}s]\!] = \ell\ \mathit{cat}\ \mathfrak{J}[\![s]\!]$
$\mathfrak{J}[\![\text{other}]\!] = \ldots$
$\mathcal{M}[\![s_1;s_2]\!]\rho\theta = \mathcal{M}[\![s_1]\!]\rho(\mathfrak{C}[\![s_2]\!]\rho\theta)\ \mathit{app}\ \mathcal{M}[\![s_2]\!]\rho\theta$
$\mathcal{M}[\![\textbf{goto } \ell]\!]\rho\theta = \langle\rangle$
$\mathcal{M}[\![\ell{:}s]\!]\rho\theta = \mathfrak{C}[\![s]\!]\rho\theta\ \mathit{cat}\ \mathcal{M}[\![s]\!]\rho\theta$
$\mathcal{M}[\![\text{other}]\!]\rho\theta = \ldots$

<u>denotational model of GL</u>

If we denote by Σ the signature corresponding to the abstract syntax, then our aim is to derive a semantic Σ-algebra S_Σ based on the denotational model given above. Simply by abstracting away from the syntactic sorts and "target tupling" where there is more than one semantic function on the same syntactic sort we get the following definiton of S_Σ.

<u>Carriers</u>
$S_{\text{Program}} = U \to C \to C$
$S_{\text{Stmt}} = [U \to C \to C] \times \text{Ide}^* \times [U \to C \to C^*]$

<u>Operators</u>
$\text{prog}_S(s) = \lambda\rho\theta.(s\downarrow 1)(\mathit{fix}(\lambda\rho'.(\lambda i^*\lambda\theta^*.\rho[i^*/\theta^*])(s\downarrow 2)((s\downarrow 3)\rho'\theta)))\theta$
$\text{semi}_S(s_1,s_2) = \langle\lambda\rho\theta.(s_1\downarrow 1)\rho((s_2\downarrow 1)\rho\theta),$
$\qquad s_1\downarrow 2\ \mathit{app}\ s_2\downarrow 2,$
$\qquad \lambda\rho\theta.(s_1\downarrow 3)\rho((s_2\downarrow 1)\rho\theta)\ \mathit{app}\ (s_2\downarrow 3)\rho\theta\rangle$
$\text{goto}_S(\ell) = \langle\lambda\rho\theta.\rho(\ell), \langle\rangle, \lambda\rho\theta.\langle\rangle\rangle$
$\text{labelled}_S(\ell,s) = \langle s\downarrow 1, \ell\ \mathit{cat}\ s\downarrow 2, \lambda\rho\theta.((s\downarrow 1)\rho\theta\ \mathit{cat}\ (s\downarrow 3)\rho\theta)\rangle$
$\text{other}_S = \ldots$

<u>semantic Σ-algebra based on denotational model of GL</u>

Though the derivation of S_Σ may seem to have been contrived (even magical) a full and rigorous description of a suitable general technique is given in §3.2.3. It is clear from this example that to ensure initial algebra semantics can be derived from semantic models in an orderly way we need a principle similar to referential transparency for denotational semantics. Mosses(1983) defined a set of *homomorphic semantic equations* for a given (syntactic) signature Σ as follows: for each operator $u \in \Sigma_{s1 \ldots sn,s}$ there is one equation of the form $\mathcal{F}_s(u(x_1,\ldots,x_n)) = t'(\mathcal{F}_{s1}(x_1),\ldots,\mathcal{F}_{sn}(s_n))$ where $t'(x'_1,\ldots,x'_n)$ is an appropriate term of $(T_{\Sigma'}(X'))_{s'}$. Briefly Σ' is a semantic signature wherein for each s of Σ we may find a corresponding s'. (Full details are given in the original paper). While this concept may be suitable for Mosses' "abstract semantic algebra" specifications and it clearly ensures that all the semantic functions $\mathcal{F}_{s'}$ so defined are indeed homomorphic, it needs to be generalized somewhat to deal with our less restricted style of semantic presentation.

We may identify the semantic operators of some signature Ω as all operators $\mathcal{F} \in \Omega_{w,s}$ where s is not a syntactic sort and w includes at most one syntactic sort. Thus we avoid the operators defining the abstract syntax and the primitive operators on the semantic sorts. It is only the semantic operators that we wish to be homomorphic so we give the following sufficient conditions applying to equations involving semantic operators: all syntactic elements (variables, constants or expressions) that occur on the right hand side of the equation also occur on the left hand side; and all syntactic elements that occur in equations involving both syntax and semantics occur only as arguments to the semantic operators. This requirement closely parallels the denotational semantics referential transparency notion.

It is interesting to note that the equational specifications of Wand (1980b) meet our requirements and therefore characterise an initial algebra semantics, yet Mosses (1983) contrasts that approach with his own by claiming "... (Wand's) 'semantic functions' were non-homomorphic, they were just operators of abstract data types that combined syntax and semantics." It seems a little unreasonable to accept indirect definitions of homomorphisms in denotational semantics while rejecting them in Wand's algebraically founded specifications.

The rather minor restrictions we place on the form of semantic presentations by enforcing the conditions given above create no real difficulties. Indeed, considering that workers in denotational semantics largely satisfied these requirements without necessarily knowing (or caring) that they are describing a homomorphism, the two conditions above can be seen as guides rather than restrictions.

3.2.3 Deriving Initial Algebra Semantics from Semantic Models

In the preceding section we derived the initial algebra semantics characterised by a particular model of a semantic presentation without giving any real hint of how it was done. Our aim here is to formalize the technique in an algebraic framework giving both an algorithmic and a category-theoretic formulation.

As a starting point we note that the basic techniques involved are abstraction away from syntactic sorts and target tupling. Now a parallel exists in work on algebraic data type specifications: the so-called *final data type extension* of Wand (1979) and Kamin (1980,1983). Basically the aim is given a Σ-algebra A, we wish to derive the *most abstract* algebra that extends A with another sort and some operators on that sort. Here we outline the method for achieving this described in Kamin (1983) and refer the reader to that source for more details.

Suppose that we have a Σ-algebra A and we wish to extend it by adding a sort N and a number of operators all involving N. First we identify a subset of those operators that we believe will sufficiently distinguish among elements of the new sort. Clearly all will refer to N in their arities. Suppose this distinguishing set is

$$
\begin{aligned}
\{\ & f_1 : N \times A_1 \times \ldots \times A_{n1} \to N, \\
& \ldots \\
& f_m : N \times B_1 \times \ldots \times B_{nm} \to N, \\
& g_1 : N \times C_1 \times \ldots \times C_{j1} \to E_1, \\
& \ldots \\
& g_k : N \times D_1 \times \ldots \times D_{jk} \to E_k\ \},
\end{aligned}
$$

where all of the E's, A's, B's, C's and D's are sorts of Σ, $k > 1$ and $m > 0$ (in fact for our specific case m will always be zero). Then the most abstract representation of N is

$$\begin{aligned} N = \ & [A_1 \times ... \times A_{n1} \rightarrow N] \times \\ & ... \times \\ & [B_1 \times ... \times B_{nm} \rightarrow N] \times \\ & [C_1 \times ... \times C_{j1} \rightarrow E_1] \times \\ & ... \times \\ & [D_1 \times ... \times D_{jk} \rightarrow E_k]. \end{aligned}$$

We then proceed to define (in whichever way is considered appropriate) the operators on N as functions over the abstract representation. This gives the *final* Σ_N-*extension* of A where Σ_N is the signature denoting the sort N and the new operators on N that are to be added to A.

A hint of the direction we are heading may be taken by imagining N as a syntactic sort. To construct its abstract representation we abstract away from N and tuple those domains if there is more than one operator on that sort. Within certain bounds this is just the process we are aiming for. Suppose we are given a semantic presentation $\langle\Omega,E\rangle$. Then the signature can be divided into $\Sigma + \Delta + \Phi$ where Σ corresponds to the abstract syntax, Δ corresponds to the semantic domains and auxiliary functions and Φ corresponds to the semantic functions. Clearly Σ and Δ are quite discrete and Φ will include no new sorts. For some model of the $\langle\Omega,E\rangle$-theory, say M_Ω, to derive the corresponding initial algebra semantics we need to construct a Σ-algebra S_Σ based on the semantic carriers of M_Ω.

To begin we take the Δ-reduct of M_Ω, M_Δ. Put simply, to derive M_Δ from M_Ω we merely "forget" about the $\Sigma+\Phi$ parts of M_Ω, so M_Δ consists only of the semantic domains and the operators on them. If we now construct the final $(\Sigma+\Phi)$-extension of M_Δ we get a different Ω-algebra, say S_Ω in which the carriers of the *syntactic* sorts are precisely the abstract domains we have been seeking. Thus, the Σ-reduct of S_Ω is the required semantic Σ-algebra S_Σ. While this may at first seem rather obscure and perhaps over-complicated, in practice it is quite straightforward and we now demonstrate the technique using the example introduced in §3.2.2.

The signature of the GL presentation (call it Ω) can easily be divided into the syntax Σ consisting of sorts Program and Stmt and the operators listed under those heading; the semantic domains Δ consisting of all other sorts and the operators listed under their headings; and the semantic functions consisting of operators *P*, *C*, *L*, *I* and *M* and all sorts of Ω. If we call the given denotational model of the GL-theory D_Ω, then the Δ-reduct of D_Ω has carriers U, C, Ide* and C* (as defined) for sorts Env, C, Idlist and Clist respectively with appropriately defined functions for each operator symbol. Now to construct the final $(\Sigma + \Phi)$-extension of D_Δ, which we will denote S_Ω, we first identify the operators that will sufficiently distinguish between elements of the new sort. This will *always* be the operators of Φ, the semantic functions. (In fact, Kamin's technique only deals with adding one sort at a time. For the sake of brevity we will work by adding the two sorts Program and Stmt in parallel. There is no problem here but in general one may need to be a little more careful, especially when several of the semantic functions interact.) Thus the distinguishing set for sort Program is

$\{ \mathcal{P}\text{: Program} \times \text{Env} \times \text{C} \to \text{C} \}$

and for sort Stmt it is

$\{ \mathcal{C}\text{: Stmt} \times \text{Env} \times \text{C} \to \text{C},$
$\mathcal{I}\text{: Stmt} \to \text{Idlist}$
$\mathcal{M}\text{: Stmt} \times \text{Env} \times \text{C} \to \text{Clist} \}.$

The semantic function $\mathcal{L}$ is not included because it is directly defined in terms of $\mathcal{I}$ and $\mathcal{M}$.

Thus the carrier of sort Program in S_Ω is $[\text{Env}_D \times C_D \to C_D]$ which is $[U \times C \to C]$. Similarly the carrier of sort Stmt in S_Ω is $[\text{Env}_D \times C_D \to C_D] \times [\text{Idlist}_D] \times [\text{Env}_D \times C_D \to \text{Clist}_D]$ which is $[U \times C \to C] \times [\text{Ide}^*] \times [U \times C \to C^*]$. Though we omit most of the details here, the definitions of the functions for each of the syntactic operators (eg. prog: Stmt $\to$ Program, semi: Stmt $\times$ Stmt $\to$ Stmt) are quite straightforward and are based on the rather trivial observation that for each operator in the distinguishing set for sort n, $g_i(n,a_1,\ldots,a_{ji}) = (n{\downarrow}i)(a_1,\ldots,a_{ji})$, where $n{\downarrow}i$ is the ith projection of n. For example, given the definitions in the model D_Ω of $\mathcal{C}_D$, $\mathcal{I}_D$ and $\mathcal{M}_D$ on the class of statements "semi(s1,s2)"

$\mathcal{C}_D(\text{semi}(s1,s2),\rho,\theta) = \mathcal{C}_D(s1,\rho,\mathcal{C}(s2,\rho,\theta))$
$\mathcal{I}_D(\text{semi}(s1,s2)) = \mathcal{I}_D(s1)$ *app* $\mathcal{I}_D(s2)$
$\mathcal{M}_D(\text{semi}(s1,s2),\rho,\theta) = \mathcal{M}_D(s1,\rho,\ \mathcal{C}_D(s2,\rho,\theta))$ *app* $\mathcal{M}_D(s2,\rho,\theta)$

and the information that

$\mathcal{C}(\text{semi}(s1,s2),\rho,\theta) = (\text{semi}(s1,s2){\downarrow}1)(\rho,\theta),$
$\mathcal{I}(\text{semi}(s1,s2)) = (\text{semi}(s1,s2){\downarrow}2)()$ and
$\mathcal{M}(\text{semi}(s1,s2),\rho,\theta) = (\text{semi}(s1,s2){\downarrow}3)(\rho,\theta),$

we may immediately deduce

$$\begin{aligned}\mathrm{semi}_S(s1,s2) = \langle & \lambda\rho\theta.s1{\downarrow}1(\rho,s2{\downarrow}1(\rho,\theta)),\\ & s1{\downarrow}2 \ \textit{app}\ s2{\downarrow}2,\\ & \lambda\rho\theta.s1{\downarrow}3(\rho,s2{\downarrow}1(\rho,\theta)) \ \textit{app}\ s2{\downarrow}3(\rho,\theta)\rangle.\end{aligned}$$

Finally to derive S_Σ, the semantic algebra we require for the initial algebra semantics, we take the Σ-reduct of S_Ω, simply forgetting all of S_Ω except the syntactic sorts and operators.

The above technique provides a "recipe" for deriving the initial algebra semantics characterized by one of our semantic models, however a more concise non-algorithmic formulation is possible. Again presume we have a semantic presentation P whose signature Ω can be divided as above into $\Sigma + \Delta + \Phi$ and a model of P called M_Ω. Then the semantic Σ-algebra S_Σ we require is the Σ-reduct of S_Ω, the final object in the subcategory of Alg_P consisting of only those models that are *relatively prime* to M_Δ, the Δ-reduct of M_Ω. A P-algebra A_Ω is prime relative to M_Δ if and only if the Δ-reduct of A_Ω is isomorphic to M_Δ and the unique homorphism $h{:}T_P \rightarrow A_\Omega$ is surjective on the carriers of Ω- Δ (ie.Σ).

The concept of *prime relative to X* is a relaxation of Kamin's (1983) insistence on *prime* algebras, that is algebras such that the unique homomorphism to them is onto. This does not suit our purposes since we do not wish to be restricted to semantic domains with no "junk" values since, for example, the domain $V \cong B + [V \rightarrow V]$ used in the denotational model of the lambda calculus presentation (§3.1.2) would not then be permissible. We treat this question in detail in the next section.

It is worth pointing out that this formulation of the relation between initial algebra semantics and our semantic models is not particularly interesting from an algebraic viewpoint nor is it exactly perspicuous. However, the algebra does provide a convenient setting for what is at least a rigorous and compact definition.

3.3 Acceptable Semantic Models

As was briefly mentioned in S3.1.1 not every model of a presentation necessarily provides a satisfactory semantics for the language being described. In other words, given some presentation P we may wish to specify a sub-class of the algebras constituting Algp as those which are acceptable semantic models. While we will never need to do so in this dissertation, we treat the problem here since it has occasionally been touched on in the literature and seems to have led to some confusion. For example, Wand (1979) goes no further than to say that the class of acceptable models will be *some* subcategory of Algp, while by far the most common approach (at least in the abstract data type literature) is to ignore the question completely, using the specification technique without ever saying exactly *what* (mathematical) object is being specified, rather relying on the intuitions of the reader. This section is largely taken from Baker-Finch (1984a).

3.3.1 A Solution for Abstract Data Types

Probably the first suggestion of a satisfactory semantics of the signature plus equations technique for abstract data type specification was the initial algebra approach clearly described in Goguen, Thatcher & Wagner (1978). There, given some presentation P, the particular object being specified is the (isomorphism class of the) initial algebra in Algp. This algebra has some attractive properties, not the least of which is the fact that it is easy to construct. Further, the popularity of the choice of the initial algebra as *the* object represented by a presentation is explained by Wand (1979) as follows: "First, its universe contains no values other than those required by the generators. Second, two values have the same semantics in the initial T-algebra if and only if they have the same semantics in every T-algebra. Thus no information is lost except that which is required by the relations". (For "T-algebra" read "algebra in Algp"). Burstall and Goguen (1982) put it somewhat more snappily: "... the initial algebra ... has *no junk*: every element of the carrier is the value of some term; *no confusion*: different terms get different values". However, if we choose this algebra to be the *only* model of the presentation, we are wasting the considerable power of the specification technique. After all, if we only want to describe a single algebra there are numerous more direct and simple methods. Further, the initial model may not necessarily be the intended or most intuitive one. Consider for example the following typical presentation for (unbounded) arrays of natural numbers.

```
sort Nat
        zero: → Nat
        succ: Nat → Nat
sort Array
        empty: → Array
        assign: Array × Nat × Nat → Array
        access: Array × Nat → Nat

Equations
1.      access(empty,i) = zero
2.      access(assign(a,j,n),i) = if i=j then n else access (a,i).
```

In the initial model it is *not* the case that assign(empty,1,5) and assign(assign(empty,1,9),1,5) are the same array. An entire history of assignments to each element of the array is maintained and this does not fit the generally accepted array concept.

Perhaps the most popular current view is that an abstract data type specification (being some presentation P) represents the algebraic theory Th_P. If we settle on this choice however, we are begging the question. The theory has a class of models that is isomorphic to Alg_P. So if we accept the algebraic theory approach, we are left with exactly the same question we must answer with respect to the equational variety: "which of the algebras are satisfactory semantics?" It is important to point out that we do not therefore reject the view that an abstract data type specification presents an algebraic theory; we are only saying that the question remains irrespective of such a choice.

A refinement of the algebraic theory approach has been put forward by Burstall & Goguen (1979) and Reichel (1980) by developing a notion similar to an algebraic theory which only has initial models. The writer of an algebraic presentation is then able to distinguish some sorts as being subject to initial interpretation. The usefulness of this approach can be displayed using our array-of-naturals example. If we can somehow insist that the Nat part of the theory is to be interpreted initially, then any model that satisfies such a restriction will satisfy our intuitive concept of such a data type. Burstall and Goguen's "data theories" and Reichel's "canons" or "initially restricting algebraic theories" provide the means by which to give a rigorous meaning to the notion outlined above. Unfortunately our lambda calculus presentation LC of §3.1.1 is not amenable to this treatment. Clearly we do not wish to be restricted to models where Abstr or W are interpreted initially since this would exclude the denotational model described in §3.1.2, but to leave Abstr and W unrestricted would admit undesirable models such as the degenerate one with single point carriers. Thus, while initially restricting algebraic theories appear to satisfactorily provide a solution for specifications of data types they do not meet the requirements for our more ambitious use of equational presentations.

3.3.2 Allowing Junk

The precise question we are considering is: "which of the algebras in Algp, for some presentation P = ‹Σ,E›, are acceptable semantics of the concept we are trying to describe?". Note that it is quite obvious that the class of such acceptable models cannot be identified solely by innate properties. It must be left up to the writer of the specifications to somehow state which are acceptable, but we can defer this consideration for now.

Clearly, the initial model will always be an acceptable model since the only objects that are equated are those so specified in the presentation. If it is not, then there can be *no* acceptable models and hence the presentation must be inadequate. It follows from the fact that there is a unique homomorphism from Initp to every other algebra in Algp that all these other algebras have "junk" or "confusion" or both. We examine how much junk and confusion is acceptable.

Firstly, we should not be concerned if the carriers contain elements that are not the value of any term. After all, this is frequently the case in denotational semantics. For example, the domain $V \cong B + [V \to V]$ used in the lambda calculus model of §3.1.2 contains many elements never reached by any semantic valuation; in particular, all the transcendental functions. The only query about allowing unlimited junk is whether we want to allow it in the syntactic carriers. If we do, then the algebra describes the "semantics" of various objects beyond the terms generated by the abstract syntax. However, if we take the view that the semantics is meant to give the meaning of a term of the language with which it is presented (i.e. the syntax is described elsewhere) then we can happily allow nonsense values in the syntactic carriers. Though this may be a satisfactory situation for the purposes of specification alone, the uses to which we will be putting the semantic presentations later in this dissertation will require us to insist that the syntactic carriers exactly reflect the language being dealt with. We will have more to say on this issue in §3.3.5, but for now we offer formalized statements of both possibilities.

An algebra A in Alg_P is an acceptable model in the sense outlined above if the unique homomorphism h: $Init_P \to A$ is a monomorphism (ie. one-one). This effectively ensures that all the elements of the carriers of the initial algebra (the terms) are separately represented in A. To disallow junk in the syntactic carriers we make the following further restriction h: $Init_P \to A$ consists of $\langle h_{s1},...,h_{sn}\rangle$ and $\{si|i \in I\}$ are the syntactic sorts. Then as well as h being a monomorphism, the h_{si} for $i \in I$ must be bijections. This extra restriction is such a minor point and is so easily catered for that we shall overlook it for the remainder of this discussion.

3.3.3 Limiting Confusion

Clearly the above restriction is unsatisfactory since it again disallows our denotational model of LC, and serves only to demonstrate that it is the limitation of confusion that must be our objective. While it is possible to give a simple general statement about junk in acceptable models, the degree of confusion allowed must be the choice of the person writing the specification. For instance, again using our lambda calculus example, there can be no *a priori* reason to think that λx.x and λy.y may be given the same meaning, or that bind(bind(arid,x,a),y,b) and bind(bind(arid,y,b),x,a) could be evaluated to equivalent representations. Thus if we wish to identify all the acceptable models we are bound to somehow specify which terms may be equated and which ones may not. One way to do this is to pick a particular algebra as the "maximally-confused" model (for want of a better name). The model is chosen on the basis that as many terms as we find acceptable to equate are equated. If we can then define a relationship between this and other algebras that have *no other* confusion, we have a way of identifying the class of such acceptable models.

For guidance in our choice of this "maximally-confused" model we are attracted by the tenets of denotational semantics (Scott and Strachey, 1971). As well, the notion of *fully abstract* models (Milner, 1977) seems related. It is reasonable to say that one of the aims of (standard) denotational semantics is to make the semantic description as abstract as possible and the concept of "fully abstract" serves to formalize this aim. For our purposes we interpret this to mean that the aim is not to differentiate between the meanings of two programming language constructs that are equivalent in the view of the semantics writer or language designer. So perhaps a denotational style of model would be a good choice as the acceptable model that gives the same meaning to as many terms as possible. We need to be a little careful here - it would be quite valid to represent an environment as a list, say $[\mathrm{Ide} \times V]^n$ which is not as abstract as $[\mathrm{Ide} \to V]$. In such a domain the two environment terms given above do not evaluate to the same element of the carrier yet none of the interrogation operators can differentiate between the two elements. Thus it seems that final algebra extensions (Kamin, 1983) may also have something to say on the choice of a maximally-confused model.

Let us look more closely at the concept of full abstraction introduced by Milner (1977). Put briefly, a semantic description is fully abstract provided two phrases (pieces of abstract syntax) are given the same meaning if and only if their substitution into the same program context (a program with a "hole" in it) always gives two programs with identical meanings. A formal algebraic definition of the concept is given in Goguen and Meseguer (1983). Thus, the effect of full abstraction is to assign different meanings to syntactic constructs only when necessary. This is exactly what we want for our maximally-confused models and is, to a degree, what the final algebra construction provides.

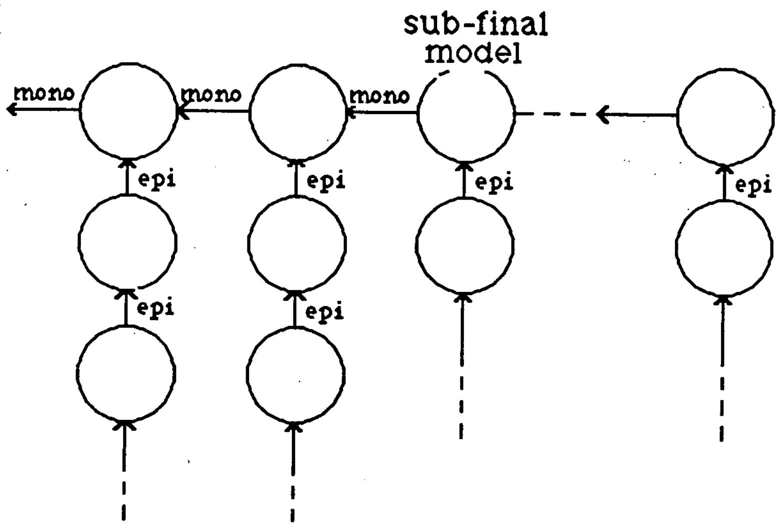
However, there is still some work to be done. The final algebra construction depends on the pre-existence of some primitive types (otherwise we would always get the algebra with single-point carriers). Similarly, full abstraction is defined with respect to the semantic domain of whole program meanings. Thus in that domain we may assign different meanings to what are intuitively equivalent whole programs (eg. $\lambda x.x$ and $\lambda y.y$) and still have a fully abstract model. (The lambda calculus is not in fact a particularly good example since whole programs are also program phrases). Hence we require more of a maximally-confused model than its being fully abstract. Rather, it is a fully abstract model with domain of program meanings X where X is chosen to identify as many programs as we consider appropriate. Having chosen such an X, Kamin's (1983) final algebra extension construction can be of assistance in deriving the appropriate other semantic domains. Unfortunately it is not quite as straightforward as we may desire since the final algebra extension depends also on the functionality of the operators; in this case particularly the semantic functions. Consider the (rather contrived) case where we have $\mathcal{P}$: Program → C and $\mathcal{C}$: Stmt → [Ide × C] where the irrelevant information of (say) the first identifier occurring in the statement is included in its semantic domain. Clearly this is not fully abstract but the final algebra construction does not preclude such a possibility. Thus we need to make a careful choice of a program meaning domain *and* semantic functions as a minimum basis for identifying a maximally-confused model.

Returning to our lambda calculus example, suppose we choose (or derive) the denotational model Den as our model that identifies as many terms as possible while still remaining acceptable. How then may we use this choice to decide which other models are appropriate? As a first approximation, let us say that algebra A in Alg_{LC} is an appropriate model if there is a Σ-homomorphism h: A→ Den. The effect of this limitation is to ensure that the terms are equated in A only if they are equated in Den. Thus we are saying that only the kind of "confusion" that occurs in Den is allowed in our "appropriate models". This is clear from the definition of Σ-homomorphism. Thus in one fell swoop we are disallowing undesirable evaulation of different terms to the same value and as a consequence ensuring that solutions of recursive equations are always like least fixed-points in the sense that a non-terminating computation will never be given a "sensible" value.

So far in this section we have not considered the allowance of extra elements in the carriers of the algebras. So the problem remains that given some A such that h: A → Den is a homomorphism (i.e. A is an acceptable model), there may be an LC-algebra B such that there is a monomorphism k: A → B. In other words B differs from A only in that it includes "junk" in its carriers. However, it is likely that there is no homomorphism B → Den. From our discussion in the previous section we would like algebras such as B to be included in our class of acceptable models. One last refinement leads to the following definition wherein we finally abandon the loose concepts "acceptable model" and "maximally-confused model".

Given a presentation P = ⟨Σ,E⟩, an algebra A in Alg_P is an ***ok-model*** of Th_P if there is a Σ-homomorphism h: A → J where J is some P-algebra such that there is a monomorphism k: D → J where D is the P-algebra distinguished as the ***sub-final model*** of Th_P.

Note that the carriers of the sub-final model can have extra elements. For example V in Den has elements that are not in the image of Val. In such cases there is clearly a model, say M, with no such extra elements and a monomorphism from that model to the chosen sub-final model, say D. Thus if there is a homomorphism A → M there must be one A → D by composition. The following diagram showing ***some*** of the morphisms(the epi-mono factorizations, Arbib and Manes, 1975) may be enlightening. The monomorphisms (one-one) indicate "same confusion, more junk"; the epimorphisms (onto) indicate "same junk, more confusion".



Note that this is really a very loose representation (for example, it suggests that the class of ok-models is always countable) but it is only intended as an aid to intuition.

3.3.4 The Category of Acceptable Models

It is interesting to consider what sort of structure the class of ok-models forms. Clearly, they form a full subcategory (call it "OK") of Alg_P since all the morphisms between objects in the subcategory are retained. It is only the ***existence*** of certain morphisms that we use to choose the objects. However, the ok-models do not form an equational variety. By the Birkhoff Variety Theorem (in Manes, 1976), if V is a variety and A is in V then all of A's quotient algebras are in V. (B is a quotient of A if there is an epimorphism A → B). Now the degenerate P-algebra (the one whose carriers are all singleton sets) is a quotient of every ok-model but it is not itself in OK, in our example at least, since there is no homomorphism Deg → D and no monomorphism D → Deg.

This has relevance to the view that these specifications represent algebraic theories. As pointed out earlier, the category of models of Thp is isomorphic to the variety of P-algebras. It is clear therefore that it is *impossible* to present a theory all of whose models are ok-models. This fact appears to cut across Wand's (1980b) hope that "such restrictions (on the class of acceptable models) seem to depend only on the theory and not on the specification".

To summarize, it has been argued that none of the approaches to the semantics of algebraic data type specifications extend satisfactorily to our use of algebraic presentations for describing programming language. A possible solution is to extend the specification to consist of a signature, some equations *and* a description of a particular, suitably abstract algebra as the sub-final model. For the applications dealt with in this thesis such restrictions on the class of models never come into play since all our proofs are either general enough to apply to *all* models, or we deal with a single specific model. However, it is necessary or at least desirable that when using equational semantic presentations to *prescribe* the semantics of a programming language the class of acceptable models be clearly delineated.

3.3.5 Freely Interpreting Syntactic Signatures

As mentioned briefly in §3.3.2, the work of later chapters from time to time requires that there be no junk in the carriers of the syntactic sorts. Basically, without this restriction structural induction over the language becomes invalid and the unnecessary imposition of such a limitation on the available proof techniques is unwarranted.

At the level of the algebras or models of a theory we can easily formalise this restriction by the qualification that the ok-models must be *reachable* on the syntactic sorts (Sanella & Wirsing, 1983). If we denote the set of syntactic sorts of some presentation P by S, then a P-algebra A is reachable on S provided $h: T_P \to A$ is such that h_S is onto for all $s \in S$. With such a condition being satisfied we may clearly use structural induction over the language within any particular ok-model. However, we will need to use induction at the level of the *theory* rather than the models so the restriction we require will need to be on the theory rather than the models of that theory. Thus we need the concept of a theory whose only models are initial ones (which can be extended to a theory whose models are "partly initial").

As a brief example consider the following presentation for natural numbers with an equality operator.

```
sort Bool
        tt: → Bool
        ff: → Bool
sort Nat
        zero: → Nat
        succ: → Nat
        eq: Nat x Nat → Bool
Equations
        eq(zero,zero) = tt
        eq(zero,succ(n)) = ff
        eq(succ(n), zero) = ff
        eq(succ(m),succ(n)) = eq(m,n)
```

Now while the theory of this presentation has equations such as eq(succ(zero), succ(zero)) = tt, eq(succ(succ(0)), succ(succ(0))) = tt and so on, it does not include the equation eq(n,n) = tt. This is simply because there is a model of the theory whose carrier for Nat has some element, say Q, that is not the value of any term (i.e. it is junk) and eq(Q,Q) = ff. Thus, by our definition of algebraic theory in §2.3, eq (n,n) = tt is excluded even though we would intuitively wish it to be true. If we could somehow constrain the theory so that Nat was freely interpreted (i.e. only the initial model is allowed) then we could be sure that the carriers for Nat were always in bijective correspondence with the set {zero, succ(zero), succ(succ(zero)) ...} so that no such Q could exist and we could apply induction over Nat to establish such properties as eq(n,n) = tt not given by the rules of equational inference alone.

Fortunately a mechanism for expressing such restrictions is already available and was mentioned briefly in §3.3.1. It is the "initially-restricting algebraic theories" of Reichel (1980) or the "data theories" of Clear (Burstall & Goguen, 1979). Indeed the term "induce" was used in earlier versions of Clear (Burstall & Goguen, 1977) which more directly suggested the effect of such restrictions. We consider a long and technical discussion of these new theories to be inappropriate at this point and refer the reader to Burstall & Goguen (1979) and Reichel (1980).

The most important aspect to us is that we may insist that some sorts must be freely interpreted in all models without being reduced to consider a sub-class of the *models* of the theory; the restriction is on the *theory* itself. Thus we may mark all of the syntactic sorts of our presentations to be freely interpreted without relinquishing our claim that we are presenting algebraic theories. To this end, in all presentations from this point on we will distinguish those sorts that are subject to initial interpetation by denoting them as *syntactic sorts* as for example:

```
syntactic sort Lambda.
      const: B → Lambda
      ...
```

It will always be exactly those sorts representing the abstract syntax that will be so marked. The first point at which we will actually need such constraints on our semantic theories will be in §4.4 and we will continue the discussion there when faced with a realistic example.

Chapter 4
Congruence of Semantic Models

The notion of semantic congruence is an important one in the denotational semantics literature, especially in relation to proofs of correctness of compilers or interpreters. In this chapter we intend to develop the concept against the algebraic background developed so far and attempt to pin down exactly what constitutes a semantic congruence, a subject discussed only in loose, general terms in the denotational semantics literature.

Our approach will be to consider a sequence of several simple examples using them to clarify our intuitive notion of congruence for each of them, with a view to evolving a formal algebraic definition of semantic congruence. Having done that, we show that the clarified definition and its algebraic foundation combine to simplify many (though not all) proofs of congruence that appear in the literature. Indeed, it makes *possible* some proofs that cannot be realised by the traditional approach . As well, we feel that our way of expressing the congruence is more direct and therefore more appealing than the predicates generally used in denotational semantics.

4.1 Algebraic View of Semantic Congruence

An important question in the study of programming language semantics is whether or not two semantic definitions are congruent; that is, do they describe (semantically) the same language. This is particularly relevant in that an established technique for demonstrating the correctness of a translation involves a hierarchy of semantic models and the establishment of such a relation between the consecutive models. Since our work here is characteristically algebraic, it is natural for us to seek an algebraic formulation of the concept of semantic congruence. Further, we consider the fixing of a *formal notion* of congruence, algebraic or otherwise, to be a valuable goal in its own right.

4.1.1. The Intuitive Concept

Simply put, we will consider two semantic models to be congruent provided they give equivalent meanings to the same language. Having stated that, it quickly becomes apparent that "equivalent meaning" requires some clarification, especially when these meanings may be values from quite different domains. Little help or guidance can be found in the denotational semantics literature other than loose generalizations. The most that is offered by Stoy (1977) is the following:

> "The exact details of such (congruence) conditions depend on the details of the definitions being compared, but the general idea is that two definitions are congruent if it can reasonably be claimed that they are defining the same language."

A more detailed discussion occurs in Milne & Strachey (1976) though it only really applies to the example language Sal around which the entire book revolves. Even so, apart from stating that if a program does not terminate under one of the semantic definitions then it must not terminate under the other, no guidelines are offered. Thus there is nothing to prevent us putting forward quite pathological conditions and claiming that they are a statement of congruence. A rather extreme example would be to claim that two definitions that always return *different* answers for identical programs are therefore congruent. So although we have a fairly clear perception of what we require of a congruence we have absolutely no formal basis on which to work.

4.1.2 Lambda Calculus Congruence Statement

The first example we consider is the lambda calculus, partly because of its smallness and partly because we have already discussed a semantic presentation LC and a (denotational) semantic model, Den. We wish to consider the congruence of Den with a particular operational semantics of the lambda calculus. The full details of the semantic definition are given at the beginning of §4.2 rather than here since it seems more desirable to have the formal definition close to the proof for easy reference at that stage. In the operational semantics, we have the same domain B of basic values, but FUN is replaced by a new domain CLO (for "closures": Landin, 1964) of triples that gather together the information required to represent the meaning of an abstraction (i.e. its body, its bound variable and an environment). Similarly ENV is replaced by the domain E of lists of identifier, value pairs. A function *apply* interprets closures, mimicking the direct application of members of FUN to arguments. The new semantic function is Eval: Lambda $\times$ E $\rightarrow$ [B + CLO] and the semantic equations are very similar to these in the denotational definition.

The best intuitive notion of congruence betwen those two semantic definitions can obviously be given in terms of two functions: one embedding U into V and the other relating the finite environments of E to elements ENV.

$$\text{Value}(u) = \begin{cases} u \textit{ in } V & \text{if } u \in B \\ \lambda a.\text{Val}[\![\eta]\!]\,\text{EN}(e)[x/a] \textit{ in } V & \text{if } u = \langle \eta, x, e\rangle \in \text{CLO} \end{cases}$$

$$\text{EN}(e) = \lambda x.\text{Value}(\text{Lookup}(x,e))$$

The function Value sends closures to the functions they encode while not affecting basic values. EN constructs a function which looks up the value of an identifier in e and embeds this value in V. Using these functions the intuitive congruence is the following:

For each $\phi \in$ Lambda,

$$\text{Value}(\text{Eval}[\![\phi]\!]\, e) = \text{Val}[\![\phi]\!]\, \text{EN}(e)$$

Two points need to be made here. First, the simplicity of the relation is largely due to the fact that the two models "look the same" in a sense, only really differing with respect to the domains involved. Second, despite the apparent simplicity this congruence *cannot* be established by the traditional techniques of denotational semantics (Turner, 1979). This problem, though it looms rather large, will be overlooked for now since we are presently concerned only with what *constitutes* a congruence and the issue of its provability is deferred to §4.2.

In §3.1.2 the point was made that we can consider denotational definitions as algebras and indeed we showed Den to be a model of the lambda calculus presentation LC. By the same argument, the operational definition we are presently discussing is also an algebra (we shall call it Op) and will also be shown in §4.2.3 to be a model of LC using a style of proof identical to that used in §3.1.2. Thus, if we assume (for now) that Op and Den have the same signature Σ, then the congruence we wish to establish is exactly a *Σ-homomorphism* provided we add identity maps for B and Lambda. By definition, if h: Op → Den is a Σ-homomorphism consisting in part of h_U: U → V and h_E: E → ENV then it must satisfy h_U(Eval⟦φ⟧ e) = Val⟦φ⟧ h_E(e) which is exactly the condition required of Value and EN.

While it may seem that there are still many loose ends we make the claim that for this example at least, an algebra homomorphism correctly constitutes a congruence.

4.1.3 Addition Expression Congruence Statement

It is not at all difficult to imagine a congruence relation that does not fit into the rather narrow definition proposed above and it is an example of such that we wish to explore here. The language consists simply of numerals and a plus sign (bold to distinguish it from the addition operator) and the semantics we will consider are the usual direct semantics and stack semantics. The definitions are brief enough to give both here and in §4.4.1.

Syntax
<Exp>: = <**N**> | <Exp> **+** <Exp> (again we will blur the Numeral/ Number distinction)

Domain
n : **N** (natural numbers)

Semantic Function
$\mathcal{E}_D$: Exp → **N**

(D1) $\mathcal{E}_D[\![N]\!] = n$
(D2) $\mathcal{E}_D[\![e_1 + e_2]\!] = \mathcal{E}_D[\![e_1]\!] + \mathcal{E}_D[\![e_2]\!]$

<u>direct semantics of addition expressions</u>

<u>Syntax</u>
<Exp>: = <N> | <Exp> + <Exp>
<u>Domains</u>

$\mathbb{N}$	(natural numbers)
$\mathbb{N}^*$	(stacks are represented by sequences of natural numbers)

<u>Semantic Function</u>
$\mathcal{E}_S$: Exp $\times\ \mathbb{N}^* \to \mathbb{N}^*$
(S1) $\mathcal{E}_S[\![n]\!]\ \zeta = n$ *cat* ζ
(S2) $\mathcal{E}_S[\![e_1 + e_2]\!]\zeta = \mathrm{add}(\mathcal{E}_S[\![e_2]\!](\mathcal{E}_S[\![e_1]\!]\zeta))$
where add $(\zeta) = (\zeta\downarrow 1 + \zeta\downarrow 2)$ *cat* (*tail*(*tail*(ζ)).

<u>stack semantics of addition expressions</u>

Thus the effect of the stack semantics is to push the "value" of the expression onto the stack. The effect of *add* is to replace the top two elements of the stack by their sum.

Again, we can consider the direct semantics and the stack semantics as many-sorted algebras (we shall refer to these algebras as Dir and Stk respectively). However, even the briefest inspection confirms that they cannot sensibly be given the same signature, so we cannot talk of their relation in terms of homomorphism. It is clear that a possible statement of congruence between the two definitions is the following:

For each e $\in$ Exp and any $\zeta \in \mathbb{N}^*$,
$\mathcal{E}_D[\![e]\!] = (\mathcal{E}_S[\![e]\!]\zeta)\downarrow 1.$

It is also clear that Dir and Stk must be models of some theories (Th_D and Th_S given in §4.4.1) so that, for example *cat* and $\_\downarrow 1$ are realisations of operators of Th_S. In this light, the congruence condition can be seen as reflecting a relation between Th_D and Th_S. In fact it is the relation between theories that conceptually corresponds to homomorphism between algebras: a theory morphism. So if we denote the theory morphism embodied in the congruence statement above by $\sigma: Th_S \rightarrow Th_D$ then what we wish to show is that Dir = U_σ(Stk) where U_σ is the derivor of σ as discussed in Chapter 2. Derivors were used directly for a very similar language by Burstall & Landin (1969). We prefer the slightly more abstract expression in terms of theory morphisms since we are working in part with theories; our semantic presentations specify algebraic theories. Further, a derivor in the sense used by Burstall & Landin is specific to a particular algebra whereas given a theory morphism $A \rightarrow B$ we can derive a model of A from any of the entire class of models of B by a single standard process.

So far we have two quite different formulations of an algebraic notion of semantic congruence: algebra homomorphism and theory morphism. In the following section we consider a more realistic example in an attempt to bring the two together.

4.1.4 DEVIL Congruence Statement

The language DEVIL was devised by Henson & Turner (1982) as a vehicle for introducing *completion semantics*, an operational semantics that they suggest should be seen as a "standard" operational version of continuation semantics. In their view, DEVIL " ... contains most of the features which force a wedge between denotational and operational definitions." Further, the domains used in the completion definition are largely based on Landin's (1964) *closures* and as such rather closely resemble the domains of the continuation definition. Despite this similarity, the proof of congruence is not at all straightforward and the best that Henson & Turner hope for is that " ... a standardisation of the operational semantics along the lines suggested here affects a corresponding standardisation of the congruence proof" and "the structure of the proof offered (in their paper) will serve as a paradigm for any such proof relating continuation and completion semantics."

The syntax of DEVIL is given below, though we have in fact altered it from the original by eliminating gotos and labels. This has the effect of shortening the definitions and the proofs without altering their complexity since the semantic domains remain virtually as they are in (Henson & Turner, 1982).

Com ::= dummy | Com; Com | Ide: = Exp | call Exp | resultis Exp |

Exp $\rightarrow$ Com, Com | **while** Exp **do** Com
Exp ::= Ide | **true** | **false** | Exp $\rightarrow$ Exp, Exp | **valof** Com | **proc** Com

Again the complete definitions are given in §4.5.1 though we repeat part of the domain definitions here to give the flavour of the completion style. First the continuation semantics:

$S = L \rightarrow [V \times T]$	stores
$U = [Ide \rightarrow D] \times K$	environments
$C = S \rightarrow S$	command continuations
$K = E \rightarrow C$	expression continuations
$F = C \rightarrow C$	function closures
$\mathcal{C}_d$: Com $\rightarrow U \rightarrow C \rightarrow C$	
$\mathcal{E}_d$: Exp $\rightarrow U \rightarrow K \rightarrow C$	

Now the completion semantics:

$S = [L \times V \times T]^*$	stores
$U = [Ide \times D]^* \times K$	environments
$C = [F \times C] + [Exp \times U \times K] + [E \times K] + \{fail\} + \{final\}$	command completions
$K = [\{update\} \times D \times C] + [\{call\} \times C] + [\{cond\} \times C \times C]$	expression completions
$F = [Com \times U]$	command closures
$\mathcal{C}_0$: Com $\rightarrow U \rightarrow C \rightarrow S \rightarrow S$	
E_0: Exp $\rightarrow U \rightarrow K \rightarrow S \rightarrow S$	

The resemblance is quite close with stores and environments in the operational semantics being the "usual" list representation of the abstract function, and command and expression completions being unions of the various types of closures that arise in the semantic clauses.

Because of this close resemblance one may expect that we could arrange for the two semantics to be models of the same theory, as is the case for our lambda calculus example. Indeed, this can easily be seen by partly expanding the functionality of $\mathcal{C}_d$ and $\mathcal{E}_d$ to $\mathcal{C}_d$: Com $\rightarrow U \rightarrow C \rightarrow S \rightarrow S$ and $\mathcal{E}_d$: Exp $\rightarrow U \rightarrow K \rightarrow S \rightarrow S$ so that there is an exact match with $\mathcal{C}_0$ and $\mathcal{E}_0$. However, for the purpose of this exercise we choose to treat them as models of different theories DA and DB where the major difference is the arity of the semantic functions; $\mathcal{C}_A$: Com × U × C × S → S, $\mathcal{E}_A$: Exp × U × K × S → S, $\mathcal{C}_B$: Com × U × C → C and $\mathcal{E}_B$: Exp × U × K → C. Under this assumption we need a theory morphism δ: DA → DB such that $\delta(\mathcal{C}_A)$ = apply.$\mathcal{C}_B$ and $\delta(\mathcal{E}_A)$ = apply.$\mathcal{E}_B$ where apply : C × S → S. However, given a continuation model Cont, and a completion model Comp, U_δ(Cont) is not Comp, as was suggested to be the case for the addition expression example since we still have a mismatch of the domains. Thus it appears that a sensible aim is to find a homomorphism between Comp and U_δ(Cont). The intuitive congruence is the following:

1. $h_U(\mathfrak{D}_0[\![d]\!]\rho\sigma) = \mathfrak{D}_d[\![d]\!]h_U(\rho)h_S(\sigma)$
2. $h_S(\mathfrak{C}_0[\![c]\!]\rho\theta\sigma) = \mathfrak{C}_d[\![c]\!]h_U(\rho)h_C(\theta)(h_S(\sigma))$
3. $h_S(\mathfrak{E}_0[\![e]\!]\rho\kappa\sigma) = \mathfrak{E}_d[\![e]\!]h_U(\rho)h_K(\kappa)(h_S(\sigma))$

where

h_U: [Ide × D]* × K → [Ide → D] × K
  $h_U(\langle\rho, \kappa\rangle) = \langle \lambda x.\ h_D(\text{lookup}\ (\rho,x)), h_K(\kappa)\rangle$
h_S: [L × V]* → [L → V]
  $h_S(\sigma) = \lambda \ell.\ \text{value}(\ell,\sigma)$
h_C: [F × C] + [Exp × U × K] + [E × K] + {fail} + {final} → [S → S]
  $h_C(\phi,\theta) = h_F(\phi)(h_C(\theta))$
  $h_C(e,\rho,\kappa) = \mathfrak{E}_d[\![e]\!]h_U(\rho)h_K(\kappa)$
  $h_C(\varepsilon,\kappa) = h_K(\kappa)(h_E(\varepsilon))$
  $h_C(\text{fail}) = \lambda\sigma.?_S$
  $h_C(\text{final}) = \lambda\sigma.\sigma$
h_K: [{update} × D × C] + [{call} × C] + [{cond} × C × C] → [E → C]
  $h_K(\text{update},\delta,\theta) = \lambda\varepsilon\sigma.\ \theta(\sigma[\delta/\varepsilon])$
  $h_K(\text{call},\theta) = \lambda\varepsilon.\ \theta(\varepsilon)$
  $h_K(\text{cond},\theta_1,\theta_2) = \lambda\nu.\ (\nu \to \theta_1 : \theta_2)$
h_F: [Com × U] → [C → C]
  $h_F(c,\rho) = \mathfrak{C}_d[\![c]\!]h_U(\rho)$

The algebraic interpretation we place on this congruence condition is that we wish to show the existence of a homomorphism embodied in the various h functions from the model Comp to a model derived from Cont. The derivor is rather obscured by details since it is extremely simple (and perhaps rather contrived) but it can be detected in equations 2 and 3 above where the right hand sides have the explicity parenthesised form $\mathfrak{C}_d[\![c]\!]\rho\theta(\sigma)$ and $\mathfrak{E}_d[\![e]\!]\rho\kappa(\sigma)$ to make quite clear that we are composing the semantic functions with "application to the store".

Our formalised algebraic notion of ***semantic congruence*** is the following:

> Given two semantic models A and B for (syntactically) the same language where A is a model of some theory Th_A and B is a model of some theory Th_B; if there exists a theory morphism $\delta: Th_A \rightarrow Th_B$ such that there is a homomorphism $h: A \rightarrow U_\delta(B)$ (or $h: U_\delta(B) \rightarrow A$) then we may say that A and B are ***congruent.***

It is quite easy to fit the simpler examples of §4.1.2 and §4.1.3 into this framework. For the lambda calculus example, since both semantics are models of the same theory, the theory morphism part of the congruence is the identity. Similarly , for the addition expression example the homomorphism is an isomorphism.

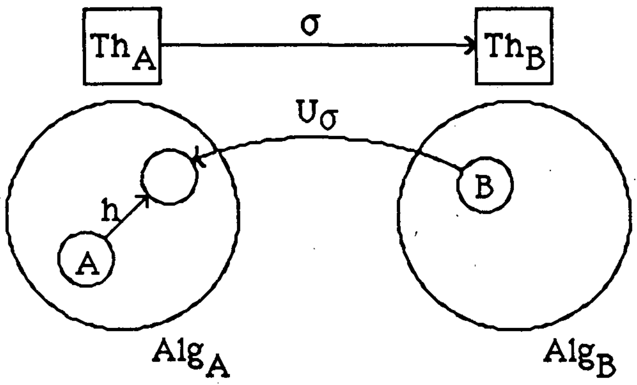
In §4.6 we will meet a complication that does not arise in the simple cases outlined so far. While at first sight it may not seem to fit the definition above it will not cause us to alter this relationship in any way.

4.1.5 Relation to Initial Algebra Semantics

In §3.2 we claimed that the initial algebra semantics approach was the most fundamental one and that other semantic styles including our own can be reduced to the simple concept of homomorphism and a semantic algebra with the same signature as the abstract syntax. Thus it is natural here to consider how our notion of semantic congruence translates into the initial algebra framework.

In outline, our technique for establishing a congruence between two semantics A and B is as follows. First present theories Th_A and Th_B of which A and B are respectively models and define an appropriate theory morphism $\sigma: Th_A \rightarrow Th_B$. Now generate $U_\sigma(B)$ and show there is a Th_A-homomorphism between A and $U_\sigma(B)$ (in either direction). A possible variation available is to present a third theory Th_C, define theory morphisms $\alpha: Th_C \rightarrow Th_A$ and $\beta: Th_C \rightarrow Th_B$, and find a Th_C homomorphism between $U_\alpha(A)$ and $U_\beta(B)$. However, this process of factoring the relation into two parts has not proved necesary in any of the examples we have ever considered and it is difficult to imagine even a most contrived case where a direct theory morphism cannot be found. Consideration of the number of proofs required to establish such a congruence (A is a model of

Th_A; B is a model of Th_B; σ is a theory morphism; and a homomorphism exists between A and $U_\sigma(B)$) may be rather daunting but they are generally very simple (especially the first three) and at least they clarify exactly what must be proven.



It should be made clear that if we are attempting to show that two semantic definitions are congruent then we need only concern ourselves with the *existence* and not the *nature* of the homomorphism. Thus unless we are interested in exactly how the two semantic definitions correspond we need not even bother writing the homomorphic relation down provided we can merely establish its existence. A rather pleasing aspect of our formulation of semantic congruences is the split into two discrete steps. The homomorphism is concerned solely with relating the semantic objects ("meaning" values) of the two definitions , while the theory morphism specifies an "implementation" of the semantic functions. Indeed, Goguen et al (1977) use the similar notion of *derivor* to formalise the concept of implementation for abstract data types. Further, this split allows us to be a little more general: a theory morphism $Th_A \rightarrow Th_B$ provides a means of deriving a Th_A-model from *every* model of Th_B so there is no need to repeat this part of the proof for each model separately. Actually, in some cases, such as where A is initial in Alg_A or where there is a homomorphism $A \rightarrow U_\sigma(T_B)$ we may establish a complete congruence between A and every Th_B-model all at once.

In terms of initial algebra semantics, the technique we have been discussing demonstrates that the following diagram exists and commutes. The abstract syntax is the initial Σ-algebra T_Σ and S_Σ andS'_Σ are the semantic Σ-algebras respectively derived from A and $U_\sigma(B)$ as described in §3.2.3.

$$\begin{tikzcd}
 & T_\Sigma \arrow[dl, "g"'] \arrow[dr, "k"] & \\
S_\Sigma \arrow[rr, "h"'] & & S'_\Sigma
\end{tikzcd}$$

The homomorphisms g and k exist in A and $U_\sigma(B)$ as (a set of) operations representing the semantic functions provided the semantic equations are homomorphic as discussed in §3.2.2. If there is a homomorphism, say $h': A \to U_\sigma(B)$ which clearly must be an identity (or isomorphism) on the syntactic subalgebra of A then $h'.g = k.h' = k$ so h is the restriction of h' to the semantic part of A and the diagram commutes. Basically, the fact that g and k are unique homomorphisms allows the mere existence of h to guarantee commutation.

The similarity with the so-called "Morris-square" (after Morris, 1973) is worth noting here. In essence, his advice is that to prove compiler correctness requires to show that a square of homomorphisms with source programs, source meanings, target programs and target meaning on the corners commutes.

$$\begin{tikzcd}
\text{source language} & L \arrow[r, "\text{compile}"] \arrow[d, "\text{semantics}"'] & T \arrow[d, "\text{semantics}"] & \text{target language} \\
\text{source meanings} & M \arrow[r, "\text{encode}"'] & U & \text{target meanings}
\end{tikzcd}$$

Now if source programs and target programs are identical as they are here, the compiler becomes completely trivial and the square collapes into the triangle diagram above.

We have yet to comment on the relationship between pairs of semantic Σ-algebras B and $U_\sigma(B)$ for some theory morphism σ. If we denote the Σ-algebra corresponding to B and $U_\sigma(B)$ as R_Σ and S'_Σ respectively, it is clear from inspection of the cases dealt with elsewhere in this thesis that S'_Σ can be obtained from R_Σ in a fashion similar to the application of U_σ. It makes little sense in this context to think of S'_Σ and R_Σ only in terms of Σ-algebras since U_σ only has an interesting effect on the semantic sorts and operators and these are "forgotten" when generating S'_Σ and R_Σ. Perhaps one intuitively helpful way to view the situation is to consider U_σ as having the effect of ***renaming*** some of the derived operators of B (in the ***clone*** of B; §2.1) and forgetting others. Thus B and $U_\sigma(B)$ can be considered as the same object viewed from the two attitudes of a Th_A-algebra and a Th_B-algebra, and their semantic Σ-algebra are similarly related. Basically the effect of U_σ is either to remove unnecessary complexity, say replacing a variable by a constant as in moving from continuation semantics to direct semantics by always supplying the identity continuation; or add in further (unnecessary) complexity by composing functions to consider extra arguments, as in going from direct to continuation semantics by composing the direct semantic function with "application of continuations to stores". In brief , these two examples are represented by the following two equations respectively:
$\mathcal{C}_D[\![c]\!]\rho\sigma = \mathcal{C}_C[\![c]\!]\rho(\lambda\sigma.\sigma)\sigma$ and $\mathcal{C}_C[\![c]\!]\rho\theta\sigma = \theta(\mathcal{C}_D[\![c]\!]\rho\sigma)$.

Our definition of semantic congruence may seem excessively restrictive especially given that to the present some such relations are not even able to be established. However, we maintain this view to be the appropriate one and that "congruence conditions" that fall outside our definition are "relations" or "facts" that may indeed be important or useful in themselves without actually being congruences. In the following sections of this chapter we will work through some examples (including the three already introduced) to test the usefulness of the algebraic framework we have set up. In most, though not all cases initiality assists us in proving the existence of appropriate homomorphisms.

4.2 Lambda Calculus Example

In this section, largely taken from Baker-Finch (1984b), we investigate in detail the congruence of the denotational and operational semantics introduced in §4.1.2. The operational definition follows, while the denotational one may be found in §3.1.2 and the presentation LC is given in §3.1.1.

<u>Domains</u>

$U = B + CLO$

$CLO = Lambda \times Ide \times E$

$E = (Ide \times U)^*$

<u>Semantic Function</u>

Eval: $Lambda \times E \to U$

(E1) $Eval[\![c\text{: constant}]\!]e = c$ *in* U

(E2) $Eval[\![x\text{: Ide}]\!]e = Lookup(x,e)$

(E3) $Eval[\![\lambda x.\eta]\!]\ e = \langle \eta,x,e\rangle$ *in* U

(E4) $Eval[\![\alpha(\beta)]\!]e = apply(Eval[\![\alpha]\!]\,e, Eval[\![\beta]\!]\,e)$

where

$$apply(a,b) = \begin{cases} Eval[\![\eta]\!]\,Extend(e,x,b) & \text{if } a = \langle \eta,x,e\rangle \\ \bot & \text{otherwise} \end{cases}$$

$Extend(e,x,u) = \langle x,u\rangle.e$

$$Lookup(x,e) = \begin{cases} 2nd\ (e{\downarrow}i) & \text{if } i \text{ exists s.t. } 1st\ (e{\downarrow}i) = x \text{ and } \\ & \forall\, j < i,\ 1st\ (e{\downarrow}j) \neq x \\ \bot & \text{otherwise} \end{cases}$$

$a{\downarrow}i$ is the ith element of the list a and 1st and 2nd respectively return the first and second item of a pair.

<u>operational semantics of lambda calculus</u>

We begin by briefly following an attempted proof of the congruence $Value(Eval[\![\phi]\!]\,e) = Val[\![\phi]\!]\,EN(e)$ described in §4.1.2 using the traditional techniques, to demonstrate how the proof breaks down in that situation. In outline, for our algebraic style of proof we intend to show that Op (the algebra corresponding to the operational semantics definition above) and Den are both models of LC and that furthermore Op is an initial model of LC. Thus by initiality there is a unique homomorphism mapping Op to Den and this can be shown to be the strong congruence desired. Den has already been shown to be in Alg_{LC} (§3.1.2) and the proof for Op is exactly analogous. By showing that the unique homomorphism $h\colon T_{LC} \to \mathrm{Op}$ is bijective (set-wise) we establish the initiality of Op. It will be seen in §4.2.3 that there are some difficulties involved in this second step.

4.2.1 The Scott-Strachey-Milne Approach

This section is a summary of a paper by Turner (1979), a least up to the point where the proof of congruence initially breaks down. The proof techniques are basically those developed by Milne (1974). First we repeat the definitions of the embedding functions and the congruence condition.

$$\mathrm{Value}(u) = \begin{cases} u \text{ } \mathit{in} \text{ } V & \text{if } u \in B \\ \lambda a.\mathrm{Val}[\![\eta]\!]\,\mathrm{EN}(e)[x/a] \text{ } \mathit{in} \text{ } V & \text{if } u = \langle \eta,x,e\rangle \in \mathrm{CLO} \end{cases}$$

$$\mathrm{EN}(e) = \lambda x.\mathrm{Value}(\mathrm{Lookup}(x,e))$$

For each $\phi \in$ Lambda,

$$\mathrm{Value}(\mathrm{Eval}[\![\phi]\!]\, e) = \mathrm{Val}[\![\phi]\!]\,\mathrm{EN}(e).$$

Structural induction over Lambda is not a valid means by which to attempt a proof of this congruence. This is clear due to the fact that the operational semantics does not directly provide the meaning of a λ-expression in terms of the meanings of its subcomponents. Rather, Eval is given as the fixed-point of a certain functional so fixed-point induction suggests itself as the appropriate proof technique.

In fact one half of the proof succeeds, namely $\mathrm{Value}(\mathrm{Eval}[\![\phi]\!]\, e) \leqslant \mathrm{Val}[\![\phi]\!]\,\mathrm{EN}(e)$ where "$\leqslant$" is the usual "less defined than or equal" operator. If we define a function F as follows:

$$F = \lambda S\lambda\phi\lambda e. \begin{cases} \mathrm{Lookup}(\phi,e) & \text{if } \phi \in \mathrm{Ide} \\ \langle \eta,x,e\rangle & \text{if } \phi = \lambda x.\eta \\ \mathrm{apply}_S(S[\![\alpha]\!]\, e, S[\![\beta]\!]\, e) & \text{if } \phi = \alpha(\beta) \end{cases}$$

then it is quite straightforward (e.g. Stoy, 1977) to show that

$\mathrm{Value}(F^k(\bot)[\![\phi]\!]\, e) \leqslant \mathrm{Val}[\![\phi]\!]\,\mathrm{EN}(e)$ by induction on k.

The converse, $\mathrm{Val}[\![\phi]\!]\,\mathrm{EN}(e) \leqslant \mathrm{Value}(\mathrm{Eval}[\![\phi]\!]\, e)$ however cannot be proved by fixpoint induction. If we follow through an attempt to establish this result

it will be seen that the proof breaks down in the induction step.
Define G as follows:

$$G = \lambda D\lambda\phi\lambda\rho. \begin{cases} \rho(a) & \text{if } \phi \in \mathrm{Ide} \\ \lambda a.D[\![\eta]\!]\rho\,[x/a] & \text{if } \phi = \lambda x.\eta \\ (D[\![\alpha]\!]\rho \mid \mathrm{FUN})(D[\![\beta]\!]\rho) & \text{if } \phi = \alpha(\beta) \end{cases}$$

At some point in the proof we must show

$$(G^k(\bot)[\![\alpha]\!]\,\mathrm{EN}(e) \mid \mathrm{FUN})(G^k(\bot)[\![\beta]\!]\,\mathrm{EN}(e)) \lessdot \mathrm{Value}(\mathrm{Eval}[\![\alpha(\beta)]\!]e).$$

By induction it is sufficient to show

$$\mathrm{Value}(\mathrm{Eval}[\![\alpha]\!]\,e)(\mathrm{Value}(\mathrm{Eval}[\![\beta]\!]\,e)) \lessdot \mathrm{Value}(\mathrm{Eval}[\![\alpha(\beta)]\!]e).$$

By the inequality established above the left hand side is an approximation to $(\mathrm{Val}[\![\alpha]\!]\,\mathrm{EN}(e) \mid \mathrm{FUN})(\mathrm{Val}[\![\beta]\!]\,\mathrm{EN}(e))$ which is $\mathrm{Val}[\![\alpha(\beta)]\!]\mathrm{EN}(e)$. So we need to establish that $\mathrm{Val}[\![\alpha(\beta)]\!]\mathrm{EN}(e) \lessdot \mathrm{Value}(\mathrm{Eval}[\![\alpha(\beta)]\!]e)$, but this is just what we are trying to prove! Thus it could be said that we must prove the theorem as a lemma to its own proof.

At this point Turner presents a less direct "congruence" based on the notion that congruent functions and closures must return congruent values when applied to congruent arguments. Though this is still quite a useful and natural concept of equivalence we argue that it does not constitute a congruence.

4.2.2 Op and Den are Models of LC

We have already shown in §3.1.2 that Den is a model of LC. The proof that Op is a model of LC is equally straightforward by choosing a carrier of Op for each sort of LC, a function of appropriate arity for each operator symbol of Σ (the signature of LC), and show that the equations ε of LC are satisfied by such a Σ-algebra. Instantiate the sorts as follows:

sort env:	E
sort W:	U
sort Abstr:	CLO

Then the operator symbols and functions correspond as follows:

Σ	Op
InjB: $B \to W$	*in* U

injA: Abstr → W	*in* U
err: → W	$\perp$
arid: → env	‹›
bind: env × Ide × W → env	Extend
find: env × Ide → W	Lookup
A: Lambda × Ide × env → Abstr	$\langle \_,\_,\_ \rangle$
appl: W × W → W	apply
M: Lambda × env → W	Eval

Again, the underline notation has been used for operations written in mixfix.

To test whether Op satisfied equations 1 to 9 of LC they may be translated into expressions of Op using the signature correspondence described above and then verified using the definitions of the operations of Op.

1. Lookup(‹›,x) = $\perp$.
   {find(arid,x) = err}
   Holds by the definition of Lookup since there is no i such that
   1st(‹›↓i) = x.
2. Lookup(Extend(e,y,u),x) = *if* x=y *then* u *else* Lookup(e,x).
   {find(bind(e,y,w),x) = *if* x=y *then* w *else* find(e,x)}
   Can easily be shown by considering the cases x=y and x≠y.
3. Eval⟦b: constant⟧e = b *in* U.
   {M(constant(b),e) = injB(b)}
   Immediate by E1 of Op.
4. Eval⟦x: Ide⟧e = Lookup(e,x).
   {M(var(x),e) = find(e,x)}
   Immediate by E2.
5. Eval⟦λx.η⟧e = ‹η,x,e› *in* U.
   {M(abstraction(x,η),e) - injA(A(η,x,e))}
   Immediate by E3.
6. Eval⟦α(β)⟧e = apply(Eval⟦α⟧e,Eval⟦β⟧e)
   {M(application(α,β),e) = appl(M(α,e),M(β,e))}
   Immediate by E4.
7. apply(b *in* U, u) = $\perp$
   {appl(injB(b),w) = err}
   Follows from definition of apply.

8. apply(<η,x,e) *in* U, u) = Eval⟦η⟧ Extend(e,x,u).
   {appl(injA(A(η,x,e)),w) = M(η,bind(e,x,w))}
   Follows from definition of apply.
9. apply($\perp$,u) = $\perp$
   {appl(err,w) = err}
   Follows from definition of apply.

Having thus shown Op to be in Alg_{LC}, we automatically have a unique homomorphism from T_{LC} to Op by the initiality of T_{LC}. To show that this is an isomorphism and hence that Op is initial in Alg_{LC} we may either show that this homomorphism is bijective or show that its inverse is a homomorphism from Op to T_{LC}.

4.2.3 Op is an Initial Model of LC+ε'

We would like to show that Op is initial in Alg_{LC} by showing h_E: $T_{LC,env} \rightarrow E$, h_W: $T_{LC,W} \rightarrow U$ and h_A:$T_{LC,Abstr} \rightarrow CLO$ to be bijective, the other carrier to carrier maps being identities. Unfortunately, h_W and h_A are not bijective and we must alter our semantic presentation slightly to proceed.

The problem lies in the fact that the members of $T_{LC,W}$ cannot be characterised by
$\{injB(b) \mid b \in B\} \cup \{injA(A(\eta,x,e)) \mid \eta \in Lambda, x \in Ide, e \in env\} \cup \{err\}$
which is what is required for a correspondence with U = B + CLO. By considering a non-terminating β-reduction, say λx.xx(λx.xx), $M_{T_{LC}}$ is defined such that $M_{T_{LC}}(\lambda x.xx(\lambda x.xx),e) = [M(\lambda x.xx(\lambda x.xx),e)]_{\varepsilon}$. This equivalence class contains no terms of the form injB(b) or injA(A(η,x,e)) or err. Thus, although all functions in T_{LC} are total, each non-terminating λ-expression may have a different meaning so $M_{T_{LC}}(\lambda x.xx(\lambda x.xx),e) \neq M_{T_{LC}}(\lambda y.yy(\lambda y.yy),e) \neq M_{T_{LC}}(\lambda x.xxx(\lambda x.xxx),e)$ and so on. Now in Op, Eval is made total by sending all non-terminating λ-expressions to $\perp$ so there is no bijection $T_{LC,W} \rightarrow U$, Op and T_{LC} are not isomorphic and therefore Op is not initial in Alg_{LC}.

To retrieve the situation we must restrict the algebraic semantics so that the error-like terms generated by non-terminating computations are all collapsed to err. Clearly the non-terminating λ-expressions cannot be finitely characterised, since this would be a solution to the halting problem.

We can however extend the set of equations ε of LC to $\varepsilon+\varepsilon'$ where ε' is the infinite (and undecidable) set of equations given below.

$\varepsilon' = \{x = err \mid x \in T_{LC,W}$ & $x \not\equiv injA(A(\eta,y,e))$ for any η,y,e
& $x \not\equiv injB(b)$ for any $b \in B\}$

In effect, ε' puts any term occurring in one of the equivalence classes that make up the carrier for W in $T_\Sigma/\equiv_\varepsilon$ that does not also contain a term such as injB(b) or injA(A(η,y,e)) into $[err]_{\varepsilon+\varepsilon'}$. The proof that h_W: $T_{LC+\varepsilon',W} \rightarrow U$ and h_A: $T_{LC+\varepsilon',Abstr} \rightarrow CLO$ are bijections now goes through easily by structural induction since $T_{LC+\varepsilon',W}$ can be characterised by the set expression we described above.

However we are left with a residual problem. Since we are now considering the category $Alg_{LC+\varepsilon'}$ rather than Alg_{LC}, Op and Den must be shown to satisfy $\varepsilon+\varepsilon'$ rather than just ε. To show that Op satisfies ε' we first translate the equations using the signature correspondence.

$\varepsilon'_{Op} = \{x = \bot \mid x \in U$ & $x \neq \langle\eta,y,e\rangle$ *in* U for any η,y,e
& $x \neq b$ *in* U for any $b \in B\}$

where "=" has replaced "≡" since Op has already been shown to satisfy ε. Clearly since U = B + CLO, ε'_{Op} reduces to the single trivially satisfied equation, $\bot = \bot$.

In trying to show that Den satisfies ε' we meet another small problem. If we translate ε' into the notation of Den we get

$\varepsilon'_{Den} = \{x = \bot \mid x \in V$ & $x \neq \lambda a.Val[\![\eta]\!]\rho\,[y/a]$ *in* V for any η,y,e
& $x \neq b$ *in* V for any $b \in B\}$

Now V = B + FUN , but there are many members of FUN that cannot be expressed in the form $\lambda a.Val[\![\eta]\!]\rho\,[y/a]$. To proceed we may alter the algebra Den simply by restricting the meaning domain so that Val is onto; i.e. Val: Lambda $\times$ ENV $\rightarrow$ (V *image* Val) where V *image* Val is the subset of V consisting of only those elements that are the result of applying Val to some λ-expression and some environment. This restriction has no effect on the semantics but reduces ε'_{Den} to an empty set. Further, there is a Σ-homomorphism (the identity) from this restriction of Den to the algebra Den itself. Thus by composition, if we show there is a homomorphism from Op to the restriction of Den, we may immediately deduce the existence of a homomorphism from Op to the full Den algebra.

Clearly such convolutions in the proof and such distortions of the semantic definitions seriously detract from what is otherwise a very straightforward and elegant technique. However we may take heart from the fact that for those cases where such unpleasant extensions to the set of equations are necessary, we can work in close analogy with the above discussion of ε'. In §4.3 we make another attack on this problem from a more basic starting point: The ways in which we make total functions from what are more naturally partial functions (Eval is a case in point).

To finalise the proof that Op is initial in LC+ε' we show that h_E: $T_{LC+\varepsilon',env} \to E$ is one-to-one by structural induction over $T_{LC+\varepsilon',env}$ (actually , the carrier of canonical term algebra isomorphic to $T_{LC+\varepsilon'}$). The details of the proof are easy but are included at the end of §4.2 for the sake of completeness, as are all other proofs. Finally to complete our proof that h: $T_{LC+\varepsilon'} \to$ Op is an isomorphism we need to show that h_E: $T_{LC+\varepsilon'} \to E$ is onto. Again, the proof is by structural induction, this time over E.

4.2.4 The Form of the Homomorphism from Op to Den

Up to this point we have shown Op to be an initial model of LC+ε' and Den to also be a model of LC+ε' and hence there is a unique Σ-homomorphism taking Op to Den. From our discussion in §4.1 we claim that this is all we need to prove to establish a congruence between Op and Den. However, we feel it proper at this expository stage to address the question of whether this Σ-homomorphism is indeed the intuitive congruence suggested in §4.1.2. In particular we are interested in the mappings U $\to$ V, CLO $\to$ FUN and E $\to$ ENV with the carrier to carrier maps being identities. It will be convenient to define these maps via $T_{LC+\varepsilon'}$ so let g: $T_{LC+\varepsilon'} \to$ Den be the unique homomorphism $\langle g_E$: $T_{LC+\varepsilon',env} \to$ ENV, g_W: $T_{LC+\varepsilon',W} \to$ V, g_A: $T_{LC+\varepsilon',Abstr} \to$ FUN$\rangle$. Then the unique homomorphism from Op to Den will be $\langle g_E . h_E'$: E $\to$ ENV, $g_W.h_W'$: U $\to$ V, $g_A.h_A'$: CLO $\to$ FUN$\rangle$ where h' denotes the inverse of h.

What we need to show is that $g_E . h_E'$ is the same as EN and $g_W.h_W'$ is the same as Value; i.e.

1. $g_W.h_W'(u) = b$ *in* V if $u = b$ *in* U, $b \in B$
2. $g_W.h_W'(u) = \lambda a.\text{Val}[\![\eta]\!]\, g_E.h_E'(e)[x/a]$ *in* V if $u = \langle\eta,x,e\rangle$ *in* U
3. $g_E.h_E'(e) = \lambda x.g_W.h_W'(\text{Lookup}(x,e))$.

The proofs are quite direct and simple and are relegated to the end of this section.

The algebraic framework has obviously been very useful, allowing us to develop a simple proof of a relation that was otherwise unattainable. The only slight question hangs over the equations ε', but as mentioned previously we demonstrate a technique to circumvent such problems in §4.3. In essence the reason that the congruence cannot be established by the traditional techniques is that they attempt to prove the homomorphism from Op to Den directly. Due to the nature of the definition of Op structural induction cannot be used and the only alternative, fixed-point (or computational) induction only goes round in circles. On the other hand, in the algebraic framework we can use structural induction over T_{LC} and indeed this is done in the proof that there is a unique homomorphism from T_{LC} to any other LC-algebra (see Goguen et al, 1977). Proving that T_{LC} and Op are isomorphic requires further structural induction over T_{LC} and also over the *domains* of Op which is quite valid since they are simple data structures. So underneath all the algebra the proof is in fact simply (though indirectly) by structural induction.

4.2.5 Proofs

Proposition

$h_E: T_{LC+\varepsilon',env} \to E$ is one-to-one.

Proof:

```
By induction over $T_{LC+\varepsilon',env}$
Suppose e1 and e2 are elements of $T_{LC+\varepsilon',env}$ and show that
$h_E(e1) = h_E(e2)$ only when e1 = e2.
case 1: e1 = arid
    $h_E(e1) = \langle\rangle$
    case 1.1: e2 = arid = e1.
    case 1.2: e2 = bind(e,x,injB(b))
        $h_E(e2) = \langle x, b \textit{ in } U\rangle.h_E(e) \neq h_E(e1)$
    case 1.3: e2 = bind(e,x,injA(A(η,y,e')))
        $h_E(e2) = \langle x, \langle \eta, y, h_E(e')\rangle \textit{ in } U\rangle.h_E(e) \neq h_E(e1)$
case 2: e1 = bind(e,x,injB(b))
    $h_E(e1) = \langle x, b \textit{ in } U\rangle.h_E(e)$
    case 2.1: e2 = arid
        $h_E(e2) = \langle\rangle \neq h_E(e1)$
```

case 2.2: e2 = bind(e',y,injB(a))

h_E(e2) = ⟨y, a *in* B⟩.h_E(e')

= h_E(e1) only when x=y, b=a and (by inductive hypothesis) e=e'

i.e. only when e1 = e2.

case 2.3 e2 = bind(e',y,injA(A(η,z,e'')))

h_E (e2) = ⟨y, ⟨η,z,h_E(e'')⟩ *in* U⟩.h_E(e') ≠ h_E(e1)

case 3: e1 = bind(e,x,injA(A(η,y,e')))

h_E (e1) = ⟨x, ⟨η,y,h_E(e')⟩ *in* U⟩.e

case 3.1: e2 = arid

h_E(e2) = ⟨⟩ ≠ h_E(e1)

case 3.2: e2 = bind(e'',z,injB(b))

h_E(e2) = ⟨z, b *in* B⟩.h_E(e'') ≠ h_E(e1)

case 3.3: e2 = bind(e'',z,injA(A(β,p,e''')))

h_E(e2) = ⟨z, ⟨β,p,h_E(e''')⟩ *in* U⟩.h_E(e'')

= h_E(e1) only when x=z, n=β, y=p (by the inductive hypothesis) e=e'' and e'=e'''

ie only when e1 = e2.

<u>Proprosition</u>:

h_E: $T_{LC+\varepsilon',env} \rightarrow$ E is onto.

<u>Proof</u>:

Let en be an element of E. We show by induction over E that there is always an element e of $T_{LC+\varepsilon',env}$ such that h_E(e) = en.

<u>case</u> 1: en = ⟨⟩

h_E(arid) = ⟨⟩

<u>case</u> 2: en = ⟨x, b *in* U⟩.en'

by inductive hypothesis, there is an e' such that h_E(e')= en', so h_E(bind(e',x,injB(b))) = ⟨x, b *in* U⟩.h_E(e') = en.

<u>case</u> 3: en = ⟨x, ⟨η,y,en'⟩ *in* U⟩.en''

by inductive hypothesis, there is an e' and an e'' such that h_E(e') = en' and h_E(e'') = en''.

so h_E(bind(e'',x,injA(A(η,y,e')))) =

⟨x, ⟨η,y,h_E(e')⟩ *in* U⟩.h_E(e'') = en.

Proposition:

$g_W.h_W'(b\ \mathit{in}\ U) = b\ \mathit{in}\ V$

Proof:

$h_W'(b\ \mathit{in}\ U) = injB(b)$

$g_W(injB(b)) = b\ \mathit{in}\ V.$

Proposition:

$g_W.h_W'(\langle\eta,x,e\rangle\ \mathit{in}\ U) = \lambda a.Val[\![\eta]\!]\ g_E.h_E'(e)[x/a]\ \mathit{in}\ V$

Proof:

$h_W'(\langle\eta,x,e\rangle\ \mathit{in}\ U) = injA(A(\eta,x,h_E'(e))),$

$g_W'(injA(A(\eta,x,h_E'(e)))) = \lambda a.Val[\![\eta]\!]\ g_E.h_E'(e)[x/a]\ \mathit{in}\ V.$

Proposition:

$g_E.h_E'(e) = \lambda x.\ g_W.h_W'(Lookup(x,e))$

Proof:

By induction over E.

case 1: $e = \langle\rangle$

$g_E.h_E'(e) = g_E(arid) = \lambda x \bot_V$

$Lookup(x,\langle\rangle) = \bot_U$

so $g_W.h_W'(Lookup(x,\langle\rangle)) = g_W.h_W'(\bot_U) = g_W(err) = \bot_V$

so $\lambda x \bot_V = \lambda x.g_W.h_W'(Lookup(x,\langle\rangle))$

case 2: $e = \langle x,u\rangle.e'$

$$\begin{aligned}
g_E.h_E'(e) &= g_E(bind(h_E'(e'),x,g_W.h_W'(u))) \\
&= g_E.h_E'(e')[x/g_W.h_W'(u)] \\
&= \lambda z. \begin{cases} g_E.h_E'(e')(z) & \text{if } z \neq x \\ g_W.h_W'(u) & \text{if } z = x \end{cases} \\
&= \lambda z. \begin{cases} \lambda x.\ g_W.h_W'(Lookup(x,e'))(z) & \text{if } z \neq x \text{ (induction)} \\ g_W.h_W'(u) & \text{if } z = x \end{cases} \\
&= \lambda z. \begin{cases} g_W.h_W'(Lookup(z,e')) & \text{if } z \neq x \\ g_W.h_W'(u) & \text{if } z = x \end{cases} \\
&= \lambda z. \begin{cases} g_W.h_W'(Lookup(z,\langle x,u\rangle.e') & \text{if } z \neq x \text{ (since } z \neq x) \\ g_W.h_W'(u) & \text{if } z = x \end{cases} \\
&= \lambda z. \begin{cases} g_W.h_W'(Lookup(z,e)) & \text{if } z \neq x \\ g_W.h_W'(Lookup(z,\langle x,u\rangle.e') & \text{if } z = x \\ & \text{(for any } e' \text{ actually)} \end{cases} \\
&= \lambda z.\ g_W.h_W'(Lookup(z,e))
\end{aligned}$$

4.3 Initial Algebra Fixed-Point Construction

A natural, convenient and very common way of defining functions is to do so in terms of a set of axioms or equations that the function must "obey". In particular, the semantic definitions given in this dissertation all use this technique. However, in the case where the equations are recursive and the function intended to be defined is *partial* (undefined for some arguments), there can be many solutions. The following example taken from Manna (1974) demonstrates this fact. Given the following equation defining F,

F(x,y) = *if* x=y *then* y+1 *else* F(x,F(x-1,y+1))

all of the following (directly defined) functions are among the possible solutions for F:

f_1(x,y) = *if* x=y *then* y+1 *else* x+1,

f_2 (x,y) =*if* x>y *then* x+1 *else* y-1,

f_3 (x,y) = *if* (x>y) & (x-y even) *then* x+1 *else* *undefined.*

So if we substitute the definitions of f_1, f_2, or f_3 for F in the above equation we get an identity.

It is natural to take f_3 as the function intended to be F above and it can be shown that f_3 is less defined than or equal to any other solution of the above recursive equation. This observation leads to the traditional approach of *least fixed-points* wherein a single distinguished element, usually denoted ω or $\perp$ and representing the "value" *undefined,* is added to the domain of the function and a partial order is constructed to reflect the notion of "less defined than or equal". It is then stated that the intended solution of a recursive equation is the *least-defined fixed-point* of that equation (actually, the functional represented by the equation); f_3 in the case above. An important point to note here is that the process relies on eliminating partial functions, converting them to total functions by adding an extra element to the domains to denote an actual *value* representing those places where the partial function is undefined.

Given this rather rudimentary foundation it is possible for us to temporarily lay aside the widely-used least fixed-point approach and ask the question: how do we judge the suitability of a fixed-point solution? We suggest the answer is simply that no "sensible" value should be returned at a point where the partial function is undefined. In the traditional approach outlined above there can be (at most) only one such solution because we add to the domain only one value that is not sensible - ω, and that solution is the least fixed-point.

However, the problems we had with initiality of Op in §4.2.3 and the term algebra construction hint at a different possibility: that of adding as many different *undefined values* as there are separate paths of non-terminating computation (considering the equations as rewrite rules). We proceed now with a definition of such a domain extension and the associated solution of recursive function definitions and follow this with an investigation of some of the properties of the construction and its application to our work.

4.3.1 Definition

In general the type of function definitions we are dealing with here feature the following elements:

(a) some domains,

(b) the arity of some functions to be defined to operate on those domains,

(c) some equations intended to define those functions , possibly directly or mutually recursive and involving:

(d) certain other pre-defined functions on those domains.

Quite likely (a) and (b) will only be implicit in the definition. If we consider the names of the domains (a), together with (b) and the arities of (d), plus the elements of the domains (treated as constant operators) to be a signature Ω, we may generate the word algebra T_Ω. We can generate the smallest Ω-congruence, $\equiv_C$ based on the equations (c). By treating the functions (d) as sets of ordered pairs, we may further generate $\equiv$, the smallest Ω-congruence containing $\equiv_C$ and (d). If we now take the quotient $T_\Omega/\equiv$ then the completion of some domain A (say) may be denoted A^+ and can be defined to be

$$A^+ \cong A \cup (T_\Omega/{\equiv_A} - \{[a]_\equiv \mid a \in A\})$$

or, more simply

$$A^+ \cong T_\Omega/{\equiv_A}$$

Similarly the completion of $f: X \rightarrow A$ is $f^+: X^+ \rightarrow A^+$ and $f^+(x) = [f(x)]_\equiv$. Again, $f^+ \cong f_{T\Omega/\equiv}$.

An example may help clarify matters. Consider the usual recursive definition of factorial over the integers, fact: $\mathbf{Z} \to \mathbf{Z}$ and $\mathrm{fact}(i) = i{=}0 \to 1; i \times \mathrm{fact}(i-1)$. The four elements outlined above are present:

(a) domain - $\mathbf{Z}$

(b) arity of fact - fact: $\mathbf{Z} \to \mathbf{Z}$

(c) equation defining fact - $\mathrm{fact}(i) = i = 0 \to 1; i \times \mathrm{fact}(i-1)$

(d) pre-defined functions - $\_{=}0 \to 1; \_ : \mathbf{Z} \times \mathbf{Z} \to \mathbf{Z}$

$\_ \times \_ : \mathbf{Z} \times \mathbf{Z} \to \mathbf{Z}$

$\_ -1 : \mathbf{Z} \to \mathbf{Z}$

In this case T_Ω has one sort, $T_{\Omega\mathcal{Z}} = \{\ulcorner \mathrm{fact}(i) \urcorner \mid i \in T_{\Omega\mathcal{Z}}\} \cup \{\ulcorner i = 0 \to 1; j \urcorner \mid i,j \in T_{\Omega\mathcal{Z}}\} \cup \{\ulcorner i \times j \urcorner \mid i,j \in T_{\Omega\mathcal{Z}}\} \cup \{\ulcorner i-1 \urcorner \mid i \in T_{\Omega\mathcal{Z}}\} \cup \mathbf{Z}$. If we respectively call c and the definitions of d ε and ε', we get

$\varepsilon = \{\langle \ulcorner \mathrm{fact}(i) \urcorner, \ulcorner i = 0 \to 1; \mathrm{fact}(i-1) \urcorner \rangle \mid i \in T_{\Omega\mathcal{Z}}\}$ and

$\varepsilon' = \{\langle \ulcorner 0 = 0 \to 1; j \urcorner, 1 \rangle\} \cup \{\langle \ulcorner i = 0 \to 1; j \urcorner, j \rangle \mid i \neq 0\} \cup \{\langle \ulcorner i \times j \urcorner, i \times j \rangle \mid i,j \in \mathbf{Z}\} \cup$
$\{\langle \ulcorner i-1 \urcorner, i-1 \rangle \mid i \in \mathbf{Z}\}$

and $T_\Omega/{\equiv}$ is as follows:

$\mathcal{Z}^+ \cong T_\Omega/{\equiv_\mathcal{Z}} \cong \mathbf{Z} + R$

where $R = \{\ulcorner \mathrm{fact}\ (i) \urcorner \mid i \in \mathbf{Z}\mathrm{neg}\} \cup \{\ulcorner \mathrm{fact}\ (i) \urcorner \mid i \in R\} \cup$
$\{\ulcorner i \times j \urcorner \mid i \in R, j \in \mathcal{Z}^+\} \cup \{\ulcorner i \times j \urcorner \mid j \in R, i \in \mathcal{Z}^+\} \cup$
$\{\ulcorner i-1 \urcorner \mid i \in R\} \cup$
$\{\ulcorner i{=}0 \to 1; j \urcorner \mid i \in R, j \in \mathcal{Z}^+\} \cup \{\ulcorner i{=}0 \to 1; j \urcorner \mid j \in R, i \in \mathcal{Z}^+\}$

$\mathrm{fact}^+(i) = i \in \mathbf{Z}\mathrm{pos} \to i!; \ulcorner \mathrm{fact}(i) \urcorner$

$i \times^+ j = i \in \mathbf{Z}\ \&\ j \in \mathbf{Z} \to i \times j; \ulcorner i \times j \urcorner$

$i\text{-}1^+ = i \in \mathbf{Z} \to i-1, \ulcorner i-1 \urcorner$

$i{=}0 \to 1;^+ j = i \in \mathbf{Z}\ \&\ j \in \mathbf{Z} \to$
$i = 0 \to 1; j\ ; \ulcorner i{=}0 \to 1; j \urcorner$

The overall effect is to add a new *undefined* element to the domain for each non-terminating computation unless it can be reduced to another term for which we already have such an element. For times when we may wish to be explicit about using such a completion, we introduce the (generic) function *afix*: $[D \to D] \to D$ distinguished from the usual least fixed-point *fix*: $[D \to D] \to D$ but allow the ambiguity of denoting both types of domain completion as D^+. Thus for the factorial example above, $\mathrm{fact}^+: \mathbf{Z}^+ \to \mathbf{Z}^+$ and $\mathrm{fact}^+ = \mathit{afix}(\lambda F \lambda i.\ i{=}0 \to 1; F\ (i-1))$.

Clearly, such a construction satisfies our criterion of giving no sensible value to points where the partial function is undefined and as will be seen in §4.3.2, if we use this concept rather than the least fixed-point for defining the operational semantics of the lambda calculus, the need to add the rather unpleasant set of equations of §4.2.3 can be avoided.

Finally, we note that our domain completions and the function *afix* depend to a degree on the context of the pre-defined functions involved in the definition. Clearly in our factorial example above we could have easily identified the pre-defined functions to be other than those used, for example

$\_ = 0 \rightarrow 1; \_ \times \_ : \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$ and

$\_ - 1 : \mathbb{Z} \rightarrow \mathbb{Z}$.

The completions will differ in the following way: for the original definition, $\mathbb{Z}^+$ will contain elements such as $[\ulcorner 3 \times \text{fact}\ (-1)\urcorner]_{\equiv}$ whereas if we used the alternative set of pre-defined functions, no such term can be generated. Thus in the two cases the elements added to $\mathbb{Z}$ to make $\mathbb{Z}^+$ will be different. This is perhaps a little unfortunate and and suggests that full details of the pre-defined functions must be given to indicate that the true nature of *afix* and the domain completions. Fortunately, for our applications such details are automatically included through our use of signatures in semantic presentations.

4.3.2 The Lambda Calculus Example Revisited

For the traditional proof of congruence of the operational and denotational semantics of the lambda calculus the usual partial order/ least fixed-point construction was used to make the functions Eval and apply total in a sensible way, presumably because it then matches the denotational semantics and possibly also because the consideration of alternatives was of no apparent value. So although it is most natural to view the interpreter as a partial function it needs to be made total so that it coincides with the denotational definition and it is therefore convenient to use the same machinery as denotational semantics.

Our experience suggests that this is not the most convenient approach in our algebraic framework. Functions do need to be made total to fit our definition of many-sorted algebras, but it would seem convenient in the case under discussion to make Op coincide with the initial model of LC. Such a change to Op could not be said to actually change the semantics it represents since the change is only to points where the partial functions Eval and apply are undefined.

By using our initial algebra fixpoint explicitly for the operational semantics, we can arrange that Op is initial in Alg_{LC}. The following definition avoides recursion by explicit use of *afix* assuming the pre-defined functions be to those identified by presentation LC. The (rather unwiedly) notation used for defining mutually recursive functions is that also used for the usual function *fix* and is discussed in Stoy (1977) and Manna (1974).

<u>Domains</u>

$U = B + CLO + \{\bot\}$

$CLO = Lambda \times Ide \times E$

$E = (Ide \times U)^*$

<u>Semantic Function</u>

$Eval\colon\ Lambda \times E^+ \to U^+$

$$
\begin{aligned}
\langle Eval, apply\rangle = afix\,(\lambda EA.\langle \lambda \ell e.\ & \begin{cases} c\ \mathit{in}\ U & \text{if } \ell = c \in B \\ Lookup(x,e) & \text{if } \ell = x \in Ide \\ \langle \eta, x, e\rangle,\ \mathit{in}\ U & \text{if } \ell = \lambda x.\eta \\ A(E\llbracket\alpha\rrbracket e,\ E\llbracket\beta\rrbracket e) & \text{if } \ell = \alpha(\beta) \end{cases} \\
,\ \lambda ab.\ & \begin{cases} E\llbracket\eta\rrbracket\, Extend(e,x,\beta) & \text{if } a = \langle \eta, x, e\rangle \\ \bot & \text{otherwise} \end{cases} \\
\rangle)\ &
\end{aligned}
$$

Extend and Lookup are unchanged.

<u>operational semantics of lambda calculus (Version 2)</u>

The (perhaps rather surprising) explicit inclusion of an element "$\bot$" in the domain U does not imply any ordering. Rather, it is a legacy of the unfortunate and somewhat unorthodox explicit use of $\bot$ in the original definitions of apply and Lookup for denoting situations other than non-terminating computations. A more standard technique would have been to use a further explicit error term (usually denoted ?)) so that in the original operational definition, $U = B + CLO + \{?\}$ and ? replaces $\bot$ in the definitions of apply and Lookup. Had this been the case, the more acceptable element ? would also have been used in version 2 of the semantics rather than the somewhat misleading $\bot$.

The proof of initiality of the algebra associated with the new definition (call it Op') is now immediate since $U^+ \cong T_{LC,U}$ and $E^+ \cong T_{LC,E}$. This algebra, Op' which is in our view a quite satisfactory completion of the (partial function) interpreter, *is* an initial model of LC so the equations ε' of §4.2.3 become unnecessary and the congruence of the operational and denotational definitions is established in the straightforward manner originally proposed. In later proofs involving operational semantics we shall have no hesitation in using the initial algebra fixed-point construction rather than the traditional least fixed-point construction.

4.3.3 General Properties and Observations

In this section we wish to consider briefly a number of the more interesting and relevant properties of the initial algebra fixed-point. This does not in any way purport to be a complete or structured investigation.

As mentioned in §4.3.1 the "choice" of predefined functions greatly affects the domain completion generated by our technique. This does not cause a problem here since the nature of our endeavour is such that signatures are always provided, but in general some similar explicit identification of the pre-defined functions is required. Thus for some set of predefined functions P, we should perhaps write $afix_P$ and D^{+P} rather than the ambiguous afix and D^+.

One of the nice features of the initial algebra fixed-point construction when it is applied to operational semantics is that it is more "computational" than the traditional least fixed-point approach. By this we mean that the *undefined* values added to the domain directly reflect all possible attempts to evaluate the function at a point where it is not defined. For example, in the case of the definition of factorial above, the result of the expression fact (-2) is [⌜fact (-2)⌝] and other expressions in that equivalence class include ⌜-2 × fact (-3)⌝, ⌜-3 × -2 × fact (-4)⌝ and so on. Every (reasonable or otherwise) computation rule will be represented. This is quite pleasant in a way since if we consider the introduced *undefined* elements as *error messages* then they contain maximum possible information about what has been asked of the interpreter and the paths it could possibly follow. In this respect at least, the initial algebra fixed-point seems more suited to operational semantics than the least fixed-point.

Monotonicity and continuity of functions are very important notions when dealing with the traditional completion technique since they provided a simple way of ascertaining whether a least fixed-point exists. The question naturally arises whether such notions are relevant to our algebraic completion. Our intuition suggests not, since a quotient algebra exists for *any* presentation and thus we should succeed with the completion and the initial algebra fixed-point for *any* function definition irrespective of monotonicity. We demonstrate this by examining two simple examples taken from Manna (1974).

<u>Example 1</u>

$\tau[F](x)$: *if* $F(x) \equiv 0$ *then* 1 *else* 0

τ is a functional corresponding to the recursive equational definition $F(x) =$ *if* $F(x) \equiv 0$ *then* 1 *else* 0. τ is a nonmonotonic functional over $[\mathbf{N}^+ \rightarrow \mathbf{N}^+]$ that has *no* fixed-points.

Our approach is the following.

(a) domain - $\mathbf{N}$

(b) arity - $F: \mathbf{N} \rightarrow \mathbf{N}$

(c) $F(x) =$ *if* $F(x) = 0$ *then* 1 *else* 0

(d) pre-defined functions -

*if*_ = 0 *then*_*else*_: $\mathbf{N} \times \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$

So calling the implicity represented signature Ω,

$T_{\Omega,N} = \mathbf{N} \cup \{$*if* $i = 0$ *then* j *else* $k \mid i,j,k \in T_{\Omega,N}\} \cup \{F(i) \mid i \in T_{\Omega,N}\}$

$T_{\Omega}/\equiv_N \; = \mathbf{N} \cup \{[F(i)]_{\equiv} \mid i \in T_{\Omega}/\equiv_N\}$

and the two summands are distinct. Thus any attempt to evaluate F results in an error message whcih is as we would wish.

<u>Example 2</u>

$\tau'[G](x)$: *if* $G(x) \equiv 0$ *then* 0 *else* 1

τ' is a functional corresponding to the recursive equational definition $G(x) =$ *if* $G(x) \equiv 0$ *then* 0 *else* 1. τ' is a nonmonotonic functional over $[\mathbf{N}^+ \rightarrow \mathbf{N}^+]$ that has *two* fixed-points, 0 and 1, but no least fixed-point.

Our approach is the following.

(a) domain - $\mathbf{N}$

(b) arity - $G: \mathbf{N} \rightarrow \mathbf{N}$

(c) $G(x) =$ *if* $G(x) = 0$ *then* 0 *else* 1

(d) pre-defined functions -

*if*_ = 0 *then*_*else*_: $\mathbf{N} \times \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$

So calling the implicity represented signature Σ,

$T_{\Sigma,N} = \mathbf{N} \cup \{$ *if* $i = 0$ *then* j *else* $k \mid i,j,k \in T_{\Sigma,N}\} \cup \{G(i) \mid i \in T_{\Sigma,N}\}$
$T_{\Sigma}/{\equiv_N} = \mathbf{N} \cup \{[G(i)]_{\equiv} \mid i \in T_{\Sigma}/{\equiv_N}\}$
and the two summands are distinct. Thus any attempt to evaluate G results in an error message which is as we would wish.

Clearly the initial algebra fixed-point construction can be useful for our purposes and we shall make further use of it in the ensuing sections. It is at least acceptable according to our earlier stated criterion of not giving sensible values where the partial function is undefined and it can be argued that for operational semantics where the concept of *computing* a result is central, our algebraic domain completion is a better reflection of the situation than the more usual least fixed-point and complete partial order style of completion.

4.4 Addition Expression Example

In this section we consider the congruence of two very simple semantic models (adapted from Stoy, 1977) that are clearly from different theories. As discussed in §4.1.3 the notion of congruence here coincides with a theory morphism so our investigation will involve defining the theory morphism, proving that it is such and examining the derived model. Since this is the first example of such a congruence (and a very simple one) we will include considerable detail that in later examples would be tiresome and could cloud the central issues. We begin by repeating the semantic definitions given in §4.1.3 and presenting theories of which they are models.

4.4.1 The Presentations and Models

We refer the reader once again to the direct and stack semantics for addition expressions which we duplicate here. The algebras associated with these definitions will be called Dir and Stk respectively.

<u>Syntax</u>

⟨Exp⟩:: = ⟨N⟩ | ⟨Exp⟩ + ⟨Exp⟩

<u>Domain</u>

$\mathbb{N}$ natural numbers

<u>Semantic Function</u>

$\mathcal{E}_D$: Exp → $\mathbb{N}$

(1) $\mathcal{E}_D[\![n]\!] = n$

(2) $\mathcal{E}_D[\![e_1 + e_2]\!] = \mathcal{E}_D[\![e_1]\!] + \mathcal{E}_D[\![e_2]\!]$

<u>Dir-direct semantics for addition expressions</u>

<u>Syntax</u>

⟨Exp⟩:: = ⟨N⟩ | ⟨Exp⟩ + ⟨Exp⟩

<u>Domains</u>

$\mathbb{N}$	natural numbers
$\mathbb{N}^*$	stacks are represented by sequences of natural numbers

<u>Semantic Function</u>

$\mathcal{E}_S$: Exp × $\mathbb{N}^*$ → $\mathbb{N}^*$

(1) $\mathcal{E}_S[\![n]\!]\zeta = n \; \mathit{cat} \; \zeta$

(2) $\mathcal{E}_S[\![e_1 + e_2]\!]\zeta = add(\mathcal{E}_S[\![e_2]\!](\mathcal{E}_S[\![e_1]\!]\zeta))$
where $add(\zeta) = (\zeta\downarrow 1 + \zeta\downarrow 2)\, \textit{cat}\,(\textit{tail}\,(\textit{tail}\,(\zeta))$.

<u>Stk - stack semantics for addition expressions</u>

As would perhaps be expected for such simple semantics involving such fundamental domains, the semantic presentations bear considerable resemblance to the definitions above.

<u>Signature Ω_D</u>
syntactic sort Exp
    const: N → Exp
    plus: Exp × Exp → Exp
sort N
    zero: → N
    succ: N → N
    sum: N × N → N
$\mathcal{E}_D$: Exp → N

<u>Equations ε_D</u>

D1. sum(zero,n) = n
D2. sum(succ(n),m) = sum(n,succ(m))
D3. $\mathcal{E}_D$(const(n)) = n
D4. $\mathcal{E}_D$(plus(e_1,e_2)) = sum($\mathcal{E}_D$(e_2), $\mathcal{E}_D$(e_1))

<u>D - direct semantics presentation</u>

<u>Signature Ω_S</u>
syntactic sort Exp
    const: N → Exp
    plus : Exp × Exp → Exp
sort N
    zero: → N
    succ: N → N
    sum: N × N → N
sort Stack
    empty: → Stack
    push: Stack × N → Stack

pop: Stack → Stack
top: Stack → N
$\mathcal{E}_S$: Exp × Stack → Stack

<u>Equations ε_S</u>

S1. sum(zero,n) = n
S2. sum(succ(n),m) = sum(n,succ(m))
S3. top(push(s,n)) = n
S4. pop(push(s,n)) = s
S5. $\mathcal{E}_S$(const(n),s) = push(s,n)
S6. $\mathcal{E}_S$(plus(e_1,e_2),s) = push(pop(pop(s') sum(top(s'),top(pop(s'))))
where s' = $\mathcal{E}_S$(e_2,$\mathcal{E}_S$(e_1,s))

<u>S - stack semantics presentation</u>

To show that Dir is a model of Th_D and Stk is a model of Th_S we need to instantiate the sorts and operator symbols of the presentations with carriers and functions from the algebras and demonstrate that the equations ε_D and ε_S respectively are satisfied. To distinguish between identical operator symbols from the two presentations we decorate them with subscripts such as $const_D$ and $const_S$. The signature correspondences are as follows:

D	Dir
$const_D$: N → Exp	‹Exp›::= ‹N›
$plus_D$: Exp × Exp → Exp	‹Exp›::= ‹Exp› + ‹Exp›
$zero_D$: → N	0
$succ_D$: N → N	_ + 1
sum_D: N × N → N	_ + _
$\mathcal{E}_D$: Exp → N	ε_D

S	Stk
$const_S$: N → Exp	‹Exp›::= ‹N›
$plus_S$: Exp × Exp → Exp	‹Exp›::= ‹Exp› + ‹Exp›
$zero_S$: → N	0
$succ_S$: N → N	_ + 1
sum_S: N × N → N	_ + _
empty: → Stack	‹›
push: Stack × N → Stack	_ *cat* _

pop: Stack $\to$ Stack	*tail*
top: Stack $\to$ N	$\_ \downarrow 1$
$\mathcal{E}_S$: Exp $\times$ Stack $\to$ Stack	$\mathcal{E}_S$

The equations of ε_D and ε_S are so trivally satisfied that we eschew any further considerations of proofs that Dir is a model of Th_D and Stk is a model of Th_S.

4.4.2 The Theory Morphism and the Derived Model

In §4.1.3 we suggested that the congruence between Dir and Stk embodied in the following statement:

For all $e \in$ Exp and $\zeta \in N^*$,
$\mathcal{E}_D[\![e]\!] = (\mathcal{E}_S[\![e]\!]\zeta)\downarrow 1$

resembles a derivor or a theory morphism. In this section we explicitly define the theory morphism σ: $Th_D \to Th_S$, prove that it is indeed a theory morphism and generate the derived model U_σ(Stk) which will later be related to Dir.

Following the convention mentioned in §2 we use σ to represent both the sort map and the operator symbol map.

$\sigma(Exp_D) = Exp_S$
$\sigma(N_D) = N_S$
$\sigma(const_D) = const_S$
$\sigma(plus_D) = plus_S$
$\sigma(zero_D) = zero_S$
$\sigma(succ_D) = succ_S$
$\sigma(sum_D) = sum_S$
$\sigma(\mathcal{E}_D)(e) = top(\mathcal{E}_S(e,s))$ for any stack expression, s. Though the choice of s is irrelevant, it must be specified for σ to be fully defined. The simplest choice is s = empty.

By the presentation lemma (§2.3), to show that σ is a theory morphism we need only show $\mathit{Eqn}(\sigma)$(D1, ... D4) are in ε_S. In other words we translate the equations D1, D2, D3 and D4 using the definition of σ and show that the new equations can be established from S1 - S6 of ε_S.

$\sigma(D1)$: $sum_S(zero_S,n) = n$

σ(D2): $\mathrm{sum}_S(\mathrm{succ}_S(n),m) = \mathrm{sum}_S(n,\mathrm{succ}_S(m))$

σ(D3): $\mathrm{top}(\mathcal{E}_S(\mathrm{const}_S(n),\mathrm{empty})) = n$

σ(D4): $\mathrm{top}(\mathcal{E}_S(\mathrm{plus}_S(e_1,e_2),\mathrm{empty})) = \mathrm{sum}_S(\mathrm{top}(\mathcal{E}_S(e_2,\mathrm{empty})),$
$\mathrm{top}(\mathcal{E}_S(e_1,\mathrm{empty})))$

Both σ(D1) and σ(D2) are exactly the equations S1 and S2 while σ(D3) can be shown from S5 and S3. The final equation σ(D4) requires a more detailed proof.

It is here that our requirement that sort Exp be freely interpreted comes into play (see §3.3.5). Without such a restriction σ(D4) cannot be established from S1 ... S6 by the usual system of equational inference based on the reflexivity, symmetry and transitivity of equality plus the substitution properties. Thus σ(D4) is not an equation of Th_S. Expressed model-theoretically (rather than proof-theorectically), if we allow a non-initial interpretation of Exp then in some algebra, say A, there is an element of Exp_A, say Q, that is not the value of any term (i.e. it is "junk") and $\mathcal{E}_{SA}(Q,s0) = s$. Clearly by substituting Q for e_2 in σ(D4) we see that the equation does not hold for A and thus is not in Th_S. Hence it is only through our insistence that Exp be freely interpreted that induction on the structure of the terms becomes a valid means by which to find equations holding in a theory.

Rather than directly deriving σ(D4) it is convenient first to establish the result $\mathcal{E}_S(e,s_1) = \mathrm{push}(s_1,\mathrm{top}(\mathcal{E}_S(e,s_2)))$ for any s_2. Using this equation, the proof of σ(D4) is as follows:

$\mathrm{top}(\mathcal{E}_S(\mathrm{plus}(e_1,e_2),\mathrm{empty}))$

$= \mathrm{sum}(\mathrm{top}(\mathcal{E}_S(e_2,\mathcal{E}_S(e_1,\mathrm{empty}))),\mathrm{top}(\mathrm{pop}(\mathcal{E}_S(e_2,\mathcal{E}_S(e_1,\mathrm{empty})))))$ - S6, S3

$= \mathrm{sum}(\mathrm{top}(\mathrm{push}(\mathcal{E}_S(e_1,\mathrm{empty}),\mathrm{top}(\mathcal{E}_S(e_2,\mathrm{empty})))),$
$\mathrm{top}(\mathrm{pop}(\mathrm{push}(\mathcal{E}_S(e_1,\mathrm{empty}),\mathrm{top}(\mathcal{E}_S(e_2,\mathrm{empty}))))))$ - above result

$= \mathrm{sum}(\mathrm{top}(\mathcal{E}_S(e_2,\mathrm{empty})),\mathrm{top}(\mathcal{E}_S(e_1,\mathrm{empty})))$ - S3, S4

The underlying result is proved by induction over the structure of expressions.

<u>case 1</u>: $e = \mathrm{const}(n)$

$\mathcal{E}_S(\mathrm{const}(n),s_1) = \mathrm{push}(s_1,n)$ - S5

$= \mathrm{push}(s_1,\mathrm{top}(\mathrm{push}(n,s_2)))$ - S3

$= \mathrm{push}(s_1,\mathrm{top}(\mathcal{E}_S(n,s_2)))$ - S5

<u>case 2</u>: $e = plus(e_1,e_2)$

$\mathcal{A}_S(plus(e_1,e_2),s_1) = push(pop(pop(s')),sum(top(s'),top(pop(s'))))$

  where $s' = \mathcal{A}_S(e_2,\mathcal{A}_S(e_1,s_1))$ - S6

  $s' = push(push(s_1,top(\mathcal{A}_S(e_1,s_2))),top(\mathcal{A}_S(e_2,\mathcal{A}_S(e_1,s_2))))$

  - inductive hypothesis

  $= push(s_1,sum(top(s'),top(pop(s'))))$ - S4

  $= push(s_1,sum(top(\mathcal{A}_S(e_2,\mathcal{A}_S(e_1,s_2))),top(\mathcal{A}_S(e_1,s_2))))$

  - S3, S4

  $= push(s_1,sum(top(\mathcal{A}_S(e_2,\mathcal{A}_S(e_1,s_2))),$
  $top(pop(push(\mathcal{A}_S(e_1,s_2),$
  $top(\mathcal{A}_S(e_2,s_3)))))))$

  - S4

  $= push(s_1,sum(top(\mathcal{A}_S(e_2,\mathcal{A}_S(e_1,s_2))),$
  $top(pop(\mathcal{A}_S(e_2,\mathcal{A}_S(e_1,s_2))))))$

  - inductive hypothesis

  $= push(s_1,top(push(pop(pop(\mathcal{A}_S(e_2,\mathcal{A}_S(e_1,s_2)))),$
  $sum(top(\mathcal{A}_S(e_2,\mathcal{A}_S(e_1,s_2))),$
  $top(pop(\mathcal{A}_S(e_2,\mathcal{A}_S(e_1,s_2))))))))$

  - S3

  $= push(s_1,top(\mathcal{A}_S(plus(e_1,e_2),s_2)))$ - S6

Having thus shown σ to be a theory morphism $Th_D \rightarrow Th_S$, we may derive a D-algebra from any S-algebra by the contravariant application of σ. In particular we are interested in deriving a D-algebra U_σ(Stk) from the S-algebra Stk. The carriers of Exp_D and N_D in U_σ(Stk) must be the carriers of Exp_S and N_S in Stk since the definition of σ says that $\sigma(Exp_D) = Exp_S$ and $\sigma(N_D) = N_S$. The following definition of U_σ(Stk) uses notation similar to the semantic definitions of Dir and Stk for comparison purposes.

<u>Syntax</u>

<Exp>:: = <N> | <Exp> + <Exp>

<u>Domain</u>

N	natural numbers

<u>Semantic Function</u>

$\mathcal{E}_{US}: Exp \rightarrow \mathbf{N}$

(1) $\mathcal{E}_{US}[\![n]\!] = (\mathcal{E}_S[\![n]\!]\langle\rangle)\downarrow 1 = (n\ \mathit{cat}\ \langle\rangle)\downarrow 1 = n$

(2) $\mathcal{E}_{US}[\![e_1 + e_2]\!] = (\mathcal{E}_S[\![e_1 + e_2]\!]\langle\rangle)\downarrow 1$

<u>U_σ(Stk) - derived semantics for addition expressions</u>

It would have been possible to come up with the direct definition $\mathcal{E}_{US}[\![e_1 + e_2]\!] = \mathcal{E}_{US}[\![e_1]\!] + \mathcal{E}_{US}[\![e_2]\!]$, but leaving it in terms of $\mathcal{E}_S$ reflects the fact that $\mathcal{E}_{US}$ is derived from $\mathcal{E}_S$. We now move on to consider how Dir and U_σ(Stk) are related.

4.4.3 The Relation Between Dir and U_σ(Stk)

In §4.1.3 it was suggested that proving the congruence $\mathcal{E}_D[\![e]\!] = (\mathcal{E}_S[\![e]\!]\langle\rangle)\downarrow 1$ was equivalent to showing Dir = U_σ(Stk). In fact we have been deliberately vague in the definitions of Dir and Stk to demonstrate that this is not necessarily the case. In Stk we have not given precise definitions of the auxiliary functions $\downarrow 1$, $\downarrow 2$ and *tail*, particularly their boundary behaviour. By considering expressions as $\langle\rangle\downarrow 1$ and *tail*$(\langle\rangle)$ it it clear that some sort of error element must be available in both domains $\mathbf{N}$ and $\mathbf{N}^*$ yet no mention is made of such in the definition. Further, no such error element is needed in the domain $\mathbf{N}$ of Dir so it is clear that Dir and U_σ(Stk) may not coincide exactly depending on how we determine the final nature of the domains.

Now the intention of Stoy (1977), from which our definitions have been adapted, is clear: all domains are complete lattices with a distinguished, incomparable error element whether it is needed or not. Thus they all have $\perp$, $\top$ and ?, with *tail*$(\langle\rangle)$ = ? and $(\langle\rangle)\downarrow 1$ = ?. In this case we could establish the full equality of Dir and U_σ(Stk). However, if the domains do not completely coincide we can still establish that there is a (one-one) homomorphism Dir $\rightarrow$ U_σ(Stk), and that homomorphism will satisfy the condition $\mathcal{E}_D[\![e]\!] = \mathcal{E}_{US}[\![e]\!] = (\mathcal{E}_S[\![e]\!]\langle\rangle)\downarrow 1$ which was the original statement of congruence.

Hence we intend to remain vague in our definitions of Stk and Dir and show that Dir and U_σ(Stk) are homomorphic irrespective of whether we have an error element in domain $\mathbf{N}$ of Dir. In other words we are establishing the congruence for several variations on Dir.

We could establish the existence of h: Dir $\rightarrow$ U_σ(Stk) by showing Dir to be initial in Th_D (this would require us to eliminate the vagueness in the definition of N), but a direct proof is simpler. Compare this with our previous example of the lambda calculus where a direct proof was not even possible using known techniques. The homomorphism (if it exists) consists of two maps h_E: $Dir_{Exp} \rightarrow US_{Exp}$ and h_N: $Dir_N \rightarrow US_N$ that satisfy the following conditions:

$h_E(const_{Dir}(n)) = const_{US}(h_N(n))$
$h_E(plus_{Dir}(e_1,e_2)) = plus_{US}(h_E(e_1),h_E(e_2))$
$h_N(zero_{Dir}) = zero_{US}$
$h_N(succ_{Dir}(n)) = succ_{US}(h_N(n))$
$h_N(sum_{Dir}(m,n))) = sum_{US}(h_N(m),h_N(n))$
$h_N(\mathcal{E}_{Dir}(e)) = \mathcal{E}_{US}(h_E(e))$

Clearly, the first five conditions are trivially satisfied by choosing h_E and h_N to be identity maps. The last condition, translated below into the notation of the semantic model definitions, requires proof by induction over the structure of expressions.

$h_N(\mathcal{E}_{Dir}(e)) = \mathcal{E}_D[\![e]\!]$ since h_N is an identity.
$\mathcal{E}_{US}(h_E(e)) = \mathcal{E}_{US}[\![e]\!]$ since h_E is an identity.
Proof that $\mathcal{E}_D[\![e]\!] = \mathcal{E}_{US}[\![e]\!]$ for all e:

<u>case 1</u>: $e = n$

$\mathcal{E}_D[\![e]\!] = n$
$\mathcal{E}_{US}[\![e]\!] = (\mathcal{E}_S[\![n]\!]\zeta)\downarrow 1 = n$

<u>case 2</u>: $e = e_1 + e_2$

$\mathcal{E}_{US}[\![e_1 + e_2]\!] = (\mathcal{E}_S[\![e_1 + e_2]\!]\zeta)\downarrow 1$
$= (add(\mathcal{E}_S[\![e_2]\!](\mathcal{E}_S[\![e_1]\!]\zeta)))\downarrow 1$ definition $\mathcal{E}_S$
$= (add(\mathcal{E}_D[\![e_2]\!]\ \mathit{cat}(\mathcal{E}_S[\![e_1]\!]\zeta)))\downarrow 1$ inductive hypothesis
$= (add(\mathcal{E}_D[\![e_2]\!]\ \mathit{cat}(\mathcal{E}_D[\![e_1]\!]\ \mathit{cat}\ \zeta))\downarrow 1$ inductive hypothesis
$= ((\mathcal{E}_D[\![e_2]\!] + \mathcal{E}_D[\![e_1]\!])\ \mathit{cat}\ \zeta)\downarrow 1$ definition add
$= (\mathcal{E}_D[\![e_1 + e_2]\!]\ \mathit{cat}\ \zeta)\downarrow 1$ definition E_D
$= \mathcal{E}_D[\![e_1 + e_2]\!]$

Thus we have completed our proof that Dir and Stk are congruent semantic models by showing that there is a theory morphism σ and a homomorphism h: Dir $\rightarrow U_\sigma$(Stk) (which is one-one by virtue of the fact that h_E and h_N are identities, and may be onto depending on the exact definition of the $\mathbb{N}$ domains).

4.4.4 Comparison with the Usual Style of Proof

It is interesting to compare the proof we have detailed in this section with the more traditional style of proof suggested by Stoy (1977). The obvious approach is to establish $\mathcal{E}_D[\![e]\!] = (\mathcal{E}_S[\![e]\!]\zeta)\downarrow 1$ by structural induction over e, which is in fact exactly what we did for the very last part of our proof above.

The question naturally arises why all the rest was necessary: defining presentations D and S aand showing Dir and Stk to be models; defining σ and showing it to be a theory morphism; deriving $U_\sigma(\mathrm{Stk})$; and finally showing that there is a homomorphism Dir $\rightarrow U_\sigma(\mathrm{Stk})$, being the only part having some correspondence with the non-algebraic proof. Part of the answer lies in the fact that we have proved something *about* the relationship $\mathcal{E}_D[\![e]\!] = (\mathcal{E}_S[\![e]\!]\zeta)\downarrow 1$ beyond merely showing it to be true. Since we have adopted a precisely defined notion of congruence in terms of theory morphisms and homomorphisms we must work within that framework. In other words we need to show that the relationship $\mathcal{E}_D[\![e]\!] = (\mathcal{E}_S[\![e]\!]\zeta)\downarrow 1$ *is* a congruence and this is where most of the effort in this section was concentrated.

It is important to note that in this case and indeed every case where the homomorphism can be directly dealt with (unlike the lambda calculus example), the proof can be greatly simplified by a different choice of semantic presentations. If we review the proof in this section it is clear that most of the work goes in showing that the equations of the theories are satisfied in various circumstances yet the final crux of the proof, that $\mathcal{E}_D[\![e]\!] = (\mathcal{E}_S[\![e]\!]\zeta)\downarrow 1$, makes no use of the equations at all. If, instead of the presentations D and S we had given only their respective signatures (call them Δ and Σ) without any equations we could have followed the same style of proof with much less effort. First, to show that Dir and Stk are respectively models of Th_Δ and Th_Σ we need only show that they have the appropriate signatures. Second, given that σ is a signature morphism, it is immediately a theory morphism $\mathrm{Th}_\Delta \rightarrow \mathrm{Th}_\Sigma$ by virtue of the fact that there are no (non-trivial) equations in the theories. Finally, showing that there is a Δ-homomorphism Dir $\rightarrow U_\sigma(\mathrm{Stk})$ is exactly the same as the non-algebraic structural induction proof.

Thus, for cases where the homomorphic relationship can be directly established without redress to initiality results the most sensible choice of semantic presentation is simply a signature Ω and the theory represented is the free theory Th_Ω. By making such a choice the work involved in the algebraic proof is exactly the same as that required to establish the congruence result by traditional techniques, so our approach does not in fact suffer on the grounds of practicality or effort required in comparison with the traditional, less-structured one.

4.5 DEVIL Example

In this section we treat a more realistic example with the intention of consolidating the work already presented. A subsidiary objective is to demonstrate the facility and expressive power of using algebraic presentations as semantic definitions, a theme discussed in Chapter 3. The language is DEVIL (Henson & Turner, 1982) though we have eliminated **goto**'s and labels since they add length to the semantic definition and proofs without adding any extra interest. DEVIL in turn was based on the language used by Strachey & Wadsworth (1974) to introduce continuation semantics.

The language was chosen by Henson & Turner because it contained "most of the features which force a wedge between denotational and operational definitions". We choose it for much the same reason but also because Henson & Turner's so-called *completion* semantics offer an interesting and unusual style of operational semantics. Further, the notation used for completion semantics is much more directly amenable to our algebraic treatment than the more usual "interpreter definition" style of operational semantics (e.g. Stoy, 1977, 1981). The abstract syntax for DEVIL is as follows:

c: Com
e: Exp
d: Dec
c:: = **dummy** | c; c | i: = e | **call** e | **resultis** e | **if** e **then** c **else** c
| **while** e **do** c | **begin** d; c **end**
e:: = i | **true** | **false** | **if** e **then** e **else** e | **valof** c | **proc** c
d:: = **var** i | d; d

The existence of result blocks (**valof** c, **resultis** e) is enough to make a direct style of denotational semantics inadequate to describe DEVIL.

4.5.1 The Presentations and Models

We begin by detailing the denotational semantic definition of DEVIL. We offer no commentary other than to note that it is a continuation semantics and refer the reader to Strachey & Wadsworth (1974) if necessary. The corresponding algebra wil be called Cont.

Semantic Domains

$T = \{true, false\}$	truth values
L	locations
$\nu: E = T + F$	expressible values
$\delta: D = L + K$	denotable values
$\nu: V = T + F$	storable values
$\sigma: S = [L \to V]$	stores
$\rho: U = [Ide \to D] \times K$	environments
$\theta: C = [S \to S]$	command continuations
$\kappa: K = [E \to C]$	expression continuations
$\phi: F = [C \to C]$	function closures

Semantic Functions

$\mathcal{D}$: $Dec \to U \to S \to [U \times S]$

$\mathcal{C}$: $Com \to U \to C \to C$

$\mathcal{E}$: $Exp \to U \to K \to C$

1. $\mathcal{D}[\![\textbf{var } i]\!]\rho\sigma = \langle \rho[i/new\ \sigma], \sigma[new\ \sigma/?]\rangle$
2. $\mathcal{D}[\![d_1; d_2]\!]\rho\sigma = \mathcal{D}[\![d_2]\!]\rho'\sigma'$

   where $\mathcal{D}[\![d_1]\!]\rho\sigma = \langle\rho',\sigma'\rangle$
3. $\mathcal{C}[\![\textbf{dummy}]\!]\rho\theta = \theta$
4. $\mathcal{C}[\![c_0; c_1]\!]\rho\theta = \mathcal{C}[\![c_0]\!]\rho(\mathcal{C}[\![c_1]\!]\rho\theta)$
5. $\mathcal{C}[\![i{:=}e]\!]\rho\theta = \mathcal{E}[\![e]\!]\rho(update(\rho[\![i]\!])\theta)$
6. $\mathcal{C}[\![\textbf{call } e]\!]\rho\theta = \mathcal{E}[\![e]\!]\rho(call\theta)$
7. $\mathcal{C}[\![\textbf{resultis } e]\!]\rho\theta = \mathcal{E}[\![e]\!]\rho(\rho{\downarrow}2)$
8. $\mathcal{C}[\![\textbf{if } e \textbf{ then } c_0 \textbf{ else } c_1]\!]\rho\theta = \mathcal{E}[\![e]\!]\rho(cond(\mathcal{C}[\![c_0]\!]\rho\theta, \mathcal{C}[\![c_1]\!]\rho\theta))$
9. $\mathcal{C}[\![\textbf{while } e \textbf{ do } c]\!]\rho\theta = fix[\lambda\theta'. \mathcal{E}[\![e]\!]\rho(cond(\mathcal{C}[\![c]\!]\rho\theta',\theta))]$
10. $\mathcal{C}[\![\textbf{begin } d; c \textbf{ end}]\!]\rho\theta\sigma = \mathcal{C}[\![c]\!]\rho'\theta\sigma'$

    where $\mathcal{D}[\![d]\!]\rho\sigma = \langle\rho',\sigma'\rangle$
11. $\mathcal{E}[\![i]\!]\rho\kappa\sigma = \kappa(\sigma(\rho[\![i]\!]|L))$
12. $\mathcal{E}[\![\textbf{true}]\!]\rho\kappa = \kappa(true)$
13. $\mathcal{E}[\![\textbf{false}]\!]\rho\kappa = \kappa(false)$
14. $\mathcal{E}[\![\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2]\!]\rho\kappa = \mathcal{E}[\![e_0]\!]\rho(cond(\mathcal{E}[\![e_1]\!]\rho\kappa, \mathcal{E}[\![e_2]\!]\rho\kappa))$
15. $\mathcal{E}[\![\textbf{valof } c]\!]\rho\kappa = \mathcal{C}[\![c]\!]\langle\rho{\downarrow}1,\kappa\rangle(?)$
16. $\mathcal{E}[\![\textbf{proc } c]\!]\rho\kappa = \kappa(\mathcal{C}[\![c]\!]\rho)$

Auxiliary Functions

new σ is a location unused in σ.

update $d\theta = \lambda\nu\sigma.\ d \in L \to \theta(\sigma[d|L/\nu]);\ ?$

call $\theta = \lambda\nu.\ \nu \in F \rightarrow (\nu|F)\theta\ ;\ ?$
cond $(\theta_0,\theta_1) = \lambda\nu.\nu \in T \rightarrow (\nu|T \rightarrow \theta_0; \theta_1); ?$

<u>Cont - continuation semantics for DEVIL</u>

The completion semantics correspond closely to the continuation semantics above. In fact the correspondence is so strong (compare the domain definitions) that Henson & Turner(1982) are moved to argue that completion semantics should be viewed as the *standard* operational semantics. An important point to note is that the semantic definition that follows is *first-order* in that the domain equations involve only the domain constructors "+", "x" (and "*") but not "→". Jumps present something of a problem in this context and Henson & Turner offer "completions" as a possible solution. These are data items (consisting in part of pieces of program text) that directly represent continuations.

A more detailed comparison of the domain equations of the two semantic definitions is worthwhile. Notice firstly that the basic domains of expressible, denotable and storable values have very similar structures. Next, the store and environment in the completion semantics are just the usual list-of-pairs representations of the mapping functions of the denotational definition. For the operational definition of F the domain of procedure values, the obvious representation is a *closure*, directly analogous to the corresponding aspect of the operational semantics of the lambda calculus (§4.2.1).

Command completions and expression completions correspond to command continuations and expression continuations respectively. Command completions may have the form $\langle text, \rho, \theta\rangle \in F \times C$ or $\langle text, \rho, \kappa\rangle \in Exp \times U \times K$ or $\langle\nu, \kappa\rangle \in E \times K$. With the first and second form we have some text to evaluate, an environment to evaluate it in and a completion to evaluate next. The third form reflects the fact that expression completions represent expression continuations $K = [E \rightarrow C]$, so given an expressible value ν and an expression completion κ, $\langle\nu,\kappa\rangle$ is the command completion corresponding to the "application" of κ to ν. Naturally in all cases in the operational semantics a store must be provided before any computation can proceed.

Expression completions take one of three main forms, in which the words "update", "call" and "cond" are used as an aid to readability and to correspond to the auxiliary functions of Cont but contain no actual information. By providing them with an expressible value and a store, an intuitive (and operational) explanation of the various forms is possible. First, run(send(⟨update,d,θ⟩,ν), σ) has the effect of assigning ν to d in σ and then running θ on that new store. Next, run(send(⟨call,θ⟩,ν), σ) expects ν to be a closure ⟨c,ρ⟩ so the effect is to execute c with the parameters ρ,θ and σ; i.e. $\mathcal{C}[\![c]\!]\rho\theta\sigma$. Finally run(send(⟨cond,θ_1,θ_2⟩,ν), σ) expects ν to be a truth value and accordingly chooses one of θ_1 or θ_2 to be run on σ. Thus the whole philosophy behind completion semantics is to offer a textual (and therefore referentially opaque) representation of the higher-order concepts of continuation semantics.

<u>Semantic Domains</u>

$T = \{\text{true}, \text{false}\}$	truth values
ℓ: L	locations
$\nu: E = T + F + \{?\}$	expressible values
$\delta: D = L + K + \{?\}$	denotable values
$\nu: V = T + F + \{?\}$	storable values
$\sigma: S = [L \times V]^* + \{?\}$	stores
$\rho: U = [Ide \times D]^* \times K$	environments
$\theta: C = [F \times C] +$	
$[Exp \times U \times K] +$	
$[E \times K] +$	
$\{fail\} + \{final\}$	command completions
$\kappa: K = [\{update\} \times D \times C] +$	
$[\{call\} \times C] +$	
$[\{cond\} \times C \times C] + \{?\}$	expression completions
$\phi: F = [Com \times U]$	command closures

<u>Semantic Functions</u>

$\mathcal{D}: Dec \to U \to S \to [U \times S]$

$\mathcal{C}: Com \to U \to C \to S \to S$

$\mathcal{E}: Exp \to U \to K \to S \to S$

1. $\mathcal{D}[\![\textbf{var}\ i]\!]\rho\sigma = \langle bind(\rho,i,new(\sigma)), set(\sigma,new(\sigma),?)\rangle$
2. $\mathcal{D}[\![d_1; d_2]\!]\rho\sigma = \mathcal{D}[\![d_2]\!]\rho'\sigma'$

   where $\mathcal{D}[\![d_1]\!]\rho\sigma = \langle\rho',\sigma'\rangle$

   $= \mathcal{D}[\![d_2]\!]((\mathcal{D}[\![d_1]\!]\rho\sigma)\downarrow 1)((\mathcal{D}[\![d_1]\!]\rho\sigma)\downarrow 2)$

3. $\mathcal{C}[\![\textbf{dummy}]\!]\rho\theta\sigma = run(\theta,\sigma)$
4. $\mathcal{C}[\![c_0;\ c_1]\!]\rho\theta\sigma = \mathcal{C}[\![c_0]\!]\rho(\langle c_1,\rho\rangle,\theta)\sigma$
5. $\mathcal{C}[\![i := e]\!]\rho\theta\sigma = \mathcal{E}[\![e]\!]\rho(update, lookup(\rho,i),\theta)\sigma$
6. $\mathcal{C}[\![\textbf{call}\ e]\!]\rho\theta\sigma = \mathcal{E}[\![e]\!]\rho(call,\theta)\sigma$
7. $\mathcal{C}[\![\textbf{resultis}\ e]\!]\rho\theta\sigma = \mathcal{E}[\![e]\!]\rho(res(\rho))\sigma$
8. $\mathcal{C}[\![\textbf{if}\ e\ \textbf{then}\ c_0\ \textbf{else}\ c_1]\!]\rho\theta\sigma = \mathcal{E}[\![e]\!]\rho(cond,\langle\langle c_0,\rho\rangle,\theta\rangle,\langle\langle c_1,\rho\rangle,\theta\rangle)\sigma$
9. $\mathcal{C}[\![\textbf{while}\ e\ \textbf{do}\ c]\!]\rho\theta\sigma = run(afix[\lambda\theta'.(e,\rho,\langle cond,\langle\langle c,\rho\rangle,\theta'\rangle,\theta\rangle)],\sigma)$
10. $\mathcal{C}[\![\textbf{begin}\ d;\ c\ \textbf{end}]\!]\rho\theta\sigma = \mathcal{C}[\![c]\!]\rho'\theta\sigma'$
    where $\mathcal{D}[\![d]\!]\rho\sigma = \langle\rho',\sigma'\rangle$
11. $\mathcal{E}[\![i]\!]\rho\kappa\sigma = run(send(\kappa,deref(lookup(\rho,i),\sigma)),\sigma)$
12. $\mathcal{E}[\![\textbf{true}]\!]\rho\kappa\sigma = run(send(\kappa,true),\sigma)$
13. $\mathcal{E}[\![\textbf{false}]\!]\rho\kappa\sigma = run(send(\kappa,false),\sigma)$
14. $\mathcal{E}[\![\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2]\!]\rho\kappa\sigma = \mathcal{E}[\![e_0]\!]\rho(cond,\langle e_1,\rho,\kappa\rangle,\langle e_2,\rho,\kappa\rangle)\sigma$
15. $\mathcal{E}[\![\textbf{valof}\ c]\!]\rho\kappa\sigma = \mathcal{C}[\![c]\!]bindres(\rho,\kappa)(fail)\sigma$
16. $\mathcal{E}[\![\textbf{proc}\ c]\!]\rho\kappa\sigma = run(send(\kappa,\langle c,\rho\rangle),\sigma)$

<u>Auxiliary Functions</u>

$$map(\sigma,\ell) = \begin{cases} 2nd(\sigma{\downarrow}n) & \text{if } 1st(\sigma{\downarrow}n) = \ell \text{ and } \forall\ m<n,\ 1st(\sigma{\downarrow}m) \neq \ell \\ ? & \text{otherwise} \end{cases}$$

$$deref(\delta,\sigma) = \begin{cases} map(\sigma,\delta|L) & \text{if } \delta \in L \\ ? & \text{otherwise} \end{cases}$$

$set(\sigma,\ell,\nu) = \langle\ell,\nu\rangle\ \textit{cat}\ \sigma$

$new(\sigma)$ is a location unused in σ.

$bind(\rho,i,\delta) = \langle\langle i,\delta\rangle\ \textit{cat}\ (\rho{\downarrow}1),\ \rho{\downarrow}2\rangle$

$bindres(\rho,\kappa) = \langle\rho{\downarrow}1,\kappa\rangle$

$$lookup(\rho,i) = \begin{cases} 2nd(\rho{\downarrow}1{\downarrow}n) & \text{if } 1st(\rho{\downarrow}1{\downarrow}n) = i \text{ and } \forall\ m<n, 1st(\rho{\downarrow}1{\downarrow}m) \neq i \\ ? & \text{otherwise} \end{cases}$$

$res(\rho) = \rho{\downarrow}2$

$send(\kappa,\nu) = \langle\nu,\kappa\rangle$

$run(fail,\sigma) = ?$

$run(final,\sigma) = \sigma$

$run((\langle c,\rho\rangle,\theta),\sigma) = \mathcal{C}[\![c]\!]\rho\theta\sigma$

$run((e,\rho,\kappa),\sigma) = \mathcal{E}[\![e]\!]\rho\kappa\sigma$

$$\mathrm{run}(\{\nu,\langle \mathrm{update},d,\theta\rangle\},\sigma) = \begin{cases} \mathrm{run}(\theta,\mathrm{set}(\sigma,d|L,\nu)) & \text{if } d \in L \\ ? & \text{otherwise} \end{cases}$$

$$\mathrm{run}(\{\nu,\langle \mathrm{call},\theta\rangle\},\sigma) = \begin{cases} \mathfrak{C}[\![c]\!]\rho\theta\sigma & \text{if } \nu \in F,\ \nu|F = \langle c,\rho\rangle \\ ? & \text{otherwise} \end{cases}$$

$$\mathrm{run}(\{\nu,\langle \mathrm{cond},\theta_1,\theta_2\rangle\},\sigma) = \begin{cases} \nu|T \rightarrow \mathrm{run}(\theta_1,\sigma);\ \mathrm{run}(\theta_2,\sigma) & \text{if } \nu \in T \\ ? & \text{otherwise} \end{cases}$$

<u>Comp - completion semantics for DEVIL</u>

Notice that the clause for $\mathfrak{C}[\![$**while** e **do** c$]\!]$ involves an infinite data structure by virtue of the use of afix. Of course, Henson &Turner used the usual fix operator, but in this case the result is the same. There is in fact a finite alternative that is no more difficult for us to treat than the above version, and it will be briefly discussed later.

Following our discussion in §4.3, since we intend to show Comp to be initial in Alg_{DA} presented below it is convenient (though not absolutely necessary, see §4.3) to use our initial algebra fixed-point construction for the completion semantics.

One final point is that we have been quite explicit as to which of the domains have an error element,?. Again this is because we intend to establish the initiality of Comp in Alg_{DA} and we are necessarily explicit about error constants in the presentation DA. Only those sorts that require error values were given them but it would present no difficulty to have an error constant of every sort if it was thought desirable to have an error element in every domain of Comp.

Since we have used the same names for many of the functions and domains in the two semantic definitions, we shall resolve any confusion that may arise by decorating those from Comp with a subscript "o" (for operational) e.g. $\mathfrak{C}_o$, $\mathfrak{E}_o$ and those from Cont with a subscript "d" (for denotational) e.g. $\mathfrak{C}_d$, $\mathfrak{E}_d$.

We now detail the presentation DA of which we will show Comp to be a model (the initial one) and DB of which we will show Cont to be a model. Rather than explicitly including the syntactic sorts we dispense with such details and use the notation of the abstract syntax. The translation from one notation to the other is purely mechanical (Goguen et al, 1977).

<u>Signature</u>

syntactic sorts as for abstract syntax of DEVIL.

sort Bool

    tt, ff: $\rightarrow$ Bool

sort Val

    injB: Bool $\rightarrow$ Val
    injA: Abstr $\rightarrow$ Val
    errV: $\rightarrow$ Val

sort Den

    injL: Loc $\rightarrow$ Den
    injK: Kont $\rightarrow$ Den
    errD: $\rightarrow$ Den

sort Store

    empty: $\rightarrow$ Store
    set: Store $\times$ Loc $\times$ Val $\rightarrow$ Store
    contents: Store $\times$ Den $\rightarrow$ Val
    errS: $\rightarrow$ Store

sort Loc

    new: Store $\rightarrow$ Loc

sort Env

    arid: $\rightarrow$ Env
    bind: Env $\times$ Ide $\times$ Den $\rightarrow$ Env
    find: Env $\times$ Ide $\rightarrow$ Den
    bindres: Env $\times$ Kont $\rightarrow$ Env
    res: Env $\rightarrow$ Kont

sort Modif

    null: $\rightarrow$ Modif
    fail: $\rightarrow$ Modif
    apply: Modif $\times$ Store $\rightarrow$ Store
    klo: Exp $\times$ Env $\times$ Kont $\rightarrow$ Modif
    loop: Exp $\times$ Com $\times$ Env $\times$ Modif $\rightarrow$ Modif

sort Kont

update: Den $\times$ Modif $\rightarrow$ Kont

call: Modif $\rightarrow$ Kont

cond: Modif $\times$ Modif $\rightarrow$ Kont

kontin: Kont $\times$ Val $\rightarrow$ Modif

errK: $\rightarrow$ Kont

sort Abstr

clo: Com $\times$ Env $\rightarrow$ Abstr

contin: Abstr $\times$ Modif $\rightarrow$ Modif

D1 : Dec $\times$ Env $\times$ Store $\rightarrow$ Env

D2 : Dec $\times$ Env $\times$ Store $\rightarrow$ Store

C: Com $\times$ Env $\times$ Modif $\times$ Store $\rightarrow$ Store

E: Exp $\times$ Env $\times$ Kont $\times$ Store $\rightarrow$ Store

<u>Equations</u>

1. contents(empty,d) = errV
2. contents(set(σ,ℓ_1,ν),injL(ℓ_2)) = *if* $\ell_1 = \ell_2$ *then* ν *else* contents(σ,injL(ℓ_2))
3. contents(set(σ,ℓ,ν),injK(κ)) = errV
4. contents(set(σ,ℓ,ν),errD) = errV
5. contents(errS,δ) = errV
6. find(arid,i) = errD
7. find(bind(ρ,i,δ),j) = *if* i=j *then* δ *else* find(ρ,j)
8. find(bindres(ρ,κ),i) = find(ρ,i)
9. res(arid) = errK
10. res(bind(ρ,i,d)) = res(ρ)
11. res(bindres(ρ,κ)) = κ
12. apply(null,σ) = σ
13. apply(fail,σ) = errS
14. apply(klo(e,ρ,κ),σ) = *E*(e, ρ,κ,σ)
15. apply(contin(clo(c,ρ),θ),σ) = *C*(c, ρ,θ,σ)
16. apply(kontin(cond(θ_1,θ_2),injB(tt)),σ) = apply(θ_1,σ)
17. apply(kontin(cond(θ_1,θ_2),injB(ff)),σ) = apply(θ_2,σ)
18. apply(kontin(cond(θ_1,θ_2),injA(a)),σ) = errS
19. apply(kontin(cond(θ_1,θ_2),errV),σ) = errS
20. apply(kontin(call(θ),injB(b)),σ) = errS
21. apply(kontin(call(θ),injA(a)),σ) = apply(contin(a,θ),σ)
22. apply(kontin(call(θ),errV),σ) = errS
23. apply(kontin(update(injL(ℓ),θ),ν),σ) = apply(θ,set(σ,ℓ,ν))

24. apply(kontin(update(injK(κ),θ),ν),σ) = errS
25. apply(kontin(update(errD,θ),ν),σ) = errS
26. apply(kontin(errK,ν),σ) = errS
27. loop(e,c,ρ,θ) = klo(e,ρ,cond(contin(clo(c,ρ),loop(e,c,ρ,θ)),θ))
28. *D1* (**var** i, ρ,σ) = bind(ρ,i,new(σ))
29. *D2*(**var** i, ρ,σ) = set(σ,new(σ),errV)
30. *D1*(d_1; d_2, ρ,σ) = *D1*(d_2, *D1*(d_1, ρ,σ),*D2*(d_1, ρ,σ))
31. *D2*(d_1; d_2, ρ,σ) = *D2*(d_2, *D1*(d_1, ρ,σ),*D2*(d_1, ρ,σ))
32. *C*(**dummy**, ρ,θ,σ) = apply(θ,σ)
33. *C*(c_1; c_2, ρ,θ,σ) = *C*(c_1, ρ,contin(clo(c_2,ρ),θ),σ)
34. *C*(i:=e, ρ,θ,σ) =*E*(e, ρ,update(find(ρ,i),θ),σ)
35. *C*(**call** e, ρ,θ,σ) = *E*(e, ρ,call(θ),σ)
36. *C*(**resultis** e, ρ,θ,σ) = *E*(e, ρ,res(ρ),σ)
37. *C*(**if** e **then** c_1 **else** c_2, ρ,θ,σ) = *E*(e, ρ,cond(contin(clo(c_1,ρ),θ),
    contin(clo(c_2,ρ),θ)),σ)
38. *C*(**while** e **do** c, ρ,θ,σ) = apply(loop(e,c,ρ,θ),σ)
39. *C*(**begin** d; c **end**, ρ,θ,σ) = *C*(c, *D1*(d,ρ,σ),θ,*D2*(d,ρ,σ))
40. *E*(i, ρ,κ,σ) = apply(kontin(κ,contents(σ,find(ρ,i))),σ)
41. *E*(**true**, ρ,κ,σ) = apply(kontin(κ,injB(tt)),σ)
42. *E*(**false**, ρ,κ,σ) = apply(kontin(κ,injB(ff)),σ)
43. *E*(**if** e_0 **then** e_1 **else** e_2, ρ,κ,σ) = *E*(e_0, ρ,cond(klo(e_1,ρ,κ),
    klo(e_2,ρ,κ)),σ)
44. *E*(**valof** c, ρ,κ,σ) = *C*(c, bindres(ρ,κ),fail,σ)
45. *E*(**proc** c, ρ,κ,σ) =apply(kontin(κ,injA(clo(c,ρ))),σ)

<u>DA - semantic presentation for DEVIL</u>

<u>Signature</u>

syntactic sorts as for abstract syntax of DEVIL.

sort Bool

  tt, ff: $\rightarrow$ Bool

sort Val

  injB: Bool $\rightarrow$ Val

  injA: Abstr $\rightarrow$ Val

  errV: $\rightarrow$ Val

sort Den

  injL: Loc $\rightarrow$ Den

  injK: Kont $\rightarrow$ Den

  errD: $\rightarrow$ Den

sort Store

  empty: $\rightarrow$ Store

  set: Store $\times$ Loc $\times$ Val $\rightarrow$ Store

  contents: Store $\times$ Den $\rightarrow$ Val

  errS: $\rightarrow$ Store

sort Loc

  new: Store $\rightarrow$ Loc

sort Env

  arid: $\rightarrow$ Env

  bind: Env $\times$ Ide $\times$ Den $\rightarrow$ Env

  find: Env $\times$ Ide $\rightarrow$ Den

  bindres: Env $\times$ Kont $\rightarrow$ Env

  res: Env $\rightarrow$ Kont

sort Modif

  null: $\rightarrow$ Modif

  fail: $\rightarrow$ Modif

  apply: Modif $\times$ Store $\rightarrow$ Store

sort Kont

  update: Den $\times$ Modif $\rightarrow$ Kont

  call: Modif $\rightarrow$ Kont

  cond: Modif $\times$ Modif $\rightarrow$ Kont

  Kontin: Kont $\times$ Val $\rightarrow$ Modif

  errK: $\rightarrow$ Kont

sort Abstr

  clo: Com $\times$ Env $\rightarrow$ Abstr

  contin: Abstr $\times$ Modif $\rightarrow$ Modif

D1 : Dec × Env × Store → Env
D2 : Dec × Env × Store → Store
C: Com × Env × Modif → Modif
E: Exp × Env × Kont → Modif

<u>Equations</u>

1. contents(empty,δ) = errV
2. contents(set(σ,ℓ_1,ν),injL(ℓ_2)) = *if* ℓ_1 = ℓ_2 *then* ν *else* contents(σ,injL(ℓ_2))
3. contents(set(σ,ℓ,ν),injK(κ)) = errV
4. contents(set(σ,ℓ,ν),errD) = errV
5. contents(errS,δ) = errV
6. find(arid,i) = errD
7. find(bind(ρ,i,δ),j) = *if* i=j *then* δ *else* find(ρ,j)
8. find(bindres(ρ,κ),i) = find(ρ,i)
9. res(arid) = errK
10. res(bind(ρ,i,δ)) = res(ρ)
11. res(bindres(ρ,κ)) = κ
12. apply(null,σ) = σ
13. apply(fail,σ) = errS
14. contin(clo(c,ρ),θ) = *C*(c, ρ,θ)
15. kontin(cond(θ_1,θ_2),injB(tt)) = θ_1
16. kontin(cond(θ_1,θ_2),injB(ff)) = θ_2
17. kontin(cond(θ_1,θ_2),injA(a)) = fail
18. kontin(cond(θ_1,θ_2),errV) = fail
19. kontin(call(θ),injB(b)) = fail
20. kontin(call(θ),injA(a)) = contin(a,θ)
21. kontin(call(θ),errV) = fail
22. apply(kontin(update(injL(ℓ),θ),ν),σ) = apply(θ,set(σ,ℓ,ν))
23. kontin(update(injK(κ),θ),ν) = fail
24. kontin(update(errD,θ),ν) = fail
25. kontin(errK,ν) = fail
26. *D1* (**var** i, ρ,σ) = bind(ρ,i,new(σ))
27. *D2* (**var** i, ρ,σ) = set(σ,new(σ),errV)
28. *D1* (d_1; d_2, ρ,σ) = *D1* (d_2, *D1* (d_1, ρ,σ),*D2* (d_1, ρ,σ))
29. *D2*(d_1; d_2, ρ,σ) = *D2*(d_2, *D1* (d_1, ρ,σ),*D2* (d_1, ρ,σ))
30. *C*(**dummy**, ρ,θ) = θ
31. *C*(c_1; c_2, ρ,θ) = *C*(c_1, ρ, *C*(c_2, ρ,θ))
32. *C*(i: = e, ρ,θ) = *E*(e, ρ,update(find(ρ,i),θ))

33. $\mathcal{C}$(**call** e, ρ,θ) = $\mathcal{E}$(e, ρ,call(θ))
34. $\mathcal{C}$(**resultis** e, ρ,θ) = $\mathcal{E}$(e, ρ,res(ρ))
35. $\mathcal{C}$(**if** e **then** c_1 **else** c_2, ρ,θ) = $\mathcal{E}$(e, ρ,cond($\mathcal{C}$(c_1, ρ,θ), $\mathcal{C}$(c_2, ρ,θ)))
36. $\mathcal{C}$(**while** e **do** c, ρ,θ) = $\mathcal{E}$(e, ρ,cond($\mathcal{C}$(c, ρ, $\mathcal{C}$(**while** e **do** c, ρ,θ)),θ))
37. apply($\mathcal{C}$(**begin** d; c **end**, ρ,θ),σ) = apply($\mathcal{C}$(c, $\mathcal{D}1$ (d, ρ,σ),θ), $\mathcal{D}2$(d, ρ,σ))
38. apply($\mathcal{E}$(i, ρ,κ),σ) = apply(kontin(κ,contents(σ,find(ρ,i))),σ)
39. $\mathcal{E}$(**true**, ρ,κ) = kontin(κ, injB(tt))
40. $\mathcal{E}$(**false**,ρ,κ) = kontin (κ, injB(ff))
41. $\mathcal{E}$(**if** e_0 **then** e_1 **else** e_2, ρ,κ) = $\mathcal{E}$(e_0, ρ,cond($\mathcal{E}$(e_1, ρ,κ),$\mathcal{E}$(e_2, ρ,κ)))
42. $\mathcal{E}$(**valof** c, ρ,κ) = $\mathcal{C}$(c, bindres(ρ,κ),fail)
43. $\mathcal{E}$(**proc** c, ρ,κ) = kontin(κ,injA(clo(c,ρ)))

DB - semantic presentation for DEVIL

As perhaps would be expected given the many similarities in the completion and continuation semantics, the two presentations DA and DB have much in common.

4.5.2 Initiality of the Completion Semantics

In this section we show that Comp is an initial model of DA and that Cont is a model of DB. To establish the congruence of Comp and Cont we then only need to prove that there is a theory morphism from DA to DB. To show that Comp and Cont are respectively models of DA and DB we need to specify a carrier from the algebra for each sort and a function for each operator symbol of the presentation and then show that the equations of the presentation are satisfied. Again, this is a very straightforward task and the details follow. Since the equations are all very simply satisfied (and there are a large number of them) we dispense with those proofs and provide only the signature correspondences, DA with Comp and DB with Cont and the equations of the presentations rewritten in the notation of the models.

<u>Comp carriers for DA sorts</u>

Bool:	$T = \{true, false\}$
Loc:	L
Val:	$E = V = T + F + \{?\}$
Den:	$D = L + K + \{?\}$
Store:	$S = [L \times V]^* + \{?\}$
Env:	$U = [Ide \times D]^* \times K$
Modif:	$C = [F \times C] + [Exp \times U \times K] + [E \times K] + \{fail\} + \{final\}$
Kont:	$K = [\{update\} \times D \times C] + [\{call\} \times C\,] + [\{cond\} \times C \times C] + \{?\}$
Abstr:	$F = [Com \times U]$

<u>Comp functions for DA operators</u>

<u>DA</u>	<u>Comp</u>
tt: → Bool	true
ff: → Bool	false
injB: Bool → Val	$\lambda\tau.\ \tau$ *in* V
injA: Abstr→ Val	$\lambda\phi.\ \phi$ *in* V
errV: → Val	$?_V$
injL : Loc → Den	$\lambda\ell.\ \ell$ *in* D
injK: Kont → Den	$\lambda\kappa.\ \kappa$ *in* D
errD: → Den	$?_D$
empty: → Store	$\langle\rangle$
set: Store × Loc × Val → Store	set
contents: Store × Den → Val	deref

errS: → Store	$?_S$
new: Store → Loc	new
null: → Modif	final
fail: → Modif	fail
apply: Modif × Store → Store	run
klo: Exp × Env × Kont → Modif	$\lambda e\rho\kappa.\ \langle e,\rho,\kappa\rangle$
loop: Exp × Com × Env × Modif → Modif	
	$\lambda ec\rho\theta.\ \mathrm{afix}[\lambda\theta'.\langle e,\rho,\langle \mathrm{cond},\langle\langle c,\rho\rangle,\theta'\rangle,\theta\rangle\rangle]$
update: Den × Modif → Kont	$\lambda\delta\theta.\ \langle \mathrm{update},\delta,\theta\rangle$
call: Modif → Kont	$\lambda\theta.\ \langle \mathrm{call},\theta\rangle$
cond: Modif × Modif → Kont	$\lambda\theta_1\theta_2.\langle \mathrm{cond},\theta_1,\theta_2\rangle$
kontin: Kont × Val → Modif	send
errK: → Kont	$?_K$
clo: Com × Env → Abstr	$\lambda c\rho.\ \langle [\![c]\!],\rho\rangle$
contin: Abstr × Modif → Modif	$\lambda\phi\theta.\ \langle\phi,\theta\rangle$
arid: → Env	$\langle\langle\rangle,?_K\rangle$
bind: Env × Ide × Den → Env	bind
find: Env × Ide → Den	lookup
bindres: Env × Kont → Env	bindres
res: Env → Kont	res
D1 : Dec × Env × Store → Env	$\lambda d\rho\sigma.\ (\mathfrak{D}[\![d]\!]\rho\sigma)\downarrow 1$
D2 : Dec × Env × Store → Store	$\lambda d\rho\sigma.\ (\mathfrak{D}[\![d]\!]\rho\sigma)\downarrow 2$
C: Com × Env × Modif × Store → Store	
	$\lambda c\rho\theta\sigma.\ \mathfrak{C}[\![c]\!]\rho\theta\sigma$
E: Exp × Env × Kont × Store → Store	$\lambda e\rho\kappa\sigma.\ \mathfrak{E}[\![e]\!]\rho\kappa\sigma$

<u>DA equations for Comp</u>

1. $\mathrm{deref}(\langle\rangle,d) = ?$
2. $\mathrm{deref}(\mathrm{set}(\sigma,\ell_1,\nu),\ell_2\ \mathit{in}\ V) = \mathit{if}\ \ell_1=\ell_2\ \mathit{then}\ \nu\ \mathit{else}\ \mathrm{deref}(\sigma,\ell_2\ \mathit{in}\ V)$
3. $\mathrm{deref}(\mathrm{set}(\sigma,\ell,\nu),\kappa\ \mathit{in}\ V) = ?$
4. $\mathrm{deref}(\mathrm{set}(\sigma,\ell,\nu),?) = ?$
5. $\mathrm{deref}(?,\delta) = ?$
6. $\mathrm{lookup}(\langle\langle\rangle,?\rangle,i) = ?$
7. $\mathrm{lookup}(\mathrm{bind}(\rho,i,\delta),j) = \mathit{if}\ i=j\ \mathit{then}\ \delta\ \mathit{else}\ \mathrm{lookup}(\rho,j)$
8. $\mathrm{lookup}(\mathrm{bindres}(\rho,\kappa),i) = \mathrm{lookup}(\rho,i)$
9. $\mathrm{res}(\langle\langle\rangle,?\rangle) = ?$
10. $\mathrm{res}(\mathrm{bind}(\rho,i,\delta)) = \mathrm{res}(\rho)$
11. $\mathrm{res}(\mathrm{bindres}(\rho,\kappa)) = \kappa$

12. $\mathrm{run}(\mathrm{final},\sigma) = \sigma$

13. $\mathrm{run}(\mathrm{fail},\sigma) = ?$

14. $\mathrm{run}(\langle e,\rho,\kappa\rangle,\sigma) = \mathcal{E}\llbracket e\rrbracket\rho\kappa\sigma$

15. $\mathrm{run}(\langle\langle c,\rho\rangle,\theta\rangle,\sigma\rangle = \mathcal{C}\llbracket c\rrbracket\rho\theta\sigma$

16. $\mathrm{run}(\mathrm{send}(\langle\mathrm{cond},\theta_1,\theta_2\rangle, \mathrm{true}\ \mathit{in}\ V),\sigma) = \mathrm{run}(\theta_1,\sigma)$

17. $\mathrm{run}(\mathrm{send}(\langle\mathrm{cond},\theta_1,\theta_2\rangle, \mathrm{false}\ \mathit{in}\ V),\sigma) = \mathrm{run}(\theta_2,\sigma)$

18. $\mathrm{run}(\mathrm{send}(\langle\mathrm{cond},\theta_1,\theta_2\rangle, \phi\ \mathit{in}\ V),\sigma) = ?$

19. $\mathrm{run}(\mathrm{send}(\langle\mathrm{cond}, \theta_1,\theta_2\rangle,?),\sigma) = ?$

20. $\mathrm{run}(\mathrm{send}(\langle\mathrm{call},\theta\rangle, \tau\ \mathit{in}\ V),\sigma) = ?$

21. $\mathrm{run}(\mathrm{send}(\langle\mathrm{call},\theta\rangle, \phi\ \mathit{in}\ V),\sigma) = \mathrm{run}(\langle\phi,\theta\rangle,\sigma)$

22. $\mathrm{run}(\mathrm{send}(\langle\mathrm{call},\theta\rangle,?),\sigma) = ?$

23. $\mathrm{run}(\mathrm{send}(\langle\mathrm{update}, \iota\ \mathit{in}\ D, \theta\rangle,\nu),\sigma) = \mathrm{run}(\theta,\mathrm{set}(\sigma,\iota,\nu))$

24. $\mathrm{run}(\mathrm{send}(\langle\mathrm{update}, \kappa\ \mathit{in}\ D, \theta\rangle\ \nu),\sigma) = ?$

25. $\mathrm{run}(\mathrm{send}(\langle\mathrm{update},?,\theta\rangle,\nu),\sigma) = ?$

26. $\mathrm{run}(\mathrm{send}(?,\nu),\sigma) = ?$

27. $\mathrm{afix}[\lambda\theta'.(e,\rho,(\mathrm{cond},\langle\langle c,\rho\rangle,\theta'\rangle,\theta))] =$
$(e,\rho,(\mathrm{cond},\langle\langle c,\rho\rangle,\mathrm{afix}[\lambda\theta'.(e,\rho,(\mathrm{cond},\langle\langle c,\rho\rangle,\theta'\rangle,\theta))]\rangle,\theta))$

28. $(\mathcal{D}\llbracket \mathbf{var}\ i\rrbracket\rho\sigma)\downarrow 1 = \mathrm{bind}(\rho,i,\mathrm{new}(\sigma))$

29. $(\mathcal{D}\llbracket \mathbf{var}\ i\rrbracket\rho\sigma)\downarrow 2 = \mathrm{set}(\sigma,\mathrm{new}(\sigma),?)$

30. $(\mathcal{D}\llbracket d_1;d_2\rrbracket\rho\sigma)\downarrow 1 = (\mathcal{D}\llbracket d_2\rrbracket(\mathcal{D}\llbracket d_1\rrbracket\rho\sigma)\downarrow 1(\mathcal{D}\llbracket d_1\rrbracket\rho\sigma)\downarrow 2)\downarrow 1$

31. $(\mathcal{D}\llbracket d_1;d_2\rrbracket\rho\sigma)\downarrow 2 = (\mathcal{D}\llbracket d_2\rrbracket(\mathcal{D}\llbracket d_1\rrbracket\rho\sigma)\downarrow 1(\mathcal{D}\llbracket d_1\rrbracket\rho\sigma)\downarrow 2)\downarrow 2$

32. $\mathcal{C}\llbracket \mathbf{dummy}\rrbracket\rho\theta\sigma = \sigma$

33. $\mathcal{C}\llbracket c_1;c_2\rrbracket\rho\theta\sigma = \mathcal{C}\llbracket c_1\rrbracket\rho(\langle c_2,\rho\rangle,\theta)\sigma$

34. $\mathcal{C}\llbracket i := e\rrbracket\rho\theta\sigma = \mathcal{E}\llbracket e\rrbracket\rho(\mathrm{update},\mathrm{lookup}(\rho,i),\theta)\sigma$

35. $\mathcal{C}\llbracket \mathbf{call}\ e\rrbracket\rho\theta\sigma = \mathcal{E}\llbracket e\rrbracket\rho(\mathrm{call},\theta)\sigma$

36. $\mathcal{C}\llbracket \mathbf{resultis}\ e\rrbracket\rho\theta\sigma = \mathcal{E}\llbracket e\rrbracket\rho(\mathrm{res}(\rho))\sigma$

37. $\mathcal{C}\llbracket \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\rrbracket\rho\theta\sigma = \mathcal{E}\llbracket e\rrbracket\rho(\mathrm{cond},\langle\langle c_1,\rho\rangle,\theta\rangle,\langle\langle c_2,\rho\rangle,\theta\rangle)\sigma$

38. $\mathcal{C}\llbracket \mathbf{while}\ e\ \mathbf{do}\ c\rrbracket\rho\theta\sigma = \mathcal{E}\llbracket e\rrbracket\rho(\mathrm{cond},\langle\langle c,\rho\rangle,\langle\langle \mathbf{while}\ e\ \mathbf{do}\ c, \rho\rangle,\theta\rangle\rangle,\theta)\sigma$

39. $\mathcal{C}\llbracket \mathbf{begin}\ d;\ c\ \mathbf{end}\rrbracket\rho\theta\sigma = \mathcal{C}\llbracket c\rrbracket((\mathcal{D}\llbracket d\rrbracket\rho\sigma)\downarrow 1)\theta((\mathcal{D}\llbracket d\rrbracket\rho\sigma)\downarrow 2)$

40. $\mathcal{E}\llbracket i\rrbracket\rho\kappa\sigma = \mathrm{run}(\mathrm{send}(\kappa,\mathrm{deref}(\mathrm{lookup}(\rho,i),\sigma)),\sigma)$

41. $\mathcal{E}\llbracket \mathbf{true}\rrbracket\rho\kappa\sigma = \mathrm{run}(\mathrm{send}(\kappa, \mathrm{true}\ \mathit{in}\ V),\sigma)$

42. $\mathcal{E}\llbracket \mathbf{false}\rrbracket\rho\kappa\sigma = \mathrm{run}(\mathrm{send}(\kappa, \mathrm{false}\ \mathit{in}\ V),\sigma)$

43. $\mathcal{E}\llbracket \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\rrbracket\rho\kappa\sigma = \mathcal{E}\llbracket e_0\rrbracket\rho(\mathrm{cond},\langle e_1,\rho,\kappa\rangle,\langle e_2,\rho,\kappa\rangle)\sigma$

44. $\mathcal{E}\llbracket \mathbf{valof}\ c\rrbracket\rho\kappa\sigma = \mathcal{C}\llbracket c\rrbracket(\mathrm{bindres}\ (\rho,\kappa))(\mathrm{fail})\sigma$

45. $\mathcal{E}\llbracket \mathbf{proc}\ c\rrbracket\rho\kappa\sigma = \mathrm{run}(\mathrm{send}(\kappa, \langle c,\rho\rangle\ \mathit{in}\ V),\sigma)$

<u>**Cont carriers for DB sorts**</u>

Bool:	T = {true, false}
Loc:	L
Val:	E = V = T + F + {?}
Den:	D = L + K + {?}
Env:	U = [Ide → D] × K
Modif:	C = [S → S] + {?}
Kont:	K = [E → C] + {?}
Abstr:	F = [C → C]

<u>**Cont functions for DB Operators**</u>

<u>DB</u>	<u>Cont</u>
tt: → Bool	true
ff: → Bool	false
injB: Bool → Val	$\lambda\tau.\ \tau$ *in* V
injA: Abstr → Val	$\lambda\phi.\ \phi$ *in* V
errV: → Val	$?_V$
injL: Loc → Den	$\lambda\ell.\ \ell$ *in* D
injK: Kont → Den	$\lambda\kappa.\ \kappa$ *in* D
errD: → Den	$?_D$
empty: → Store	$\lambda\ell.?_V$
set: Store × Loc × Val → Store	$\lambda\sigma\ell\nu.\ \sigma[\ell/\nu]$
contents: Store × Den → Val	$\lambda\sigma\delta.\ \sigma(\delta\vert L)$
errS: → Store	$?_S$
new: Store → Loc	new
null: → Modif	$\lambda\sigma.\sigma$
fail: → Modif	$?_C$
apply: Modif× Store → Store	$\lambda\theta\sigma.\ \theta(\sigma)$
update: Den × Modif → Kont	update
call: Modif → Kont	call
cond: Modif × Modif → Kont	cond
kontin: Kont × Val → Modif	$\lambda\kappa\nu.\ \kappa(\nu)$
errK: → Kont	$?_K$
clo: Com × Env → Abstr	$\lambda c\rho.\ \mathcal{C}[\![c]\!]\rho$
contin: Abstr × Modif → Modif	$\lambda\phi\theta.\ \phi(\theta)$
arid: → Env	$\langle\lambda i.?_D,\ ?_K\rangle$
bind: Env × Ide × Den → Env	$\lambda\rho i\delta.\ \rho[i/\delta]$
find: Env × Ide → Den	$\lambda\rho i.\ (\rho\downarrow 1)[\![i]\!]$
bindres: Env × Kont → Env	$\lambda\rho\kappa.\ \langle\rho\downarrow 1,\kappa\rangle$

res: Env → Kont	$\lambda\rho.\rho{\downarrow}2$
$D1$: Dec × Env × Store → Env	$\lambda d\rho\sigma.\ (\mathfrak{D}\llbracket d\rrbracket\rho\sigma){\downarrow}1$
$D2$: Dec × Env × Store → Store	$\lambda d\rho\sigma.\ (\mathfrak{D}\llbracket d\rrbracket\rho\sigma){\downarrow}2$
C: Com × Env × Modif →Modif	$\lambda c\rho\theta.\ \mathfrak{C}\llbracket c\rrbracket\rho\theta$
E: Exp × Env × Kont → Modif	$\lambda e\rho\kappa.\ \mathfrak{E}\llbracket e\rrbracket\rho\kappa$

<u>DB equations for Cont</u>

1. $(\lambda\ell.?)(d|L) = ?$
2. $(\sigma[\ell_1/\nu])((\ell_2\ \mathit{in}\,D)|L) = \mathit{if}\ \ell_1 = \ell_2\ \mathit{then}\ \nu\ \mathit{else}\ \sigma((\ell_2\,\mathit{in}\ D)|L)$
3. $(\sigma[\ell/\nu])((\kappa\ \mathit{in}\,D)|L) = ?$
4. $(\sigma[\ell/\nu])(?|L) = ?$
5. $?(\delta|L) = ?$
6. $(\langle\lambda i.?,\ ?\rangle{\downarrow}1)\llbracket i\rrbracket = ?$
7. $(\rho[i/\delta]\downarrow 1)\llbracket j\rrbracket = \mathit{if}\ i{=}j\ \mathit{then}\,\delta\ \mathit{else}\,(\rho{\downarrow}1)\llbracket j\rrbracket$
8. $(\langle\rho{\downarrow}1,\kappa\rangle{\downarrow}1)\llbracket i\rrbracket = (\rho{\downarrow}1)\llbracket i\rrbracket$
9. $\langle\lambda i.?,?\rangle{\downarrow}2 = ?$
10. $(\rho[i/\delta]){\downarrow}2 = \rho{\downarrow}2$
11. $\langle\rho{\downarrow}1,\kappa\rangle{\downarrow}2 = \kappa$
12. $(\lambda\sigma'.\sigma')(\sigma) = \sigma$
13. $?(\sigma) = ?$
14. $\mathfrak{C}\llbracket c\rrbracket\rho\,(\theta) = \mathfrak{C}\llbracket c\rrbracket\rho\theta$
15. $\mathrm{cond}(\theta_1,\theta_2)(\mathrm{true}\ \mathit{in}\,V) = \theta_1$
16. $\mathrm{cond}(\theta_1,\theta_2)(\mathrm{false}\ \mathit{in}\,V) = \theta_2$
17. $\mathrm{cond}(\theta_1,\theta_2)(\phi\ \mathit{in}\,V) = ?$
18. $\mathrm{cond}(\theta_1,\theta_2)? = ?$
19. $\mathrm{call}(\theta)(\tau\ \mathit{in}\,V) = ?$
20. $\mathrm{call}(\theta)(\phi\ \mathrm{in}\ V) = \phi(\theta)$
21. $\mathrm{call}(\theta)? = ?$
22. $\mathrm{update}(\ell\ \mathit{in}\,D,\theta)(\nu)(\sigma) = \theta(\sigma[\ell/\nu])$
23. $\mathrm{update}\ (\kappa\ \mathit{in}\,D,\theta)(\nu) = ?$
24. $\mathrm{update}(?,\theta)(\nu) = ?$
25. $?(\nu) = ?$
26. $(\mathfrak{D}\llbracket \mathbf{var}\ i\rrbracket\rho\sigma){\downarrow}1 = \rho[i/\mathrm{new}\ (\sigma)]$
27. $(\mathfrak{D}\llbracket \mathbf{var}\ i\rrbracket\rho\sigma){\downarrow}2 = \sigma[\mathrm{new}\ (\sigma)/?]$
28. $(\mathfrak{D}\llbracket d_1;d_2\rrbracket\rho\sigma){\downarrow}1 = (\mathfrak{D}\llbracket d_2\rrbracket(\mathfrak{D}\llbracket d_1\rrbracket\rho\sigma){\downarrow}1\ (\mathfrak{D}\llbracket d_1\rrbracket\rho\sigma){\downarrow}2){\downarrow}1$
29. $(\mathfrak{D}\llbracket d_1;d_2\rrbracket\rho\sigma){\downarrow}2 = (\mathfrak{D}\llbracket d_2\rrbracket(\mathfrak{D}\llbracket d_1\rrbracket\rho\sigma){\downarrow}1\ (\mathfrak{D}\llbracket d_1\rrbracket\rho\sigma){\downarrow}2){\downarrow}2$
30. $\mathfrak{C}\llbracket \mathbf{dummy}\rrbracket\rho\theta = \theta$
31. $\mathfrak{C}\llbracket c_1;c_2\rrbracket\rho\theta = \mathfrak{C}\llbracket c_1\rrbracket\rho\,(\mathfrak{C}\llbracket c_2\rrbracket\rho\theta)$
32. $\mathfrak{C}\llbracket i{:=}e\rrbracket\rho\theta = \mathfrak{E}\llbracket e\rrbracket\rho\,(\mathrm{update}(\rho\llbracket i\rrbracket)\theta)$

33. $\mathcal{C}[\![\textbf{call } e]\!]\rho\theta = \mathcal{E}[\![e]\!]\rho(\text{call}\,\theta)$
34. $\mathcal{C}[\![\textbf{resultis } e]\!]\rho\theta = \mathcal{E}[\![e]\!]\rho(\rho\downarrow 2)$
35. $\mathcal{C}[\![\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2]\!]\rho\theta = \mathcal{E}[\![e]\!]\rho(\text{cond}(\mathcal{C}[\![c_1]\!]\rho\theta, \mathcal{C}[\![c_2]\!]\rho\theta))$
36. $\mathcal{C}[\![\textbf{while } e \textbf{ do } c]\!]\rho\theta = \mathcal{E}[\![e]\!]\rho(\text{cond}(\mathcal{C}[\![c]\!]\rho(\mathcal{C}[\![\textbf{while } e \textbf{ do } c]\!]\rho\theta),\theta)$
37. $\mathcal{C}[\![\textbf{begin } d; c \textbf{ end}]\!]\rho\theta\sigma = \mathcal{C}[\![c]\!]\rho(\mathcal{D}[\![d]\!]\rho\sigma)\downarrow 1\,\theta(\mathcal{D}[\![d]\!]\rho\sigma)\downarrow 2$
38. $\mathcal{E}[\![i]\!]\rho\kappa\sigma = \kappa(\sigma(\rho[\![i]\!]|L))(\sigma)$
39. $\mathcal{E}[\![\textbf{true}]\!]\rho\kappa = \kappa(\text{true})$
40. $\mathcal{E}[\![\textbf{false}]\!]\rho\kappa = \kappa(\text{false})$
41. $\mathcal{E}[\![\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2]\!]\rho\kappa = \mathcal{E}[\![e_0]\!]\rho(\text{cond}(\mathcal{E}[\![e_1]\!]\rho\kappa, \mathcal{E}[\![e_2]\!]\rho\kappa))$
42. $\mathcal{E}[\![\textbf{valof } c]\!]\rho\kappa = \mathcal{C}[\![c]\!]\langle\rho\downarrow 1,\kappa\rangle(?)$
43. $\mathcal{E}[\![\textbf{proc } c]\!]\rho\kappa = \kappa(\mathcal{C}[\![c]\!]\rho \textit{ in } V)$

Having shown Comp to be a model of Th_{DA}, to establish its initiality it is sufficient to show that the unique homomorphism h: $T_{DA} \rightarrow$ Comp is bijective. Had we not used the initial algebra fixed point completion for our definition of Comp this would not have been true and we would need to apply the technique of §4.2.3 to proceed. However as things stand, statements such as "**while true do dummy**" are given the same meaning in T_{DA} as Comp by virtue of our more appropriate fixed point construction.

As a consequence, when showing h to be bijective we need only consider the "sensible" elements of the carriers of T_{DA}. Such elements are *exactly* the ones generated by the operator symbols and constants of the semantic sorts of DA and not by the semantic operators *D1, D2, C* and *E*. So we need to generate those parts of the carriers of T_{DA} and show them to be isomorphic to the *uncompleted* semantic domains of Comp since the elements added for the completion are the same in both algebras. It is quite simple to generate such parts of the carriers of T_{DA} by identifying those operator symbols and constants that are constructors, as outlined by Guttag & Horning (1978).

$T_{DA,Bool} = \{tt, ff\} \cong T = \{true, false\}$
$T_{DA,Loc} = \{new(\sigma) \mid \sigma \in T_{DA,Store}\}$
L is unspecified in Comp save that new must return a location unused in the current store. The above set of locations is certainly sufficient to ensure this.
$T_{DA,Val} = \{injB(\tau) \mid \tau \in T_{DA,Bool}\} \cup \{injA(\phi) \mid \phi \in T_{DA,Abstr}\} \cup \{errV\}$
$\cong V = T + F + \{?\}$

$T_{DA,Den} = \{injL(\ell) \mid \ell \in T_{DA,Loc}\} \cup \{injK(\kappa) \mid \kappa \in T_{DA,Kont}\} \cup \{errD\}$
$\quad \cong D = L + K + \{?\}$
$T_{DA,Store} = (\{empty\} \cup \{set(\sigma,\ell,\nu) \mid \sigma \in T_{DA,Store}, \ell \in T_{DA,Loc}, \nu \in T_{DA,Val}\} \cup \{errS\})^+$
$\quad \cong S = ([L \times V]^* + \{?\})^+$
$T_{DA,Env} = \{arid\} \cup \{bind(\rho,i,\delta) \mid \rho \in T_{DA,Env}, i \in Ide, \delta \in T_{DA,Den}\} \cup \{bindres(\rho,\kappa) \mid \rho \in T_{DA,Env}\}$
$\quad \cong U = [Ide \times D]^* \times K$
$T_{DA,Modif} = \{contin(\phi,\theta) \mid \phi \in T_{DA,Abstr}, \theta \in T_{DA,Modif}\} \cup \{klo(e,\rho,\kappa) \mid e \in Exp, \rho \in T_{DA,Env}, \kappa \in T_{DA,Kont}\} \cup \{kontin(\kappa,\nu) \mid \kappa \in T_{DA,Kont}, \nu \in T_{DA,Val}\} \cup \{null\} \cup \{fail\}$
$\quad \cong C = [F \times C] + [Exp \times U \times K] + [E \times K] + \{final\} + \{fail\}$
$T_{DA,Kont} = \{update(\delta,\theta) \mid \delta \in T_{DA,Den}, \theta \in T_{DA,Modif}\} \cup \{call(\theta) \mid \theta \in T_{DA,Modif}\} \cup \{cond(\theta_1,\theta_2) \mid \theta_1,\theta_2 \in T_{DA,Modif}\} \cup \{errK\}$
$\quad \cong K = [\{update\} \times D \times C] + [\{call\} \times C] + [\{cond\} \times C \times C] + \{?\}$
$T_{DA,Abstr} = \{clo(c,\rho) \mid c \in Com, \rho \in T_{DA,Env}\}$
$\quad \cong F = [Com \times U]$

4.5.3 The Theory Morphism

To finish our proof of the congruence of the completion semantics and continuation semantics of DEVIL we need only show that there is a theory morphism δ: $Th_{DA} \to Th_{DB}$, thus ensuring that there is a derived model $U_\delta(Cont)$ in Alg_{DA} and there is a homomorphism Comp $\to U_\delta(Cont)$ due to the initiality of Comp established above. Note that we do not need to write down $U_\delta(Cont)$ at any stage since it is completely determined by the definition of δ and the process of proving δ to be a theory morphism is exactly parallel to showing the derived algebra to be a model of DA.

As may be expected due to the close similarity between DA and DB, the theory morphism δ: $Th_{DA} \to Th_{DB}$ is easily defined and verified. In fact, δ takes the sorts of DA to the sorts of DB with the same names and similarly matches identical operator symbols from DA and DB with the following exceptions:

$\delta(klo)(e,\rho,\kappa) = \mathcal{E}_B(e, \rho,\kappa)$
$\delta(loop)(e,c,\rho,\theta) = \mathcal{E}_B(e, \rho, cond(\mathcal{C}_B(c, \rho, \mathcal{C}_B(\textbf{while } e \textbf{ do } c, \rho,\theta)),\theta))$
$\delta(\mathcal{C}_A)(c, \rho,\theta,\sigma) = apply_B(\mathcal{C}_B(c, \rho,\theta),\sigma)$
$\delta(\mathcal{E}_A)(e, \rho,\kappa,\sigma) = apply_B(\mathcal{E}_B(e, \rho,\kappa),\sigma)$

Due to the apparent similarites of DA and DB, showing that δ is a theory morphism (by showing that the δ-translation of the equations of DA hold in DB; presentation lemma §2.3) is quite straightforward. All of the equations of DA either map exactly on to equations of DB, or are trivially true, or require a maximum of two steps to establish them as a consequence of the equations of DB. For this reason we dispense with the tedium of a detailed proof.

As mentioned above (§4.5.1), there is an alternative for the **while** -statement clause in the completion semantics that does not require the generation of an infinite data structure for its command completion:
$\mathfrak{C}_0[\![$**while** e **do** c$]\!]\rho\theta\sigma = \mathcal{E}_0[\![e]\!]\rho\,(\mathrm{cond},\langle\langle c,\rho\rangle,\langle\langle$**while** e **do** c, $\rho\rangle,\theta\rangle\rangle,\theta)\sigma$
Henson & Turner's reasons for not using this interpreter-oriented version are not entirely clear, but the most likely explanation is that they were aiming to make completion semantics as abstract, or as much like the denotational semantics as possible. The infinite version also helps to simplify their proof of congruence by bringing the two semantics closer together. In constrast, our style of proof is marginally easier for the finite version. The presentation DA requires the following equation to replace that for $\mathcal{C}$(**while** e **do** c, ρ,θ,σ) :

38'. $\mathcal{C}$(**while** e **do** c, ρ,θ,σ) =

$\mathcal{E}$(e,ρ,cond(contin(clo(c,ρ),contin(clo(**while** e **do** c, ρ),θ)),θ),σ)

and the operator loop: Exp × Com × Env × Modif → Modif can be eliminated entirely, since its sole purpose was to construct an infinite term of sort Modif in the initial model of DA. The elimination of the loop operator also further simplifies the theory morphism, δ.

4.6 Stoy's PL Example

Our final detailed example is an algebraic version of the congruence proof by Stoy (1981). The two definitions are very different both in notation and their underlying concepts, one being an interpreter based on continuations and operating on strings of text, the other a standard direct denotational semantics. As such, for our algebraic proof more emphasis will need to be placed on finding appropriate theories and showing the two semantics to be models than has been the case in earlier proofs.

We make only one minor alteration to Stoy's definitions. Though he gives the direct (and continuation) semantics for the full language PL, the interpreter (and hence the congruence) is only defined for a *kernel* of the language.

e:: = i | e(e) | **proc**(i): e | **rec** i(i): e | c **res** e | b |
  0e | eΩe | **if** e **then** e **else** e | **let** i=e **in** e |
  iterate i **to** e **from** e **while** e
c:: = i:=e | **while** e **do** c | c; c | **if** e **then** c **else** c | ()

full syntax of PL

e:: = i | i(i) | **proc** (i): e | **rec** i(i): e | c **res** e | b | 0i | iΩi
c:: = i:=e | **while** i **do** c | c; c | **if** i **then** c **else** c | ()

syntax of kernel PL

Stoy cites Dennis (1974) as showing that any PL program can be transformed into an equivalent one in the kernel language, thus justifying an immediate simplification of the problem at least in terms of the length of proofs. We take the view that since the congruence involves only the kernel language, any reference to the full language is peripheral and possibly distracting. Therefore our denotational definition is of the kernel language.

It should be noted that since the semantics are in terms of an environment only rather than the more usual environment plus store, a number of the syntactic constructs in the language do not behave as their appearance may suggest. Thus for example, assignments are more like local declarations so the result of the following PL program is 3 rather than 4.

```
( i:=3;
  p:=proc (j): (i:=4 res j);
  k:=p(i)  ) res i
```

For the purpose of comparison we shall briefly outline the main steps in the congruence proof as Stoy approaches it. First define a continuation denotational semantics to act as a bridge between the interpreter and the direct denotational semantics. The two denotational definitions are shown to be congruent by defining appropriate predicates and showing by structural induction over PL that they are satisified. The proof that these predicates exist becomes that most difficult part since they are not monotonic and thus the fixed-point result of Tarski (1955) is of no help and an induction over the complexity of approximations to reflexive domains is required. The relationship between the interpeter and the continuation semantics is then considered. Here, predicates are defined whereby the interpeter can be shown (by fixed-point induction) to be weaker than the continuation semantics and the continuation semantics can be shown (by structural induction) to be weaker than the interpreter. Again, the existence of the predicates must be established. Finally, since the sense of "weaker" is different in each case, further work is required to combine the two results concerning the continuation semantics and the interpreter into a final relation.

In constrast, our proof requires no intermediate semantic definition and consists simply of showing the interpreter to be an initial model of a particular theory Th_C and showing that there is a derived model of the direct denotational semantics that is also a model of Th_C, exactly as we have done in earlier examples.

4.6.1 The Interpreter

We begin by reconstructing Stoy's (1981) description of the PL interpreter, noting that simpler examples using the same notation and concepts can be found in Stoy (1977). Unfortunately, some of the details of the operation of the PL interpreter are either loosely explained or referred to Dennis (1976) which we are unable to obtain. When showing the interpreter to be a model of Th_C (presented below), we will therefore need to use our intuitive understanding to make such operations explicit.

The interpreter operates on *states* which are textual representations of the current stage of program execution. Thus, *Interpret* is a function from states to states defined as follows

Interpret = *fix*($\lambda\phi\lambda\sigma$. *Terminal*(σ) $\rightarrow$ σ, ϕ(*Stop* (σ))) where a *Terminal* state occurs when an error arises or when the execution is successfully completed. In our usual fashion we differ from Stoy's intention and construe the above equation to be in terms of the algebraic fixed-point. The syntax of states is as follows:

σ:: = **eval** e **in** ρ; κ | **perform** c **in** ρ; θ |
  assign ν **to** i **in** ρ; θ | **done** ν | **error**

Here κ and θ denote the "continuations" of expression and command evaluation, and their syntax reflects the notion of "incomplete states".

κ:: = **done** <> | **assign** <> **to** i **in** ρ; θ
θ:: = **perform** c **in** <>; θ | **eval** e **in** <>; κ

The symbol <> represents missing components (values or environments), so that
Append '*val*': ν *to* κ
and
Append '*env*': ρ *to* θ
both produce complete states. (This is an example of the operations whose definitions must be formalised before the proof is possible). Environments , ρ, are data structures with the following auxiliary operations defined on them:
Has (ρ,i) tests whether ρ has a component with selector i.
Select (ρ,i) gives the component of ρ with selector i.
Append i:ν *to* ρ gives a new structure containing ν with selector i, replacing any component with selector i in ρ and leaving all other components as in ρ. In the following tables describing *Terminal* and *Step* the notation $\sigma \equiv$ ⌜**eval** e **in** ρ; κ⌝ tests whether σ is of the specified syntactic form and also introduces names for the various components, which may be used in an arm of a conditional invoked by satisfying the test. The symbol $\equiv$ bears little relation to the use of the same symbol in the denotational definition of §4.6.2.

```
*Terminal* (σ) ≡ (σ ≡ ⌜**done** *v*⌝) ∨ (σ ≡ ⌜**error**⌝)
*Step*(σ) ≡
σ ≡ ⌜**eval** e **in** ρ; κ⌝ →
      e ≡ ⌜i⌝ →
            (*Has*(ρ,i) → *Append 'val'*: *Select*(ρ,i) *to* κ,
            ⌜**error**⌝),
      e ≡ ⌜$i_0$ ($i_1$)⌝ →
            *Has*(ρ,$i_0$) → *Has*(ρ, $i_1$) →
                  *Select*(ρ,$i_0$) ≡ ⌜**function** ($i_2$): $e_2$ **in** $ρ_2$⌝ →
                     ⌜**eval** $e_2$ **in** (*Append* $i_2$: *Select*(ρ,$i_1$) *to* $ρ_2$); κ⌝,
                  *Select*(ρ,$i_0$) ≡ ⌜**recfun** $i_2$ ($i_3$) : $e_2$ **in** $ρ_2$⌝ →
                     ⌜**eval** $e_2$ **in** (*Append* $i_3$: *Select*(ρ,$i_1$) *to*
                                    (*Append* $i_2$: *Select*(ρ,$i_0$) *to* $ρ_2$)); κ⌝,
                  ⌜**error**⌝,
            ⌜**error**⌝, ⌜**error**⌝,
      e ≡ ⌜**proc** (i): $e_0$⌝ →
            *Append 'val'*: ⌜**function** (i): $e_0$ **in** ρ⌝ *to* κ,
      e ≡ ⌜**rec** $i_0$ ($i_1$) : $e_0$⌝ →
            *Append 'val'* : ⌜**recfun** $i_0$ ($i_1$) : $e_0$ **in** ρ⌝ *to* κ,
      e ≡ ⌜c **res** $e_0$⌝ →
            ⌜**perform** c **in** ρ; ⌜**eval** $e_0$ **in** ⟨⟩; κ⌝⌝,
      e ≡ ⌜B⌝ →
            *Append 'val'*: *Rep*(B) *to* κ
      e ≡ ⌜0i⌝ → ...
      e ≡ ⌜$i_1$Ω$i_2$⌝ → ...
      ⌜**error**⌝,
σ ≡ ⌜**perform** c **in** ρ; θ⌝ →
      c ≡ ⌜i: = e⌝ →
            ⌜**eval** e **in** ρ; ⌜**assign** ⟨⟩ **to** i **in** ρ; θ⌝⌝,
      c ≡ ⌜**while** i **do** $c_0$ ⌝ →
            *Has*(ρ, i) → *Select*(ρ, i) →
                  ⌜**perform** $c_0$ **in** ρ; ⌜**perform** ⌜**while** i **do** $c_0$⌝ **in** ⟨⟩; θ⌝⌝,
                  *Append 'env'*:ρ *to* θ,
            ⌜**error**⌝,
      c ≡ ⌜$c_0$; $c_1$⌝ →
            ⌜**perform** $c_0$ **in** ρ; ⌜**perform** $c_1$ **in** ⟨⟩; θ⌝⌝,
      c ≡ ⌜**if** i **then** $c_0$ **else** $c_1$⌝ →
            *Has*(ρ,i) → *Select*(ρ, i) → ⌜**perform** $c_0$ **in** ρ; θ⌝,
                ⌜perform c$_1$ in ρ; θ⌝,
            ⌜error⌝,
        c ≡ ⌜()⌝ →
            Append 'env': ρ to θ,

        ⌜error⌝,
    σ ≡ ⌜assign ν to i in ρ; θ⌝ →
        Append 'env': (Append i:ν to ρ) to θ,

    ⌜error⌝
```

Int - the PL interpreter

The careful reader may note that by a simple extension of the _ → _, _ operator, certain type checks have been factored out from the clauses for **while** and **if** statements in Stoy's original definition.

The interpreter is the first semantic definition we have dealt with that does not seem to fit the initial algebra semantics template of a homomorphism from a free syntactic algebra to a semantic algebra of the same signature, as discussed in §3.2. However, by inspecting the syntax of states and continuations we can make various distinctions and observations. For example we may readily view the command continuation ⌜**eval** e **in** ‹›; κ⌝ as a function from expressions and expression continuations to command continuations. The full range of such observations will be detailed when showing the interpreter (the algebra Int) to be a model of Th_C where C is the continuation semantics scheme presented below.

<u>Signature</u>

Sort B

... unspecified but including tt and ff

Sort E

injB: B → E

fun: Ide × Exp × Env → E

rec: Ide × Ide × Exp × Env → E

errE: → E

ok: E → B

sort Env

arid: $\rightarrow$ Env

bind: Env $\times$ Ide $\times$ E $\rightarrow$ Env

find: Env $\times$ Ide $\rightarrow$ E

has: Env $\times$ Ide $\rightarrow$ E

sort C

eval: Exp $\times$ K $\rightarrow$ C

perf: Com $\times$ C $\rightarrow$ C

run: C $\times$ Env $\rightarrow$ E

sort K

assign: Ide $\times$ Env $\times$ C $\rightarrow$ K

done: $\rightarrow$ K

send: K $\times$ E $\rightarrow$ E

call: E $\times$ E $\times$ K $\rightarrow$ E

$\mathcal{P}$: Exp $\rightarrow$ E

$\mathcal{E}$: Exp $\times$ Env $\times$ K $\rightarrow$ E

$\mathcal{C}$: Com $\times$ Env $\times$ C $\rightarrow$ E

<u>Equations</u>

1. find(arid,i) = errE
2. find(bind(ρ,i,ν),j) = *if* i=j *then* ν *else* find(ρ,j)
3. ok(injB(b)) = tt
4. ok(fun(i,e,ρ)) = tt
5. ok(rec(i,j,e,ρ)) = tt
6. ok(errE) = ff
7. has(ρ,i) = ok(find(ρ,i))
8. run(eval(e,κ),ρ) = $\mathcal{E}$(e, ρ,κ)
9. run(perf(c,θ),ρ) = $\mathcal{C}$(c, ρ,θ)
10. send(assign(i,ρ,θ),ν) = *if* ok(ν) *then* run(θ,bind(ρ,i,ν)) *else* errE
11. send(done,ν) = ν
12. send(κ,errE) = errE
13. call(fun(i,e,ρ),ν,κ) = $\mathcal{E}$(e, bind(ρ,i,ν),κ)
14. call(rec(i,j,e,ρ),ν,κ) = $\mathcal{E}$(e, bind(bind(ρ,i,rec(i,j,e,ρ)),j,ν),κ)
15. call (injB(b),ν,κ) = errE
16. call(errE,ν,κ) = errE
17. $\mathcal{P}$(e) = $\mathcal{E}$(e, arid,done)
18. $\mathcal{E}$(i, ρ,κ) = *if* has(ρ,i) *then* send(κ,find(ρ,i)) *else* errE
19. $\mathcal{E}$(i_1(i_2), ρ,κ) = *if* has(ρ,i_1) *then* *if* has(ρ,i_2) *then*
    call(find(ρ,i_1),find(ρ,i_2)) *else* errE *else* errE

20. $\mathcal{E}$(**proc** (i):e, ρ,κ) = send(κ,fun(i,e,ρ))
21. $\mathcal{E}$(**rec** i(j): e, ρ,κ) = send(κ,rec(i,j,e,ρ))
22. $\mathcal{E}$(c **res** e, ρ,κ) = $\mathcal{C}$(c, ρ,eval(e,κ))
23. $\mathcal{E}$(b, ρ,κ) = send(κ,injB(b))
24. $\mathcal{E}$(Oi, ρ,κ) = ...
25. $\mathcal{E}$($i_1 \Omega i_2$, ρ,κ) = ...
26. $\mathcal{C}$(i:=e, ρ,θ) = $\mathcal{E}$(e, ρ,assign(i,ρ,θ))
27. $\mathcal{C}$(**while** i **do**c, ρ,θ) = *if* has(ρ,i) *then if* find(ρ,i) *then*
$\mathcal{C}$(c, ρ,perf(**while** i **do** c,θ))
else run(θ,ρ) *else* errE
28. $\mathcal{C}$(c_1;c_2, ρ,θ) = $\mathcal{C}$(c_1, ρ,perf(c_2,θ))
29. $\mathcal{C}$(**if** i **then** c_1 **else** c_2, ρ,θ) = *if* has(ρ,i) *then if* find(ρ,i) *then*
$\mathcal{C}$(c_1, ρ,θ) *else* $\mathcal{C}$(c_2, ρ,θ)
else errE
30. $\mathcal{C}$((), ρ,θ) = run(θ,ρ)

C - continuation semantics presentation

To show that the interpreter is a model of Th_C we need to specify first the carriers for the sorts of C and then the functions corresponding to the operator symbols of C. Since the interpreter operates on states (and continuations) that are syntactic objects, it is appropriate that we choose certain sets of strings as the carriers of the sorts.

Int_B = B
Int_E = *Rep*(b) | ⌜**function** (i): e **in** ρ⌝ |
⌜**recfun** i (j): e **in** ρ⌝ | ⌜**error**⌝
Int_{Env} = $[Ide \times Int_E]^*$
Int_C = ⌜**perform** c **in** <>; θ⌝ | ⌜**eval** e **in** <>; κ⌝
Int_K = ⌜**done** <>⌝ | ⌜**assign** <> **to** i **in** ρ; θ⌝

At this stage, having fixed upon the domains involved in the interpreter's operation we are able to give appropriate rigorous definitions of the auxiliary functions left loosely described in the original paper.

Append 'val': $Int_E \times Int_K \rightarrow$ state
Append 'env': $Int_{Env} \times Int_C \rightarrow$ state
Append: $Ide \times Int_E \times Int_{Env} \rightarrow Int_{Env}$
Select: $Int_{Env} \times Ide \rightarrow Int_E$

$\mathit{Has}: \mathrm{Int}_{\mathrm{Env}} \times \mathrm{Ide} \to \mathrm{B}$

$$\mathit{Append}\ \text{'}\mathit{val}\text{'}: \nu\ \mathit{to}\ \ulcorner\mathbf{done}\langle\rangle\urcorner = \begin{cases} \ulcorner\mathbf{error}\urcorner & \text{if } \nu = \ulcorner\mathbf{error}\urcorner \\ \ulcorner\mathbf{done}\ \nu\urcorner & \text{otherwise} \end{cases}$$

$$\mathit{Append}\ \text{'}\mathit{val}\text{'}: \nu\ \mathit{to}\ \ulcorner\mathbf{assign}\ \langle\rangle\ \mathbf{to}\ \mathrm{i}\ \mathbf{in}\ \rho;\theta\urcorner = \begin{cases} \ulcorner\mathbf{error}\urcorner & \text{if } \nu = \ulcorner\mathbf{error}\urcorner \\ \ulcorner\mathbf{assign}\ \nu\ \mathbf{to}\ \mathrm{i}\ \mathbf{in}\ \rho;\theta\urcorner & \text{otherwise} \end{cases}$$

$\mathit{Append}\ \text{'}\mathit{env}\text{'}: \rho\ \mathit{to}\ \ulcorner\mathbf{perform}\ \mathrm{c}\ \mathbf{in}\ \langle\rangle;\ \theta\urcorner = \ulcorner\mathbf{perform}\ \mathrm{c}\ \mathbf{in}\ \rho;\theta\urcorner$
$\mathit{Append}\ \text{'}\mathit{env}\text{'}: \rho\ \mathit{to}\ \ulcorner\mathbf{eval}\ \mathrm{e}\ \mathbf{in}\ \langle\rangle;\kappa\urcorner = \ulcorner\mathbf{eval}\ \mathrm{e}\ \mathbf{in}\ \rho;\kappa\urcorner$
$\mathit{Append}\ \mathrm{i}{:}\nu\ \mathit{to}\ \rho = \langle \mathrm{i}, \nu\rangle\ \mathit{cat}\ \rho$

$$\mathit{Select}(\rho,\mathrm{i}) = \begin{cases} 2\mathrm{nd}(\rho\downarrow\mathrm{n}) & \text{if n exists s.t. } 1\mathrm{st}(\rho\downarrow\mathrm{n}) = \mathrm{i} \text{ and } \forall\, \mathrm{m}<\mathrm{n},\ 1\mathrm{st}(\rho\downarrow\mathrm{m}) \neq \mathrm{i} \\ \ulcorner\mathbf{error}\urcorner & \text{otherwise} \end{cases}$$

$\mathit{Has}(\rho,\mathrm{i}) = \mathit{Select}(\rho,\mathrm{i}) \neq \ulcorner\mathbf{error}\urcorner$

This definition of *Has* has been used rather than the more natural version

$$\mathit{Has}(\rho,\mathrm{i}) = \begin{cases} \text{true} & \text{if n exists s.t. } 1\mathrm{st}(\rho\downarrow\mathrm{n}) = \mathrm{i} \\ \text{false} & \text{otherwise} \end{cases}$$

The two definitions are clearly not equivalent by virtue of the fact that $\mathrm{Int}_{\mathrm{E}}$ includes the error element, ⌜**error**⌝. However, it can be shown by structural induction over PL that an environment with the error value bound to an identifier never arises during the interpretation of any program, thus ensuring that the two definitions of *Has* are effectively equivalent in the current application. Further, albeit informal justification for the acceptibility of the first definition of *Has* exists in the observation that the interpreter's sole purpose for *Has* is to "protect" the function *Select* from failure (i.e. returning an error).

The style of many of the "functions" derived from the interpreter is reflected by our earlier observation that the string ⌜**eval** e **in** $\langle\rangle;\kappa$⌝ can be considered as the result of applying a function to an expression and an expression continuation. This idea has much in common with the pioneering work of Goguen et al (1977). Thus the operator correspondence is as follows:

C	Int
injB: B → E	*Rep*
fun: Ide × Exp × Env → E	λieρ. ⌜**function** (i): e **in** ρ⌝
rec: Ide × Ide × Exp × Env → E	λijeρ. ⌜**recfun** i(j): e **in** ρ⌝
errE: → E	⌜**error**⌝
ok: E → B	λ*v*. ¬(*v* ≡ ⌜**error**⌝)
arid: → Env	⟨⟩
bind: Env × Ide × E → Env	λρi*v*. *Append* i:*v* *to* ρ
find: Env × Ide → E	*Select*
has: Env × Ide → B	*Has*
eval: Exp × K → C	λeκ. ⌜**eval** e**in** ⟨⟩;κ⌝
perf: Com × C → C	λcθ. ⌜**perform** c **in** ⟨⟩;θ⌝
run: C × Env → E	λθρ. *Interpret* (*Append* '*env*': ρ *to* θ)
assign: Ide × Env × C → K	λiρθ. ⌜**assign** ⟨⟩ **to** i **in** ρ;θ⌝
done: → K	⌜**done** ⟨⟩⌝
send: K × E → E	λκ*v*. *Interpret* (*Append* '*val*': *v* *to* κ)
call: E × E × K → E	λ$v_1 v_2$κ. *Interpret* (v_1 ≡ ⌜**function** (i): e **in** ρ⌝ → ⌜**eval** e **in** (*Append* i:v_2 *to* ρ);κ⌝, v_1 ≡ ⌜**recfun** i(j): e **in** ρ⌝ → ⌜**eval** e **in** (*Append* j:v_2 *to* (*Append* i: ⌜**recfun** i(j):e **in** ρ⌝ *to* ρ));κ⌝, ⌜**error**⌝)
P: Exp → E	λe. *Interpret* (⌜**eval** e **in** arid; ⌜**done** ⟨⟩⌝⌝)
E: Exp × Env × K → E	λeρκ. *Interpret* (⌜**eval** e **in** ρ;κ⌝)
C: Com × Env × C → E	λcρθ. *Interpret* (⌜**perform** c **in** ρ;θ⌝)

The rather messy version of call above is due to the informal notion of the binary operator ≡ having the *side-effect* of pattern-matching and binding appropriate syntactic forms to variables. A more rigorous (though less expedient) alternative notation would be the *analytic* syntax of McCarthy (1962) which consists in part of selector functions that return substrings of syntactic forms.

The following table of equations is the translation of the equations of

C according to the signature correspondence above. It can be easily established that the interpreter satisfies all of the equations, thus showing Int to be a model of Th_C.

1. $\mathit{Select}(\langle\rangle, i) = \ulcorner\mathbf{error}\urcorner$
2. $\mathit{Select}(\mathit{Append}\ i{:}\,v\ \mathit{to}\ \rho, j) = i{=}j \rightarrow v,\ \mathit{Select}(\rho, j)$
3. $\neg(\mathit{Rep}(b) = \ulcorner\mathbf{error}\urcorner) = \mathrm{true}$
4. $\neg(\ulcorner\mathbf{function}\ (i){:}\ e\ \mathbf{in}\ \rho\urcorner \equiv \ulcorner\mathbf{error}\urcorner) = \mathrm{true}$
5. $\neg(\ulcorner\mathbf{recfun}\ i(j){:}\ e\ \mathbf{in}\ \rho\urcorner \equiv \ulcorner\mathbf{error}\urcorner) = \mathrm{true}$
6. $\neg(\ulcorner\mathbf{error}\urcorner \equiv \ulcorner\mathbf{error}\urcorner) = \mathrm{false}$
7. $\mathit{Has}(\rho, i) = \neg(\mathit{Select}(\rho, i) = \ulcorner\mathbf{error}\urcorner)$
8. $\mathit{Interpret}(\mathit{Append}\ \text{'}\mathit{env}\text{'}{:}\ \rho\ \mathit{to}\ \ulcorner\mathbf{eval}\ e\ \ \mathbf{in}\ \langle\rangle;\kappa\urcorner) =$
   $\quad\mathit{Interpret}(\ulcorner\mathbf{eval}\ e\ \mathbf{in}\ \rho;\ \kappa\urcorner)$
9. $\mathit{Interpret}(\mathit{Append}\ \text{'}\mathit{env}\text{'}{:}\ \rho\ \mathit{to}\ \ulcorner\mathbf{perform}\ c\ \mathbf{in}\langle\rangle;\ \theta\urcorner) =$
   $\quad\mathit{Interpret}(\ulcorner\mathbf{perform}\ c\ \mathbf{in}\ \ \rho;\ \theta\urcorner)$
10. $\mathit{Interpret}(\mathit{Append}\ \text{'}\mathit{val}\text{'}{:}\,v\ \ \mathit{to}\ \ulcorner\mathbf{assign}\ \langle\rangle\ \mathbf{to}\ i\ \mathbf{in}\ \rho;\ \theta\urcorner) =$
    $\quad\neg(v \equiv \ulcorner\mathbf{error}\urcorner) \rightarrow$
    $\qquad\mathit{Interpret}(\mathit{Append}\ \text{'}\mathit{env}\text{'}{:}\ (\mathit{Append}\,i{:}\,v\ \ \mathit{to}\,\rho)\ \mathit{to}\,\theta),$
    $\qquad\ulcorner\mathbf{error}\urcorner$
11. $\mathit{Interpret}(\mathit{Append}\ \text{'}\mathit{val}\text{'}{:}\,v\ \ \mathit{to}\ \ulcorner\mathbf{done}\ \langle\rangle\urcorner) = \ulcorner\mathbf{done}\ v\urcorner$
12. $\mathit{Interpret}(\mathit{Append}\ \text{'}\mathit{val}\text{'}{:}\ \ulcorner\mathbf{error}\urcorner\ \ \mathit{to}\,\kappa) = \ulcorner\mathbf{error}\urcorner$
13. $\mathit{Interpret}(\ulcorner\mathbf{function}\ (i){:}\ e\ \mathbf{in}\ \rho\urcorner \equiv \ulcorner\mathbf{function}\ (i'){:}\ e'\ \mathbf{in}\ \rho'\urcorner \rightarrow$
    $\qquad\qquad\ulcorner\mathbf{eval}\ e'\ \mathbf{in}\ (\mathit{Append}\ \ i'{:}\,v\ \ \mathit{to}\ \ \rho');\kappa\urcorner,$
    $\qquad\ulcorner\mathbf{function}\ (i){:}\ e\ \mathbf{in}\ \rho\urcorner \equiv \ulcorner\mathbf{recfun}\ i'(j'){:}\ e'\ \mathbf{in}\ \rho'\urcorner \rightarrow$
    $\qquad\qquad\ulcorner\mathbf{eval}\ e'\ \mathbf{in}\ (\mathit{Append}\ \ j'{:}\,v\ \ \mathit{to}$
    $\qquad\qquad\ (\mathit{Append}\ \ i'{:}\ \ulcorner\mathbf{recfun}\ i'(j'){:}\ e'\ \mathbf{in}\ \rho'\urcorner\ \ \mathit{to}\,\rho'));\kappa\urcorner,$
    $\qquad\ulcorner\mathbf{error}\urcorner) =$
    $\quad\mathit{Interpret}(\ulcorner\mathbf{eval}\ e\ \mathbf{in}\ (\mathit{Append}\ \ i{:}\,v\ \ \mathit{to}\ \ \rho);\kappa\urcorner)$
14. $\mathit{Interpret}(\ulcorner\mathbf{recfun}\ i(j){:}\ e\ \mathbf{in}\ \rho\urcorner \equiv \ulcorner\mathbf{function}\ (i'){:}\ e'\ \mathbf{in}\ \rho'\urcorner \rightarrow$
    $\qquad\qquad\ulcorner\mathbf{eval}\ e'\ \mathbf{in}\ (\mathit{Append}\ i'{:}\,v\ \ \mathit{to}\,\rho');\kappa\urcorner,$
    $\qquad\ulcorner\mathbf{recfun}\ i(j){:}\ e\ \mathbf{in}\ \rho\urcorner \equiv \ulcorner\mathbf{recfun}\ i'(j'){:}\ e'\ \mathbf{in}\ \rho'\urcorner$
    $\qquad\qquad\ulcorner\mathbf{eval}\ e'\ \mathbf{in}\ (\mathit{Append}\ j'{:}\,v\ \ \mathit{to}$
    $\qquad\qquad\quad(\mathit{Append}\ i'{:}\ \ulcorner\mathbf{recfun}\ i'(j'){:}\ e'\ \mathbf{in}\ \rho'\urcorner\ \ \mathit{to}\,\rho'));\kappa\urcorner,$
    $\qquad\ulcorner\mathbf{error}\urcorner) =$
    $\quad\mathit{Interpret}(\ulcorner\mathbf{eval}\ e\ \mathbf{in}\ (\mathit{Append}\ j{:}\,v\ \ \mathit{to}$
    $\qquad\qquad(\mathit{Append}\ i{:}\ \ \ulcorner\mathbf{recfun}\ i(j){:}\ e\ \mathbf{in}\ \rho\urcorner\ \mathit{to}\,\rho));\kappa\urcorner)$

```
15.  Interpret(Rep(b) ≡ ⌜function (i'): e' in ρ'⌝ →
                    ⌜eval e' in (Append i': v to ρ');κ⌝,
               Rep(b) = ⌜recfun i'(j'): e' in ρ'⌝ →
                    ⌜eval e' in (Append j':v to
                         (Append i': ⌜recfun i'(j'): e' in ρ'⌝ to ρ')); κ⌝,
               ⌜error⌝) =
          ⌜error⌝
16.  Interpret(⌜error⌝ ≡ ⌜function (i'):e' in ρ'⌝ →
                    ⌜eval e' in (Append i': v to ρ');κ⌝,
               ⌜error⌝ ≡ ⌜recfun i'(j'): e' in ρ'⌝ →
                    ⌜eval e' in (Append j':v to
                         (Append i': ⌜recfun i'(j'): e' in ρ'⌝ to ρ')); κ⌝,
               ⌜error⌝) =
          ⌜error⌝
17.  Interpret (⌜eval e in arid; done ⟨⟩⌝) =
          Interpret (⌜eval e in arid; done ⟨⟩⌝)
18.  Interpret (⌜eval ⌜i⌝ in ρ;κ⌝) =
          Has(ρ,i) → Interpret (Append 'val': Select(ρ,i) to κ),
                    ⌜error⌝
19.  Interpret (⌜eval ⌜$i_1(i_2)$⌝ in ρ;κ⌝) =
          Has(ρ,$i_1$) → Has(ρ,$i_2$) →
          Interpret (Select(ρ,$i_1$) ≡ ⌜function (i'): e' in ρ'⌝ →
               ⌜eval e' in (Append i': Select(ρ,$i_2$) to ρ'); κ⌝,
                    Select(ρ,$i_1$) ≡ ⌜recfun i'(j'): e' in ρ'⌝ →
               ⌜eval e' in (Append j': Select(ρ,$i_2$) to
                         (Append i': ⌜recfun i'(j'): e' in ρ'⌝ to ρ'));κ⌝,
          ⌜error⌝), ⌜error⌝, ⌜error⌝
20.  Interpret(⌜eval ⌜proc (i):e⌝ in ρ;κ⌝) =
          Interpret (Append 'val': ⌜function (i): e in ρ⌝ to κ)
21.  Interpret (⌜eval ⌜rec i(j):e⌝ in ρ;κ⌝) =
          Interpret (Append 'val': ⌜recfun i(j): e in ρ⌝ to κ)
22.  Interpret (⌜eval ⌜c res e⌝ in ρ;κ⌝) =
          Interpret(⌜perform c in ρ; ⌜eval e  in ⟨⟩; κ⌝⌝)
23.  Interpret (⌜eval ⌜b⌝ in ρ;κ⌝) =
          Interpret(Append 'val': Rep(b) to κ)
24.  Interpret (⌜eval ⌜0i⌝ in ρ;κ⌝) = ...
25.  Interpret (⌜eval ⌜$i_1 Ω i_2$⌝ in ρ;κ⌝)  = ...
26.  Interpret (⌜perform ⌜i:=e⌝ in ρ;θ⌝) =
          Interpret (eval e in ρ; ⌜assign ⟨⟩ to i  in ρ;θ⌝⌝)
```

```
27.   Interpret(⌜perform ⌜while i do c⌝ in ρ;θ⌝) =
          Has(ρ,i) → Select(ρ,i) →
              Interpret(⌜perform c  in ρ;
                          ⌜perform ⌜while i do  c⌝ in ⟨⟩; θ⌝⌝),
              Interpret(Append 'env': ρ to θ),
          ⌜error⌝
28.   Interpret(⌜perform ⌜$c_1$;$c_2$⌝ in ρ;θ) =
          Interpret(⌜perform $c_1$ in ρ; ⌜perform $c_2$ in ⟨⟩; θ⌝⌝)
29.   Interpret(⌜perform ⌜if i then $c_1$ else $c_2$⌝ in ρ;θ⌝) =
          Has(ρ,i) → Select(ρ,i) →
              Interpret(⌜perform $c_1$ in ρ;θ⌝),
              Interpret(⌜perform $c_2$ in ρ;θ⌝),
          ⌜error⌝
30.   Interpret(⌜perform ⌜()⌝ in ρ;θ) =
          Interpret(Append 'env': ρ to θ)
```

All that remains is to show Int to be initial in Alg_C. To achieve this we follow the same procedure that was employed in §4.5.2, considering only the sensible parts of the carriers of T_C and generating these by recognising the operator symbols and constants of C that are constructors. Once again, since we are assuming the initial algebra fixed-point completion to have been employed in the interpreter definition we may assume that elements of the carriers of T_C other than those generated by the constructors will also occur in the carriers of Int.

$T_{C,E}$ = {injB(b) | b ∈ B} ∪ {fun (i,e,ρ) | i ∈ Ide, e ∈ Exp, ρ ∈ $T_{C,Env}$} ∪
    {rec(i,j,e,ρ) | i,j ∈ Ide, e ∈ Exp, ρ ∈ $T_{C,Env}$} ∪ {errE}
≅ Int_E = *Rep*(b) | ⌜ **function** (i): e **in** ρ⌝ | ⌜**recfun** i (j): e **in** ρ⌝ | ⌜**error**⌝
    for b ∈ B, i,j ∈ Ide, e ∈ Exp, ρ ∈ Int_{Env}
$T_{C,Env}$ = {arid} ∪ {bind(ρ,i,ν) | ρ ∈ $T_{C,Env}$, i ∈ Ide, ν ∈ $T_{C,E}$}
≅ Int_{Env} = $[\mathrm{Ide} \times \mathrm{E}_{Int}]^*$
$T_{C,C}$ = {eval(e,κ) | e ∈ Exp, κ ∈ $T_{C,K}$} ∪ {perf(c_1,θ) | c ∈ Com, θ ∈ $T_{C,C}$}
≅ Int_C = ⌜**eval** e **in** ⟨⟩; κ⌝ | ⌜**perform** c **in** ⟨⟩; θ⌝
    for e ∈ Exp, c ∈ Com, κ ∈ Int_K, θ ∈ Int_C
$T_{C,K}$ = {assign(i,ρ,θ) | i ∈ Ide, ρ ∈ $T_{C,Env}$, θ ∈ $T_{C,C}$} ∪ {done}
≅ Int_K = ⌜**assign** ⟨⟩ **to** i **in** ρ;θ⌝ | ⌜**done** ⟨⟩⌝
    for i ∈ Ide, ρ ∈ Int_{Env}, θ ∈ Int_C

It is interesting at this point to note that we can immediately construct a completion semantics (Henson & Turner, 1982) similar to that given for DEVIL in §4.5.1 that is also initial in Th_C. The appropriate domain equations are given below; the semantic functions are the obvious ones corresponding to the semantic operators of the presentation C.

$E = B + F + \{?\}$	expressible values
$U = [\mathrm{Ide} \times E]^*$	environments
$C = [\mathrm{Com} \times C] + [\mathrm{Exp} \times K]$	command completions
$K = [\{\mathrm{update}\} \times \mathrm{Ide} \times U \times C] + \{\mathrm{final}\}$	expression completions
$F = [\{\mathrm{function}\} \times \mathrm{Ide} \times \mathrm{Exp} \times U] +$	
$[\{\mathrm{recfun}\} \times \mathrm{Ide} \times \mathrm{Ide} \times \mathrm{Exp} \times U]$	closures

Thus such a completion semantics is *isomorphic* to Stoy's interpreter. On this basis Henson & Turner's call for completion semantics to be considered as the *standard* operational semantics is difficult to sustain since it is only its *notation* that distinguishes it from a host of other "different" operational definitions.

4.6.2 The Direct Denotational Semantics

The denotational definition given below differs from that given in Stoy (1981) in two ways; it gives the semantics of the kernel language rather than PL and corrects several minor (probably typographical) errors in the original definition. The most curious is Stoy's definition of *cond* which reads as follows:

For $x,y \in D$ and b in some domain including T,

$$\begin{array}{lll} \mathit{cond}\,\langle x,y\rangle b \equiv & (b|T) \equiv \mathit{true} \rightarrow x, & \\ & (b|T) \equiv \mathit{false} \rightarrow y, & \\ & ?_D & \\ \text{where } b \rightarrow x,y \equiv & x & \text{if } b \equiv \mathit{true} \\ & y & \text{if } b \equiv \mathit{false} \\ & \bot_D \top_D, ?_D & \text{if } b \equiv \bot_T, \top_T, ?_T. \end{array}$$

So, while $\_\rightarrow\_,\_$ is strict, *cond* is not. Thus, given a clause such as $\mathcal{C}[\![\textbf{if } e \textbf{ then } c_0 \textbf{ else } c_1]\!]\rho = \mathit{cond}\langle \mathcal{C}[\![c_0]\!]\rho, \mathcal{C}[\![c_1]\!]\rho\rangle(\mathcal{E}[\![e]\!]\rho)$, the implication is that in the case where the evaluation of e does not terminate and $\mathcal{E}[\![e]\!]\rho = \bot$ the result of the if statement is an error, ?, rather than $\bot$. Not only is this most unusual, but no interpreter can possibly behave in such a fashion, since it would need to recognise that it had embarked on a non-terminating computation, recover and report an error. As we are proving the congruence of the denotational semantics and an interpreter it is clear that such a definition of *cond* must be unacceptable. In our definition we assume the (completely) strict version of *cond* defined as follows:

$$\mathit{cond}\langle x,y\rangle b \equiv (b|T) \rightarrow x,y$$

<u>Semantic Domains</u>

B	basic values including *true* and *false*
$F = [E \rightarrow E]$	function values
$E = B + F + \{?\}$	expressible values
$U = [Ide \rightarrow E] + \{?\}$	environments

<u>Semantic Functions</u>

$\mathcal{P}: Exp \rightarrow E$

$\mathcal{E}: Exp \rightarrow U \rightarrow E$

$\mathcal{C}: Com \rightarrow U \rightarrow U$

$\mathcal{E}[\![i]\!]\rho \equiv \rho[\![i]\!]$
$\mathcal{E}[\![i_0(i_1)]\!]\rho \equiv \mathit{strict}(\rho[\![i_0]\!] \mid F)(\rho[\![i_1]\!])$
$\mathcal{E}[\![\mathbf{proc}\ (i){:}e]\!] \equiv \mathit{strict}(\lambda\rho\lambda\nu\,\mathcal{E}[\![e]\!](\rho[\nu/i])\ \mathit{in}\ E)$
$\mathcal{E}[\![\mathbf{rec}\ i_0(i_1){:}\ e]\!] \equiv \mathit{strict}(\lambda\rho.\mathit{fix}\,[\lambda\phi\lambda\nu\,\mathcal{E}[\![e]\!](\rho[\phi\ \mathit{in}\ E/i][\nu/i_1])]\ \mathit{in}\,E)$
$\mathcal{E}[\![c\ \mathbf{res}\ e]\!]\rho \equiv \mathcal{E}[\![e]\!](\mathcal{C}[\![c]\!]\rho)$
$\mathcal{E}[\![b]\!] \equiv \mathit{strict}(\lambda\rho.(b\ \mathit{in}\ E))$
$\mathcal{E}[\![0i]\!] = \ldots$
$\mathcal{E}[\![i_0\Omega i_1]\!] = \ldots$
$\mathcal{C}[\![i{:=}e]\!]\rho \equiv \mathit{strict}(\lambda\nu.\rho[\nu/i])(\mathcal{E}[\![e]\!]\rho)$
$\mathcal{C}[\![\mathbf{while}\ i\ \mathbf{do}\ c]\!] \equiv \mathit{fix}(\lambda\theta\lambda\rho.\mathit{cond}\langle\theta(\mathcal{C}[\![c]\!]\rho), \rho\rangle(\rho[\![i]\!]))$
$\mathcal{C}[\![c_0;\, c_1]\!]\rho \equiv \mathcal{C}[\![c_1]\!](\mathcal{C}[\![c_0]\!]\rho)$
$\mathcal{C}[\![\mathbf{if}\ i\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1]\!]\rho \equiv \mathit{cond}\langle\mathcal{C}[\![c_0]\!]\rho, \mathcal{C}[\![c_1]\!]\rho\rangle(\rho[\![i]\!])$
$\mathcal{C}[\![()]\!]\rho \equiv \rho$
$\mathcal{P}[\![e]\!] \equiv \mathcal{E}[\![e]\!](\lambda i.?)$

<u>Auxiliary Definitions</u>
The *separated* sum is implied in the domain equations. For any domain D and a,b ∈ D, a ≡ b is *true* if a is the same element as b and *false* otherwise. The symbol "=" is reserved to denote a continuous (completely strict) equality predicate.
for b ∈ T (truth values) and x,y ∈ D,

$$\begin{array}{ll}
b \to x,y \equiv x & \text{if } b \equiv \mathit{true} \\
\qquad\quad y & \text{if } b \equiv \mathit{false} \\
\qquad\quad \bot,\top,? & \text{if } b \equiv \bot,\top,? \\
\mathit{strict}\ f\ x \equiv \bot,\top,? & \text{if } x \equiv \bot,\top,? \\
\qquad\qquad f(x) & \text{otherwise}
\end{array}$$

ie *strict* f produces a completely strict version of f.

<u>Den - direct denotational semantics of PL</u>

The theory presentation D (of which we intend to show Den to be a model) that follows is influenced largely by the requirement that there must be a theory morphism $Th_C \to Th_D$ and Th_C is constrained quite significantly by the requirement that Int must be its initial model. Further, since our presentation must be first-order in the sense that we have no way of including operator symbols that have a domain consisting of other operator symbols (cf. Parsaye-Ghomi, 1981) we have no way of directly representing functions such as *strict*. Our approach is not to consider ⊥ and ⊤ in presentation D (indeed, this has been our policy for all examples except the first consideration of the lambda calculus) and explicitly test for error values at appropriate points. Hence corresponding to the equation

$$\mathcal{C}[\![i{:=}e]\!]\rho \equiv \mathit{strict}(\lambda\nu.\rho[\nu/i])\mathcal{E}[\![e]\!]\rho$$

in the denotational definition, we use the following equation in D

$$\mathcal{C}(i{:=}e,\rho) = \mathit{if}\ \mathrm{ok}(\mathcal{E}(e,\rho))\ \mathit{then}\ \mathrm{bind}\ (\rho,i,\mathcal{E}(e,\rho))\,\mathit{else}\ \mathrm{errE}$$

where ok(ν) is ff for ν = errE and tt for other sensible values. In the denotational semantics, the function corresponding to ok is a doubly strict test for equality with ?.

One of the important differences between the direct denotational semantics and the continuation-style interpreter is that the direct definition requires the concept of an 'error environment' so that errors can be propagated. In contrast, since the interpreter uses continuations, improper commands immediately give rise to improper results and their effects need not be propagated by means of the environment produced. Our presentations must reflect this situation and since we maintain the need to construct a theory morphism between them, sort Env in D represents the proper environments and sort U represents the sum of Env with an error environment. Further, since $\mathcal{E}_C$ and $\mathcal{C}_C$ in Th_C are only concerned with proper environments it is convenient to give two separate equations for $\mathcal{E}_D$ and $\mathcal{C}_D$ for each phrase of the abstract syntax; one for proper environments, one for the error environment.

<u>Signature</u>.

Sort B

...	unspecified but including tt and ff

Sort E

- injB: B $\rightarrow$ E
- fun: Ide $\times$ Exp $\times$ Env $\rightarrow$ E
- rec: Ide $\times$ Ide $\times$ Exp $\times$ Env $\rightarrow$ E
- errE: $\rightarrow$ E
- apply: E $\times$ E $\rightarrow$ E
- ok: E $\rightarrow$ B
- *if _ then _ else _*: B $\times$ E $\times$ E $\rightarrow$ E

Sort Env

- arid: $\rightarrow$ Env
- bind: Env $\times$ Ide $\times$ E $\rightarrow$ Env
- find: Env $\times$ Ide $\rightarrow$ E

Sort U

$\quad$ injE: Env $\to$ U

$\quad$ errU: $\to$ U

$\quad$ *if* _ *then* _ *else* _: B $\times$ U $\times$ U $\to$ U

$\mathcal{P}$: Exp $\to$ E

$\mathcal{E}$: Exp $\times$ U $\to$ E

$\mathcal{C}$: Com $\times$ U $\to$ U

<u>Equations</u>

D1. ok(injB(b)) = tt

D2. ok(fun(i,e,ρ)) = tt

D3. ok(rec(i,j,e,ρ)) = tt

D4. ok(errE) = ff

D5. apply(fun(i,e,ρ),ν) = $\mathcal{E}$(e, injE(bind(ρ,i,ν)))

D6. apply(rec(i,j,e,ρ),ν) = $\mathcal{E}$(e, injE(bind(bind(ρ,i,rec(i,j,e,ρ)),j,ν)))

D7. apply (injB(b),ν) = errE

D8. apply (errE,ν) = errE

D9. find(arid,i) = errE

D10. find(bind(ρ,i,ν),j) = *if* i=j *then* ν *else* find(ρ,j)

D11. *if* tt *then* a *else* b = a

D12. *if* ff *then* a *ele* b = b

D13. $\mathcal{P}$(e) = $\mathcal{E}$(e, injE(arid))

D14. $\mathcal{E}$(i, injE(ρ)) = *if* ok(find(ρ,i)) *then* find(ρ,i) *else* errE

D15. $\mathcal{E}$(i, errU) = errE

D16. $\mathcal{E}$(i_1(i_2), injE(ρ)) = *if* ok(find(ρ,i_2)) *then* apply(find(ρ,i_1),find(ρ,i_2))
else errE

D17. $\mathcal{E}$(i_1(i_2), errU) = errE

D18. $\mathcal{E}$(**proc** (i):e, injE(ρ)) = fun(i,e,ρ)

D19. $\mathcal{E}$(**proc** (i):e, errU) = errE

D20. $\mathcal{E}$(**rec** i(j):e, injE(ρ)) = rec(i,j,e,ρ)

D21. $\mathcal{E}$(**rec** i(j):e, errU) = errE

D22. $\mathcal{E}$(c **res** e, ρ) = $\mathcal{E}$(e,$\mathcal{C}$(c,ρ))

D23. $\mathcal{E}$(b, injE(ρ)) = injB(b)

D24. $\mathcal{E}$(b, errU) = errE

D25. $\mathcal{E}$(0i, ρ) = ...

D26. $\mathcal{E}$($i_1 \Omega i_2$, ρ) = ...

D27. $\mathcal{C}$(i:=e, injE(ρ)) = *if* ok($\mathcal{E}$(e, injE(ρ))) *then* bind(ρ,i,$\mathcal{E}$(e, injE(ρ)))
else errU

D28. $\mathcal{C}$(i:=e, errU) = errU

```
D29.  𝒞(while i do c, injE(ρ)) = if ok(find(ρ,i)) then
                                     if find(ρ,i) then
                                        𝒞(while i do c, 𝒞(c, injE(ρ)))
                                     else injE(ρ)
                                  else errU
D30.  𝒞(while i do c, errU) = errU
D31.  𝒞(c1;c2, ρ) = 𝒞(c2, 𝒞(c1, ρ))
D32.  𝒞(if i then c1 else c2, injE(ρ)) = if ok(find(ρ,i)) then
                                             if find(ρ,i) then 𝒞(c1, injE(ρ))
                                             else 𝒞(c2, injE(ρ))
                                          else errU
D33.  C(if i then c1 else c2, errU) = errU
D34.  𝒞((), ρ) = ρ
```

<u>D - direct semantics presentation</u>

When showing Den to be a model of Th_D below, various 'degrees' of strictness will be important, so we repeat what we consider to be the standard definitions.

A function f is *strict* if $f(\bot) \equiv \bot$

A function f is *doubly strict* if $f(\bot) \equiv \bot$ and $f(\top) \equiv \top$

A function f is *completely strict* if $f(\bot) \equiv \bot$, $f(\top) \equiv \top$ and $f(?) \equiv ?$

A useful higher-order function which we shall employ is *dstrict* defined as follows

dstrict f $x \equiv \bot,\top$	if $x \equiv \bot,\top$
$f(x)$	otherwise.

So *dstrict* produces a doubly strict version of a function while *strict* produces a completely strict version. It is convenient to define a further equality predicate to complement ≡ and = defined earlier. Let == denote a doubly strict equality predicate. The usefulness of such a predicate is that it allows us to compare proper elements to ? without always returning ? (as = would do) yet returning ⊥ or ⊤ if the element is ⊥ or ⊤ (as ≡ would not do). This allows us to easily define a function in Den modeling the operator ok of D.

The correspondence between the sorts of D and the domains of the denotational semantics is the obvious one:

$Den_B = B$

$\mathrm{Den_E} = \mathrm{E} = \mathrm{B} + [\mathrm{E} \to \mathrm{E}] + \{?\}$
$\mathrm{Den_{Env}} = [\mathrm{Ide} \to \mathrm{E}]$
$\mathrm{Den_U} = [\mathrm{Ide} \to \mathrm{E}] + \{?\}$

The operator correspondence is as follows:

D	Den
injB: $\mathrm{B} \to \mathrm{E}$	*in* E
fun: $\mathrm{Ide} \times \mathrm{Exp} \times \mathrm{Env} \to \mathrm{E}$	$\lambda ie\rho.\ (\textit{dstrict}(\lambda\rho'.(\lambda\nu.\mathcal{E}[\![e]\!](\rho'[\nu/i]))$ $\textit{in}\ \mathrm{E})(\rho))$
rec: $\mathrm{Ide} \times \mathrm{Ide} \times \mathrm{Exp} \times \mathrm{Env} \to \mathrm{E}$	$\lambda ije\rho.\ (\textit{dstrict}(\lambda\rho'.\textit{fix}[\lambda\phi.\lambda\nu.$ $\mathcal{E}[\![e]\!](\rho[\phi\ \textit{in}\ \mathrm{E}/i][\nu/j])]\ \textit{in}\,\mathrm{E})(\rho))$
errE: $\to \mathrm{E}$	$?_\mathrm{E}$
apply: $\mathrm{E} \times \mathrm{E} \to \mathrm{E}$	$\lambda\nu_1\nu_2.\ (\nu_1 \vert [\mathrm{E} \to \mathrm{E}])(\nu_2)$
ok: $\mathrm{E} \to \mathrm{B}$	$\lambda\nu.\ \neg(\nu == ?_\mathrm{E})$
if_then_else: $\mathrm{B} \times \mathrm{E} \times \mathrm{E} \to \mathrm{E}$	$\lambda b\nu_1\nu_2.\ ((b \vert \mathrm{T}) \to \nu_1, \nu_2)$
arid: $\to \mathrm{Env}$	$\lambda i.?_\mathrm{E}$
bind: $\mathrm{Env} \times \mathrm{Ide} \times \mathrm{E} \to \mathrm{Env}$	$\lambda\rho i\nu.\ (\rho[\nu/i])$
find: $\mathrm{Env} \times \mathrm{Ide} \to \mathrm{E}$	$\lambda\rho i.\ (\rho[\![i]\!])$
injE: $\mathrm{Env} \to \mathrm{U}$	*in* U (injection of $[\mathrm{Ide} \to \mathrm{E}]$ into $[\mathrm{Ide} \to \mathrm{E}] + \{?\}$)
errU: $\to \mathrm{U}$	$?_\mathrm{U}$
if_then_else: $\mathrm{B} \times \mathrm{U} \times \mathrm{U} \to \mathrm{U}$	$\lambda b\rho_1\rho_2.\ ((b \vert \mathrm{T}) \to \rho_1, \rho_2)$
P: $\mathrm{Exp} \to \mathrm{E}$	$\mathcal{P}$
E: $\mathrm{Exp} \times \mathrm{U} \to \mathrm{E}$	$\mathcal{E}$
C: $\mathrm{Com} \times \mathrm{U} \to \mathrm{U}$	$\mathcal{C}$

The equations that follow are translations of the equations of D into the notation of Den and must be shown to be satisfied by Den to complete the proof that Den is a model of $\mathrm{Th_D}$. Though we refrain from including any detailed proofs, some commentary is offered for those equations that are not absolutely trivial.

D1. $\neg((b\ \textit{in}\ \mathrm{E}) == ?_\mathrm{E}) \equiv \textit{true}$
Note that this depends on there being no error element in B. If this is not considered satisfactory, then the equation ok(injB(b)) = tt could be changed to ok(injB(b)) = okB(b), and the new operator appropriately defined in D and implemented in Den.

D2. $\neg(\textit{dstrict}(\lambda\rho'.(\lambda\nu.\mathcal{E}[\![e]\!]\rho'[\nu/i])\ \textit{in}\ \mathrm{E})(\rho) == ?_\mathrm{E}) \equiv \textit{true}$

D3. $\neg(\textit{dstrict}\ \lambda\rho'.\textit{fix}[\lambda\phi\lambda\nu.\mathcal{E}[\![e]\!](\rho'[\phi\ \textit{in}\ E/i][\nu/j])]\ \textit{in}\ E)(\rho) == ?_E \equiv \textit{true}$

D2 and D3 hold because the function is made completely strict in ν at *application* time (see the clauses for $\mathcal{E}[\![i_0(i_1)]\!]\rho$ and $\mathcal{F}(i_1(i_2), \rho)$), and because $\rho \in Den_{Env} = [Ide \to E]$ and thus ρ is not $?_U$.

D4. $\neg(?_E == ?_E) \equiv \textit{false}$

D5. $(\textit{dstrict}(\lambda\rho.(\lambda\nu.\mathcal{E}[\![e]\!]\rho'[\nu/i])\ \textit{in}\ E)(\rho)|[E \to E])(\nu) \equiv \mathcal{E}[\![e]\!]\rho[\nu/i]$

D6. $(\textit{dstrict}(\lambda\rho'.\textit{fix}[\lambda\phi\lambda\nu.\mathcal{E}[\![e]\!](\rho'[\phi\ \textit{in}\ E/i][\nu/j])]\ \textit{in}\ E)(\rho)|[E \to E])(\nu)$

$\equiv \mathcal{E}[\![e]\!](\rho[\phi'\ \textit{in}\ E/i][\nu/j])$

where $\phi' \equiv \textit{fix}[\lambda\phi\lambda\nu.\mathcal{E}[\![e]\!](\rho[\phi'\ \textit{in}\ E/i][\nu/j])]$

Again in D5 and D6, $\rho \in Den_{Env} = [Ide \to E]$. These results are established by structural induction over Exp in the denotational model. In fact, the stronger result that $\mathcal{E}$ and $\mathcal{C}$ are completely strict is easier to establish. This appears as Lemma 6.2 in (Stoy, 1981).

D7. $(b\ \textit{in}\ E\ |\ [E \to E])(\nu) \equiv ?_E$

D8. $(?_E\ |\ [E \to E])(\nu) \equiv ?_E$

D9. $(\lambda i.?_E)(i) \equiv ?_E$

D10. $(\rho[\nu/i])[\![j]\!] \equiv i{=}j \to \nu, \rho[\![j]\!]$ $\qquad \rho \in Den_{Env}$

D11. $\textit{true} \to a,b \equiv a$

D12. $\textit{false} \to a,b \equiv b$

D13. $\mathcal{P}[\![e]\!] \equiv \mathcal{E}[\![e]\!](\lambda i.?_E\ \textit{in}\ U)$

D14. $\mathcal{E}[\![i]\!]\rho \equiv \neg(\rho[\![i]\!] == ?_E) \to \rho[\![i]\!], ?_E$ $\qquad \equiv \rho[\![i]\!]$

by definition of $\_\to\_,\_$

D15. $\mathcal{E}[\![i]\!]?_U \equiv ?_E$ $\qquad \equiv (?_U)[\![i]\!]$

D16. $\mathcal{E}[\![i_0(i_1)]\!]\rho \equiv \neg(\rho[\![i_1]\!] == ?_E) \to (\ \rho[\![i_0]\!]\ |\ [E \to E])(\rho[\![i_1]\!]), ?_E$

$\rho \in Den_{Env}$

D17. $\mathcal{E}[\![i_0(i_1)]\!]?_U \equiv ?_E$

D18. $\mathcal{E}[\![\textbf{proc}\ (i){:}e]\!]\rho \equiv \textit{dstrict}(\lambda\rho'.(\lambda\nu.\mathcal{E}[\![e]\!]\rho[\nu/i])\ \textit{in}\ E)(\rho)$

$\rho \in Den_{Env}$

D19. $\mathcal{E}[\![\textbf{proc}\ (i){:}e]\!]?_U \equiv ?_C$

D20. $\mathcal{E}[\![\textbf{rec}\ i(j){:}e]\!]\rho \equiv$

$\textit{dstrict}(\lambda\rho'.\textit{fix}[\lambda\phi\lambda\nu.\mathcal{E}[\![e]\!]\rho[\phi\ \textit{in}\ E/i][\nu/j]]\ \textit{in}\ E)(\rho)$

$\rho \in Den_{Env}$

D21. $\mathcal{E}[\![\textbf{rec}\ i(j){:}e]\!]?_U \equiv ?_E$

D22. $\mathcal{E}[\![c\ \textbf{res}\ e]\!]\rho \equiv \mathcal{E}[\![e]\!](\mathcal{C}[\![c]\!]\rho)$

D23. $\mathcal{E}[\![b]\!]\rho \equiv b\ \textit{in}\ E$ $\qquad \rho \in Den_{Env}$

D24. $\mathcal{E}[\![b]\!]?_U \equiv ?_E$

D25. $\mathcal{E}[\![0i]\!]\rho \equiv \ldots$

D26. $\mathcal{E}[\![i_0 \Omega i_1]\!]\rho \equiv \ldots$

D27. $\mathcal{C}[\![i{:=}e]\!]\rho \equiv \neg(\mathcal{E}[\![e]\!]\rho == ?_E) \rightarrow \rho[\mathcal{E}[\![e]\!]\rho / i], ?_U \qquad \rho \in Den_{Env}$

D28. $\mathcal{C}[\![i{:=}e]\!]?_U \equiv ?_U$

D29. $\mathcal{C}[\![\textbf{while } i \textbf{ do } c]\!]\rho \equiv$

$\neg(\rho[\![i]\!] == ?_E) \rightarrow (\rho[\![i]\!] \rightarrow \mathcal{C}[\![\textbf{while } i \textbf{ do } c]\!](\mathcal{C}[\![c]\!]\rho), \rho), ?_U$

$\rho \in Den_{Env}$

D30. $\mathcal{C}[\![\textbf{while } i \textbf{ do } c]\!]?_U \equiv ?_U$

D31. $\mathcal{C}[\![c_0;c_1]\!]\rho \equiv \mathcal{C}[\![c_1]\!](\mathcal{C}[\![c_0]\!]\rho)$

D32. $\mathcal{C}[\![\textbf{if } i \textbf{ then } c_0 \textbf{ else } c_1]\!]\rho \equiv$

$\neg(\rho[\![i]\!] == ?_E) \rightarrow \rho[\![i]\!] \rightarrow \mathcal{C}[\![c_0]\!]\rho, \mathcal{C}[\![c_1]\!]\rho, ?_U \qquad \rho \in Den_{Env}$

D33. $\mathcal{C}[\![\textbf{if } i \textbf{ then } c_0 \textbf{ else } c_1]\!]?_U \equiv ?_U$

D34. $\mathcal{C}[\![()]\!]\rho \equiv \rho$

4.6.3 The Congruence and a Theory Morphism

The definition of the congruence of the two semantic definitions of PL is not quite as straightforward on first inspection as the preceding examples, so some precursory discussion seems appropriate. In loose terms, the basic requirement is that we obtain the same result for a program under the evaluation implied by the two semantic definitions. Clearly, the concept of "the same result" will need to cater for the difference in function values for the two semantics; a string like $\ulcorner$**function** (i): e **in** ρ$\urcorner$ for the interpreter, an element of $[E \to E]$ for the denotational semantics. For this purpose we may define a function $e: E_{Int} \to E_{Den}$ as follows:

$$
\begin{array}{ll}
e(\nu) = \mathrm{b}\ \mathit{in}\ \mathrm{E} & \text{if } \nu = \mathit{Rep}(\mathrm{b}) \\
\qquad ?_{\mathrm{E}} & \text{if } \nu = \ulcorner\textbf{error}\urcorner \\
\qquad \mathit{dstrict}(\lambda\rho'.(\lambda\nu.\mathcal{E}[\![\mathrm{e}]\!]\rho'[\nu/\mathrm{i}])\ \mathit{in}\ \mathrm{E})(u(\rho)) & \\
 & \text{if } \nu = \ulcorner\textbf{function}(\mathrm{i})\text{: e } \textbf{in}\ \rho\urcorner \\
\qquad \mathit{dstrict}(\lambda\rho'.\mathit{fix}[\lambda\phi.\lambda\nu.\mathcal{E}[\![\mathrm{e}]\!]\rho'[\phi\ \mathit{in}\ \mathrm{E}/\mathrm{i}][\nu/\mathrm{j}]]\ \mathit{in}\ \mathrm{E})(u(\rho)) & \\
 & \text{if } \nu = \ulcorner\textbf{recfun}\ \mathrm{i(j)}\text{: e } \textbf{in}\ \rho\urcorner
\end{array}
$$

where $u(\rho) = (\lambda \mathrm{i}.e(\mathit{Select}(\rho,\mathrm{i})))\ \mathit{in}\ \mathrm{U}$

With these definitions, the congruence can be precisely given as:

$$e(\mathit{Interpret}(\ulcorner\textbf{eval}\ \mathrm{e}\ \textbf{in}\ \mathrm{arid};\ \textbf{done}\ \langle\rangle\urcorner)) = \mathcal{P}[\![\mathrm{e}]\!]$$

Since $\mathcal{P}$ is defined in terms of $\mathcal{E}$ and $\mathcal{C}$, the following two equations also suggest themselves:

$$e(\mathit{Interpret}(\ulcorner\textbf{eval}\ \mathrm{e}\ \textbf{in}\ \rho;\ \kappa\urcorner)) = \mathrm{send}(k(\kappa), \mathcal{E}[\![\mathrm{e}]\!](u(\rho)))$$
$$e(\mathit{Interpret}(\ulcorner\textbf{perform}\ \mathrm{c}\ \textbf{in}\ \rho;\ \theta\urcorner)) = \mathrm{run}(c(\theta), \mathcal{C}[\![\mathrm{c}]\!](u(\rho)))$$

But what are k and c ? There are no domains in the direct denotational semantics corresponding to expression continuations or command continuations, so there is no way that functions such as k and c can be instantiated. In the same vein, send and run are similar mysteries.

If we shelve these apparent difficulties for the moment and consider an algebraic formulation of the problem, the congruence statement is clearly something *like* the form required. First we have what looks like a theory morphism γ:

$\gamma(\mathcal{A}_C)(e) = \mathcal{A}_D(e)$
$\gamma(\mathcal{E}_C)(e,\rho,\kappa) = \text{send}(\kappa,\mathcal{E}_D(e,\rho))$
$\gamma(\mathcal{C}_C)(c,\rho,\theta) = \text{run}(\theta,\mathcal{C}_D(c,\rho))$

and a C-homomorphism Int $\rightarrow U_\gamma$(Den) consisting of the functions *e, u, k, c*. In this context, some clarification can be made: γ is not a theory morphism since send and run are operators from C, not D.

This suggests that D has been chosen inappropriately and we intend to present a new theory Th_{Dx} which is derived from Th_D by adding sorts K and C and appropriate operators to D. We repeat here the point made briefly in §3.1.2 that the crux of a semantic definition is the abstract syntax, the semantic domains and the valuations and we are therefore free to install them in any algebra we wish, selecting whatever other sorts and operator symbols we consider appropriate for our purposes without affecting the intended semantics.

It is also asserted in §3.2 that the semantics embodied in any model of Th_D or Th_{Dx} is the homomorphism indicated by *P,E* and *C* from the algebra defined by the abstract syntax to a semantic algebra with the same signature. Since the operators and sorts we intend to add to D do not "interfere" with any of the sorts of D, the semantic homomorphism in any model of Th_D remains unchanged no matter how that model is extended to be a model of Th_{Dx}. Given that view, the sorts and operators added in Dx can be viewed as "extra baggage".

As an analogy, consider adding a sort string and some typical operators to a complete specification of the data type Stack-of-Integers as given in §2.2. Provided the definitions are fairly standard (not pathological), we still have an implementation of a stack in the models of the theory.

The extensions to D to create Dx are as follows:
sort C

eval: $\text{Exp} \times \text{K} \rightarrow \text{C}$
perf: $\text{Com} \times \text{C} \rightarrow \text{C}$
run: $\text{C} \times \text{Env} \rightarrow \text{E}$

sort K

assign: Ide $\times$ Env $\times$ C $\rightarrow$ K

done: $\rightarrow$ K

send: K $\times$ E $\rightarrow$ E

<u>Equations</u>

x1. $\text{run}(\text{eval}(e,\kappa),\rho) = \text{send}(\kappa,\mathcal{E}(e,\text{injE}(\rho)))$

x2. $\text{run}(\text{perf}(c,\theta),\rho) = \text{run}(\theta,\mathcal{C}(c,\text{injE}(\rho)))$

x3. $\text{run}(\theta,\text{errU}) = \text{errE}$

x4. $\text{run}(\theta, \textit{if } b \textit{ then } \rho_1 \textit{ else } \rho_2) = \textit{if } b \textit{ then } \text{run}(\theta,\rho) \textit{ else } \text{run}(\theta,\rho_2)$

x5. $\text{send}(\text{assign}(i,\rho,\theta),\nu) = \textit{if } \text{ok}(\nu) \textit{ then } \text{run}(\theta,\text{bind}(\rho,i,\nu)) \textit{ else } \text{errE}$

x6. $\text{send}(\text{done},\nu) = \nu$

x7. $\text{send}(\kappa,\text{errE}) = \text{errE}$

x8. $\text{send}(\kappa, \textit{if } b \textit{ then } \nu_1 \textit{ else } \nu_2) = \textit{if } b \textit{ then } \text{send}(\kappa,\nu_1) \textit{ else } \text{send}(\kappa,\nu_2)$

<u>Dx - extensions to direct semantic presentation</u>

Given such extensions to D, the definition of γ above will serve as the basis of the definition of a theory morphism $\text{Th}_C \rightarrow \text{Th}_{Dx}$ (we wil prove it to be such later). The final requirement is that we must extend Den in a way that parallels the extension of D to Dx. Since, as pointed out above, such extensions have absolutely no effect on the semantics, *any* carriers and operators that satisfy the x-equations will do and as such we need not even bother specifying them. However the simplest choice is the trivial one with single point carriers for sorts C and K. Clearly the initial algebra semantics derived from any such extension to Den is identical to the initial algebra semantics derived from Den (§3.2.3).

A similar, though much simpler case can be given in terms of the addition expression example of §4.4. Instead of giving the congruence as $\mathcal{E}_D[\![e]\!] = (\mathcal{E}_S[\![e]\!]\zeta)\downarrow 1$, an equally acceptable statement would have been $\mathcal{E}_S[\![e]\!]\zeta = \mathcal{E}_D[\![e]\!]$ *cat* ζ. In the language of the algebraic presentations this would be $\mathcal{E}_S(e,s) = \text{push}(s,\mathcal{E}_D(e))$, which cannot be a theory morphism since push and $\mathcal{E}_D$ are not operators from the same theory. The solution is to extend the direct semantics presentation by adding a stack. Clearly the stack plays no part in the evaluation of expressions and thus has no effect on the semantics being defined. In 4.4 we avoided this problem by using $\mathcal{E}_D(e) = \text{top}\,(\mathcal{E}_S(e,s))$ where top is in some sense the *opposite* of push.

Such an alternative is clearly not available for our semantics of PL.

We now proceed with the details of the definition of the theory morphism γ: $Th_C \rightarrow Th_{DI}$.

<u>Sorts</u> :

$\gamma(B) = B$
$\gamma(E) = E$
$\gamma(Env) = Env$ (note: not U)
$\gamma(C) = C$
$\gamma(K) = K$

<u>Operator Symbols</u> :

$\gamma(injB)(b) = injB(b)$
$\gamma(fun)(i,e,\rho) = fun(i,e,\rho)$
$\gamma(rec)(i,j,e,\rho) = rec(i,j,e,\rho)$
$\gamma(errE) = errE$
$\gamma(ok)(\nu) = ok(\nu)$
$\gamma(arid) = arid$
$\gamma(bind)(\rho,i,\nu) = bind(\rho,i,\nu)$
$\gamma(find)(\rho,i) = find(\rho,i)$
$\gamma(has)(\rho,i) = ok(find(\rho,i))$
$\gamma(eval)(e,\kappa) = eval(e,\kappa)$
$\gamma(perf)(c,\theta) = perf(c,\theta)$
$\gamma(run)(\theta,\rho) = run(\theta,\rho)$
$\gamma(assign)(i,\rho,\theta) = assign(i,\rho,\theta)$
$\gamma(done) = done$
$\gamma(send)(\kappa,\nu) = send(\kappa,\nu)$
$\gamma(call)(\nu_1,\nu_2,\kappa) = send(\kappa,apply(\nu_1,\nu_2))$
$\gamma(\mathcal{P})(e) = \mathcal{P}(e)$
$\gamma(\mathcal{E})(e, \rho,\kappa) = send(\kappa,\mathcal{E}(e, injE(\rho)))$
$\gamma(\mathcal{C}(c, \rho,\theta) = run(\theta,\mathcal{C}(c, injE(\rho)))$

The equations that follow must be shown to hold in Th_{DI} to establish that γ is a theory morphism. Brief proof outlines are given, though they are all very straightforward.

$\gamma(1)$ find(arid,i) = errE
$\gamma(2)$ find(bind(ρ,i,ν),j) = *if* i=j *then* ν *else* find(ρ,j)

$\gamma(3)$ ok(injB(b)) = tt

$\gamma(4)$ ok(fun(i,e,ρ)) = tt

$\gamma(5)$ ok(rec(i,j,e,ρ)) = tt

$\gamma(6)$ ok(errE) = ff

$\gamma(7)$ ok(find(ρ,i)) = ok(find(ρ,i))

$\gamma(8)$ run(eval(e,κ),ρ) = send(κ,$\mathcal{E}$(e, injE(ρ)))

$\gamma(9)$ run(perf(c,θ),ρ) = run(θ,$\mathcal{C}$(c, injE(ρ)))

$\gamma(10)$ send(assign(i,ρ,θ),ν) = *if* ok(ν) *then* run(θ,bind(ρ,i,ν))
else errE

$\gamma(11)$ send(done,ν) = ν

$\gamma(12)$ send(κ,errE) = errE

$\gamma(13)$ send(κ,apply(fun(i,e,ρ),ν)) = send(κ,$\mathcal{E}$(e,injE(bind(ρ,i,ν))))

$\gamma(14)$ send(κ,apply(rec(i,j,e,ρ),ν)) =
send(κ,$\mathcal{E}$(e, injE(bind(bind(ρ,i,rec(i,j,e,ρ)),j,ν))))

$\gamma(15)$ send(κ,apply(injB(b),ν)) = errE

$\gamma(16)$ send(κ,apply(errE,ν)) = errE

$\gamma(17)$ $\mathcal{P}$(e) = send(done,$\mathcal{E}$(e, injE(arid)))

$\gamma(18)$ send(κ,$\mathcal{E}$(i, injE(ρ))) = *if* ok(find(ρ,i)) *then* send(κ,find(ρ,i))
else errE

$\gamma(19)$ send(κ,$\mathcal{E}$(i_1(i_2), injE(ρ))) =
if ok(find(ρ,i_1)) *then* *if* ok(find(ρ,i_2)) *then*
send(κ,apply(find(ρ,i_1), find(ρ,i_2)))
else errE *else* errE

$\gamma(20)$ send(κ,$\mathcal{E}$(**proc** (i):e, injE(ρ))) = send(κ,fun(i,e,ρ))

$\gamma(21)$ send(κ,$\mathcal{E}$(**rec** i(j):e, injE(ρ))) = send(κ,rec(i,j,e,ρ))

$\gamma(22)$ send(κ,$\mathcal{E}$(c **res** e, injE(ρ))) = run(eval(e,κ),$\mathcal{C}$(c, injE(ρ)))

$\gamma(23)$ send(κ,$\mathcal{E}$(b, injE(ρ))) = send(κ,injB(b))

$\gamma(24)$ send(κ,$\mathcal{E}$(0_i, injE(ρ))) = ...

$\gamma(25)$ send(κ,$\mathcal{E}$($i_1 \Omega i_2$, injE(ρ))) = ...

$\gamma(26)$ run(θ,$\mathcal{C}$(i:=e, injE(ρ))) = send(assign(i,ρ,θ),$\mathcal{E}$(e, injE(ρ)))

$\gamma(27)$ run(θ,$\mathcal{C}$(**while** i **do** c, injE(ρ))) =
if ok(find(ρ,i)) *then* *if* find(ρ,i) *then*
run(perf(**while** i **do** c,θ),$\mathcal{C}$(c, injE(ρ)))
else run(θ,ρ) *else* errE

$\gamma(28)$ run(θ,$\mathcal{C}$(c_1;c_2, injE(ρ))) = run(perf(c_2,θ),$\mathcal{C}$(c_1, injE(ρ)))

$\gamma(29)$ run(θ,$\mathcal{C}$(**if** i **then** c_1 **else** c_2, injE(ρ))) =
if ok(find(ρ,i)) *then* *if* find(ρ,i) *then*
run(θ,$\mathcal{C}$(c_1,injE(ρ)))
else run(θ,$\mathcal{C}$(c_2,injE(ρ))) *else* errE

$\gamma(30)$ run(θ, $\mathcal{C}$((), injE(ρ))) = run(θ,injE(ρ))

The proof outlines follow:

"find" equations -

$\gamma(1)$: D9
$\gamma(2)$: D10

"ok" equations -

$\gamma(3)$: D1
$\gamma(4)$: D2
$\gamma(5)$: D3
$\gamma(6)$: D4
$\gamma(7)$: trivial

"run" equations -

$\gamma(8)$: x1
$\gamma(9)$: x2

"send" equations -

$\gamma(10)$: x5
$\gamma(11)$: x6
$\gamma(12)$: x7

"call" equations -

$\gamma(13)$: D5
$\gamma(14)$: D6
$\gamma(15)$: D7, x7
$\gamma(16)$: D8, x7

"$\mathcal{P}$" equations -

$\gamma(17)$: D13, x6

"$\mathcal{E}$" equations -

$\gamma(18)$: D14, x8
$\gamma(19)$: D16, x8
$\gamma(20)$: D18
$\gamma(21)$: D20
$\gamma(22)$: D22, x1
$\gamma(23)$: D23
$\gamma(24)$: D25 ...
$\gamma(25)$: D26 ...

"$\mathcal{C}$" equations -

$\gamma(26)$: D27, x4, x5, x3
$\gamma(27)$: D29, x4, x2, x3

$\gamma(28)$: D31, x2
$\gamma(29)$: D32, x4, x3
$\gamma(30)$: D34

Thus we have shown γ to be a theory morphism and Int to be initial in Alg_C. There must therefore be a homomorphism from Int to U_γ(Den), and this constitutes the semantic congruence.

Chapter 5
Compiler Correctness

It was noted earlier (§4.1.5) that certain similarities appear to exist between our notion of semantic congruence and the algebraic approach to establishing the correctness of compilers (or "translation algorithms" as practitioners would be more inclined to call them). It is this correspondence we wish to exploit in the current chapter.

The first published attempt to formalise what it means for a compiler to be *correct* and then to follow through an example proof was by McCarthy & Painter (1967) where they treated the translation of arithmetic expressions into sequences of instructions for a single-address accumulator machine. However, it was Burstall & Landin (1969) that first suggested the (explicit) use of algebra for such a proof, based on indentifying abstract syntax with the word algebra for some signature. The idea was further developed by Milner & Wehrauch (1972) and by Morris (1972,1973), leading to the diagrammatic representation known generally as the "Morris Square".

Essentially the idea is that given a source language whose abstract syntax can be identified with some signature we may construct the following diagram.

$$\begin{tikzcd}
\text{source language} & S \arrow[r, "\text{compile}", "\gamma"'] \arrow[d, "\text{semantics}"', "\phi"] & T \arrow[d, "\psi"', "\text{semantics}"] & \text{target language} \\
\text{source meanings} & M \arrow[r, "\varepsilon", "\text{encode}"'] & U & \text{target meanings}
\end{tikzcd}$$

In this diagram S, M, T and U are all Ω-algebras, S in particular is the Ω-word algebra T_Ω and γ, ϕ, ψ, ε are all Ω-homomorphisms. In this context a proof of compiler correctness consists of a proof that the above diagram commutes; in other words $\phi.\varepsilon = \gamma.\psi$. Clearly, since S is the initial Ω-algebra, ϕ, $\phi.\varepsilon$ and $\gamma.\psi$ are all unique and the proof is reduced to showing that there *is* some homomorphism ε: M $\rightarrow$ U. Actually, Morris' original diagram has a decode homomorphism δ: $U \to M$ rather than ε, though in the text of (Morris, 1973) ε: $M \to U$. is used. We will comment on the appropriateness of the choice between ε and δ in §5.4.

An important part of the work involved is left out of the above diagram, however. T is the algebra of programs that may possibly be produced by the compiler, rather than the algebra of programs that may be written in the target language. If the abstract syntax of the target language is identified with some signature Σ, then its semantics will be given by a homomorphism from the initial Σ-algebra (denoted T_0 by Morris) to some Σ-algebra of target meanings, U_0. The Ω-algebras T and U are then *derived* from T_0 and U_0, generally in an ad hoc though enlightened manner. Burstall & Landin (1969) however, explicity include the derivors in their diagram and we feel this to be more informative since the compiler description is embodied in the derivation of T from T_0.

$$\begin{tikzcd}
S \arrow[r, "\gamma"] \arrow[d, "\phi"'] & T \arrow[d, "\psi"] & T_0 \arrow[l, dashed, "\pi"'] \arrow[d, "\psi_0"] \\
M \arrow[r, "\epsilon"'] & U & U_0 \arrow[l, dashed, "\pi"']
\end{tikzcd}$$

The fact that the two derivor arrows are both labelled π implies that U is derived from U_0 in the same way as T is derived from T_0. This in turn ensures that ψ is a homomorphism (since ψ_0 is a homomorphism), a fact presented under the title "homomorphism of restriction lemma" in Burstall & Landin (1969). In our framework, π is the derivor U_τ associated with some signature morphism τ: $\Sigma \to \overline{\Omega}$.

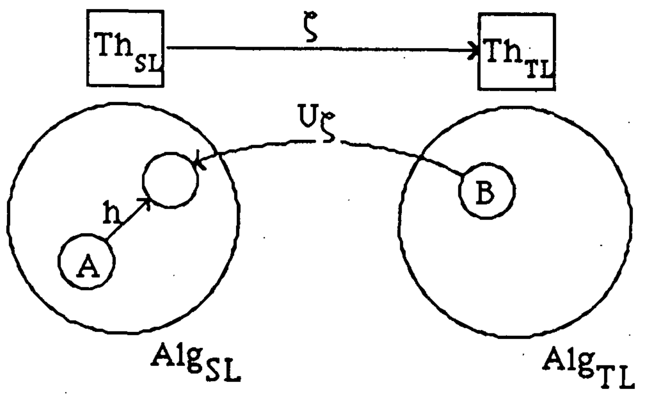
Further work on this approach to compiler correctness has included the contribution by Thatcher, Wagner & Wright (1979) to clarify the construction and semantics of flow charts (as Morris used for T_0 and U_0) by using a more categorical approach involving the notion of continuous algebraic theories (Wagner, Wright, Goguen & Thatcher, 1976). Henson (1983) extends the technique to source languages that require continuation semantics, though that work does not adhere completely to the commuting square advice. Other papers that approach compiler correctness from an algebraic viewpoint include Mosses (1980) and Wand (1980a) but they employ quite different techniques to the traditional one discussed above.

Returning to the proposed connection between our work on semantic congruences and the notion of compiler correctness, we consider the following diagram derived from the one given in §4.1.5 to represent the (homomorphic) congruence of semantic models.

$$\begin{tikzcd}
T_\Omega \arrow[r, "\cong"] \arrow[d, "g"'] & T_\Omega \arrow[d, "k"] \\
S_\Omega \arrow[r, "h"'] & S'_\Omega
\end{tikzcd}$$

Clearly the relation represented by the above diagram can be simply and directly extended to treat compiler correctness by discarding the requirement that the syntactic signatures be identical, thus generalising the isomorphism $T_\Omega \to T_\Omega$ to a homomorphism $T_\Omega \to A_\Omega$ for some algebra of target programs A_Ω.

Since in this dissertation we prefer to treat semantic definitions as algebras with the (homomorphic) semantic valuations included as operators of the signature, in contrast with the initial algebra semantics approach, a more direct representation of our compiler correctness proof technique is as follows:



Now this is exactly the diagram of §4.1.5 except that we no longer insist that SL and TL present the semantics of the same language.

In summary, our approach to establishing the correctness of a compiler is as follows. Given a source language presentation SL and a particular model of Th_{SL}, say A, and a target language presentation TL and a particular model of Th_{TL}, say B, first define a theory morphism $\zeta: Th_{SL} \to Th_{TL}$. Now derive the SL-algebra $U_\zeta(B)$ and show there is a homomorphism $h: A \to U_\zeta(B)$. Intuitively ζ is the compiler *definition*, while h restricted to the syntactic part of the signature of SL is the compilation *process*. Relating this back once again to the diagram of Burstall & Landin (1969) reproduced above, h incorporates the homomorphisms γ and ε while U_ζ incorporates both applications of the derivor π. The semantic homomorphisms ϕ, ψ_0 and ψ are respectively part of the algebras A, B and $U_\zeta(B)$. It is inappropriate to appeal to the "homomorphism of restrictions lemma", nor is there any need to, since ψ is a homomorphism by virtue of the fact that $U_\zeta(B)$ is a model of Th_{SL} rather than any property of the way it is derived from B.

The body of this chapter follows an example proof for the source language and target language described in the next section. Our example differs markedly from those of Morris (1973) and Thatcher et al (1979) not only in the way we describe the semantics, but most importantly in the style of our target language. In contrast to the use of flow charts in those papers, our target language is much more like an assembler language with flow of control being wrought by branch instructions and as such is much more "realistic".

5.1 Semantic Presentations of a Source Language and a Target Language

The source language we will deal with is a simplified version of the one treated by Morris (1973) and it shall be referred to as SL. An SL program is a sequence of commands which may be assignments, conditionals, while loops or dummies. The expressions of the language have boolean values and the value of an uninitialised variable is distinguished as an error, though such an access does not affect the continued execution of the program.

We now proceed with a presentation SLP of the semantics of SL. In the sequel the part of the signature that describes the abstract syntax of SL will be referred to under the name Σ, so that Σ contains only the sorts Program, Com and Exp (and Ide) and the operators among them.

<u>Signature</u>
syntactic sort Program
    prog: Com → Program
syntactic sort Com
    continue: → Com
    seq: Com × Com → Com
    assign: Ide × Exp → Com
    if: Exp × Com × Com → Com
    while: Exp × Com → Com
syntactic sort Exp
    var: Ide → Exp
    true: → Exp
    false: → Exp
    not: Exp → Exp
    and: Exp × Exp → Exp
    or: Exp × Exp → Exp
sort U
    arid: → U
    bind: U × Ide × Bool → U
    find: U × Ide → Bool
sort Bool
    tt: → Bool
    ff: → Bool

$\neg$: Bool $\to$ Bool

$\wedge$: Bool $\times$ Bool $\to$ Bool

$\vee$: Bool $\times$ Bool $\to$ Bool

err: $\to$ Bool

$\mathcal{P}$: Program $\to$ U

$\mathcal{C}$: Com $\times$ U $\to$ U

$\mathcal{E}$: Exp $\times$ U $\to$ Bool

Equations

1. find(arid, i) = err
2. find(bind(ρ,i,b),j) = *if* i=j *then* b *else* find(ρ,j)
3. $\neg$(tt) = ff
4. $\neg$(ff) = tt
5. $\neg$(err) = err
6. tt $\wedge$ b = b
7. ff $\wedge$ b = *if* b=err *then* err *else* ff
8. err $\wedge$ b = err
9. tt $\vee$ b = *if* b=err *then* err *else* tt
10. ff $\vee$ b = b
11. err $\vee$ b = err
12. $\mathcal{P}$(prog(c)) = $\mathcal{C}$(c,arid)
13. $\mathcal{C}$(continue, ρ) = ρ
14. $\mathcal{C}$(seq(c_1, c_2), ρ) = $\mathcal{C}$(c_2, $\mathcal{C}$(c_1, ρ)
15. $\mathcal{C}$(assign(i,e), ρ) = bind(ρ,i,$\mathcal{E}$(e, ρ))
16. $\mathcal{C}$(if(e,c_1,c_2), ρ) = *if* $\mathcal{E}$(e, ρ) *then* $\mathcal{C}$(c_1, ρ) *else* $\mathcal{C}$(c_2, ρ)
17. $\mathcal{C}$(while(e,c), ρ) = *if* $\mathcal{E}$(e, ρ) *then* $\mathcal{C}$(while(e,c), $\mathcal{C}$(c, ρ)) *else* ρ
18. $\mathcal{E}$(var(i), ρ) =find(ρ,i)
19. $\mathcal{E}$(true, ρ) = tt
20. $\mathcal{E}$(false, ρ) = ff
21. $\mathcal{E}$(not(e), ρ) = $\neg\mathcal{E}$(e, ρ)
22. $\mathcal{E}$(and(e_1,e_2), ρ) = $\mathcal{E}$(e_1, ρ) $\wedge$ $\mathcal{E}$($e_2\rho$)
23. $\mathcal{E}$(or(e_1,e_2), ρ) = $\mathcal{E}$(e_1, ρ) $\vee$ $\mathcal{E}$(e_2, ρ)

SLP - semantic presentation for source language SL

As mentioned above, the target language TL we will deal with can be viewed as a simple assembler language, basically made up of instructions that move values (booleans for our purpose) about in locations with the sequence of instruction execution being controlled by labels and jumps (both conditional and unconditional). The machine upon which they operate consists only of a store with no registers and no explicit stack, so the operators of the language are permitted to manipulate any of the locations in the store. We intend the compiler to implement an implicit stack as a sequence of locations, hence the presentation of TL has the (otherwise rather mysterious) notion of two constants of sort Loc, f0 being the base location of the segment where the values of variables are stored and s0 being the base location of the segment set aside for the stack.

We now proceed with a presentation TLP of the semantics of TL. The part of the signature that describes the abstract syntax of TL will be referred to as Ω, thus Ω consists only of the sorts Programs, Instr, Loc and Tag and the operators among them. We defer discussion and consideration of actual SLP and TLP models to §5.3.

<u>Signature</u>

syntactic sort Program

  prog: Instr → Program

syntactic sort Instr

  dummy: → Instr

  seq: Instr × Instr → Instr

  move: Loc × Loc → Instr

  ldt: Loc → Instr

  ldf: Loc → Instr

  com: Loc → Instr

  or: Loc × Loc → Instr

  lab: Tag → Instr

  br: Tag → Instr

  brt: Loc × Tag → Instr

syntactic sort Tag

  t0: → Tag

  nxt: Tag → Tag

syntactic sort Loc

  f0: → Loc

  s0: → Loc

  nxt: Loc → Loc

sort Store

empty: $\rightarrow$ Store

set: Store $\times$ Loc $\times$ Bool $\rightarrow$ Store

val: Store $\times$ Loc $\rightarrow$ Bool

sort Bool

tt: $\rightarrow$ Bool

ff: $\rightarrow$ Bool

$\neg$: Bool $\rightarrow$ Bool

$\wedge$: Bool $\times$ Bool $\rightarrow$ Bool

$\vee$: Bool $\times$ Bool $\rightarrow$ Bool

err: $\rightarrow$ Bool

sort Env

arid: $\rightarrow$ Env

bind: Env $\times$ Tag $\times$ C $\rightarrow$ Env

find: Env $\times$ Tag $\rightarrow$ C

bindall: Env $\times$ Taglist $\times$ Clist $\rightarrow$ Env

sort C

null: $\rightarrow$ C

fail: $\rightarrow$ C

ass: Loc $\times$ Bool $\rightarrow$ C

apply: C $\times$ Store $\rightarrow$ C

sort Clist

newc: $\rightarrow$ Clist

catc: C $\times$ Clist $\rightarrow$ Clist

sort Taglist

newt: $\rightarrow$ Taglist

catt: Tag $\times$ Taglist $\rightarrow$ Taglist

$\mathcal{P}$: Program $\rightarrow$ Store

$\mathcal{I}$: Instr $\times$ Env $\times$ C $\rightarrow$ C

$\mathcal{D}$: Instr $\rightarrow$ Env

$\mathcal{T}$: Instr $\times$ Taglist $\rightarrow$ Taglist

$\mathcal{M}$: Instr $\times$ Env $\times$ C $\times$ Clist $\rightarrow$ Clist

<u>Equations</u>

1. val(empty,ℓ) = err
2. val(set(σ,ℓ_1,b),ℓ_2) = *if* $\ell_1 = \ell_2$ *then* b *else* val(σ,ℓ_2)
3. find(arid,t) = fail
4. find(bind(ρ,t_1,θ),t_2) = *if* $t_1 = t_2$ *then* θ *else* find(ρ,t_2)
5. $\neg$(tt) = ff

6. $\neg(\mathrm{ff}) = \mathrm{tt}$
7. $\neg(\mathrm{err}) = \mathrm{err}$
8. $\mathrm{tt} \wedge \mathrm{b} = \mathrm{b}$
9. $\mathrm{ff} \wedge \mathrm{b} =$ *if* b=err *then* err *else* ff
10. $\mathrm{err} \wedge \mathrm{b} = \mathrm{err}$
11. $\mathrm{tt} \vee \mathrm{b} =$ *if* b=err *then* err else tt
12. $\mathrm{ff} \vee \mathrm{b} = \mathrm{b}$
13. $\mathrm{err} \vee \mathrm{b} = \mathrm{err}$
14. $\mathrm{apply}(\mathrm{null},\sigma) = \sigma$
15. $\mathrm{apply}(\mathrm{ass}(\ell,\mathrm{b}),\sigma) = \mathrm{set}(\sigma,\ell,\mathrm{b})$
16. $\mathrm{bindall}(\rho,\mathrm{newt}, \mathrm{cl}) = \rho$
17. $\mathrm{bindall}(\rho,\mathrm{catt}(\mathrm{t},\mathrm{tl}),\mathrm{catc}(\theta,\mathrm{cl})) = \mathrm{bindall}(\mathrm{bind}(\rho,\mathrm{t},\theta),\mathrm{tl},\mathrm{cl})$
18. $\mathcal{P}(\mathrm{prog}(\mathrm{c})) = \mathrm{apply}(\mathcal{I}(\mathrm{c},\mathcal{D}(\mathrm{c}),\mathrm{null}),\mathrm{empty})$
19. $\mathcal{D}(\mathrm{c}) = \mathrm{bindall}(\mathrm{arid},\mathcal{T}(\mathrm{c},\mathrm{newt}),\mathcal{M}(\mathrm{c}, \mathcal{D}(\mathrm{c}),\mathrm{null},\mathrm{newc}))$
20. $\mathcal{T}(\mathrm{seq}(\mathrm{c}_1,\mathrm{c}_2), \mathrm{tl}) = \mathcal{T}(\mathrm{c}_2,\mathcal{T}(\mathrm{c}_1, \mathrm{tl})$
21. $\mathcal{T}(\mathrm{lab}(\mathrm{t}), \mathrm{tl}) = \mathrm{catt}(\mathrm{t},\mathrm{tl})$
22. $\mathcal{T}(.\mathrm{other}., \mathrm{tl}) = \mathrm{tl}$
23. $\mathcal{M}(\mathrm{seq}(\mathrm{c}_1,\mathrm{c}_2), \rho,\theta,\mathrm{cl}) = \mathcal{M}(\mathrm{c}_2, \rho,\theta,\mathcal{M}(\mathrm{c}_1, \rho, \mathcal{I}(\mathrm{c}_2, \rho,\theta),\mathrm{cl}))$
24. $\mathcal{M}(\mathrm{lab}(\mathrm{t}), \rho,\theta,\mathrm{cl}) = \mathrm{catc}(\theta,\mathrm{cl})$
25. $\mathcal{M}(.\mathrm{other}., \rho,\theta,\mathrm{cl}) = \mathrm{cl}$
26. $\mathcal{I}(\mathrm{dummy}, \rho,\theta) = \theta$
27. $\mathcal{I}(\mathrm{seq}(\mathrm{c}_1,\mathrm{c}_2), \rho,\theta) = \mathcal{I}(\mathrm{c}_1, \rho, \mathcal{I}(\mathrm{c}_2, \rho,\theta))$
28. $\mathrm{apply}(\mathcal{I}(\mathrm{move}(\ell_1,\ell_2), \rho,\theta),\sigma) = \mathrm{apply}(\theta\ \mathrm{set}(\sigma,\ell_2,\mathrm{val}(\sigma,\ell_1)))$
29. $\mathrm{apply}(\mathcal{I}(\mathrm{ldt}(\ell), \rho,\theta),\sigma) = \mathrm{apply}(\theta,\mathrm{set}(\sigma,\ell,\mathrm{tt}))$
30. $\mathrm{apply}(\mathcal{I}(\mathrm{ldf}(\ell), \rho,\theta),\sigma) = \mathrm{apply}(\theta,\mathrm{set}(\sigma,\ell,\mathrm{ff}))$
31. $\mathrm{apply}(\mathcal{I}(\mathrm{com}(\ell), \rho,\theta),\sigma) = \mathrm{apply}(\theta,\mathrm{set}(\sigma,\ell,\neg(\mathrm{val}(\sigma,\ell))))$
32. $\mathrm{apply}(\mathcal{I}(\mathrm{or}(\ell_1,\ell_2), \rho,\theta),\sigma) = \mathrm{apply}(\theta,\mathrm{set}(\sigma,\ell_2,\mathrm{val}(\sigma,\ell_1) \vee \mathrm{val}(\sigma,\ell_2)))$
33. $\mathcal{I}(\mathrm{lab}(\mathrm{t}), \rho,\theta) = \theta$
34. $\mathcal{I}(\mathrm{br}(\mathrm{t}), \rho,\theta) = \mathrm{find}(\rho,\mathrm{t})$
35. $\mathrm{apply}(\mathcal{I}(\mathrm{brt}(\ell,\mathrm{t}), \rho,\theta),\sigma) = \mathrm{apply}(($*if* $\mathrm{val}(\sigma,\ell)$ *then*
    $\mathrm{find}(\rho,\mathrm{t})$ *else* $\theta),\sigma)$

<u>TLP - semantic presentation for target language TL</u>

Briefly, we anticipate the compiler from SL to TL to have the following overall features. First, for simplicity we assume there to be a pre-determined homomorphic relation between identifiers (Ide) and locations (Loc) in the segment whose base address is $\ell 0$, though we will consider replacing this with "symbol-table" information in §5.3. Second, expressions will be evaluated on an implicit stack of locations in the segment whose base address is s0. Further, the top-of-stack pointer is to be simulated at *compile* time, so expressions will be translated into instruction sequences with absolute addresses.

5.2 Compilers as Homomorphisms

Were we to closely follow the advice of Morris for defining our compiler and verifying its correctness, the first requirement would be to derive a Σ-algebra (Σ is the abstract syntax of SL) from T_Ω (Ω is the abstract syntax of TL), thus giving a homomorphism γ: S → T where S is T_Σ, the algebra of SL programs and T is the derived Σ-algebra ($U_\tau(T_\Omega)$ for some signature morphism τ: $\Sigma \to \Omega$). This is basically the top line of the commuting square diagram of the chapter introduction.

$$\begin{tikzcd}
\overset{T_\Sigma}{S} \arrow[r] \arrow[d, dash, dashed] & \overset{U_\tau(T_\Omega)}{T} \arrow[d, dash, dashed] & \overset{T_\Omega}{T_O} \arrow[l, dashed, "U_\tau"'] \arrow[d, dash, dashed] \\
{} & {} & {}
\end{tikzcd}$$

Even given the simple and fairly standard features outlined in §5.1 of the compiler we intend to construct, this task appears to be quite impossible.

The following example pinpoints one of the problem areas. Consider the (intended) translation of the SL commands **i := a or (not(b))** and **i := (not(b)) or c**. The code sequences we expect to produce are something like the following, assuming for concreteness that a, b, c and i are mapped to locations ℓ1, ℓ2, ℓ3 and ℓ4 respectively.

i := a or (not(b))	move(ℓ1,s0)
	move(ℓ2,s1)
	com(s1)
	or(s1,s0)
	move(s0,ℓ4)

i := (not(b)) or c	move(ℓ2,s0)
	com(s0)
	move(ℓ3,s1)
	or(s1,s0)
	move(s0,ℓ4)

If we now concentrate on the coding of the sub-expression **not(b)**, it can be seen that in the first case the corresponding instruction sequence is

move(ℓ2,s1)
com(s1)

whereas in the second case the corresponding instruction sequence is

move(ℓ2,s0)
com(s0).

Clearly, if it is possible to obtain two different target programs for the same source (sub)-expression we cannot have a homomorphism from the algebra of source program s to the algebra of compiler-produced target programs T. Hence it is impossible to express such a compiler by deriving an appropriate Σ-algebra from T_Ω or, equivalently, by defining a signature morphism $\tau: \Sigma \to \Omega$.

The intuitive explanation for this problem can be found in the fact that in addition to the structure of the source expression being translated, the compiler needs to be informed as to where the current top-of-stack is. In other words, each expression is translated into instructions that leave the result *in some location*, and the instructions vary according to the choice of location. Thus, the compilation function (for expressions at least) may be thought of as taking a source expression *and* a location, producing target instruction sequences such as follows:

compile(var(i),ℓ) = move(compile(i),ℓ)
compile(not(e),ℓ) = seq(compile(e,ℓ),com(ℓ))
compile(or(e_1,e_2),ℓ) = seq(compile(e_1,ℓ),
    compile(e_2,nxt(ℓ)),or(nxt(ℓ),ℓ)))

(Note that here and elsewhere we allow the slight abuse of notation whereby seq takes any number of arguments to save us writing seq(c,seq(c,c)) and so on.)

Now the definition of compile above is roughly what we want for a signature morphism $\tau : \Sigma \to \Omega$, but falls short in that locations are not permissible arguments since they do not occur in Σ.

This situation is reminiscent of the one that arose in §4.6.3 where the desired congruence did not appear to consist of a theory morphism for similar reasons: sorts and operators of the source theory were being referred to as though they were present in the target theory. That is

$\gamma(\mathcal{E}_C)(e,\rho,\kappa)$ = send(κ,$\mathcal{E}_D(e,\rho)$)
$\gamma(\mathcal{C}_C)(c,\rho,\theta)$ = run(θ, $\mathcal{C}_D(c,\rho)$)

where send, run, κ and c occur in the continuation semantics presentation but not in the direct semantics presentation. Our solution in that case was to extend the target theory by the addition of sorts and operators with the aim of making γ a theory morphism from the source theory to that new target theory. Clearly the semantic valuation operators and the semantic sorts are unaffected by such additions so the semantics of the language is unchanged (see §4.6.3).

Since the work in this chapter is intended to be based on exploiting the observed similarities between our notions of semantic congruence and compiler correctness, we are naturally guided to seek a solution for the present problem that is similar to the approach taken in §4.6.

Our intention in §5.3 is to make appropriate extensions to Ω (and TLP) so that a signature morphism (later a theory morphism) can be found that is the analogue of the compiler we have in mind. Given that the function compile: $\mathrm{Exp} \times \mathrm{Loc} \rightarrow \mathrm{Instr}$ roughly expresses the pattern upon which we are basing the compiler definition, the natural choice for a signature morphism would seem to be of the form τ: $\mathrm{Exp} \rightarrow [\mathrm{Loc} \rightarrow \mathrm{Instr}]$ suggesting that we need to extend Ω by adding a sort that adequately represents $[\mathrm{Loc} \rightarrow \mathrm{Instr}]$. This is the basic approach that we intend to take in the next section, though further similar extensions will be required when considering $\tau(\mathrm{Com})$.

5.3 A Compiler

In this section we intend to detail an extension of Ω to Ω' (and TLP to TLPX) so that a satisfactory signature morphism $\tau: \Sigma \rightarrow \Omega'$ may be defined such that it expresses the translation algorithm we intend. We will discuss the requirements for expressions first, along the lines hinted at in the previous section and then treat commands in an analogous way.

The first extension to Ω we require for τ to parallel the effect of "compile" (§5.2) is the addition of a sort we shall call LtoInstr to represent functions from locations to instructions, [Loc $\rightarrow$ Instr]. The necessary operator symbols will include the notion of applying an abstraction (the objects of sort LtoInstr) to a location to produce an instruction. Recall that for the compilation of expressions we need to provide the location in which the compiled instructions are to leave the resultant value. Thus, the abstraction will be resolved (grounded) by supplying a particular location when compiling commands that explicitly contain an expression.

On the other hand, the choice of operator symbols to act as constructors of LtoInstr is not so clear cut. The most direct approach is to include an operator symbol for each instruction type, such as:

ldf-abstr: $\rightarrow$ LtoInstr
br: Tag $\rightarrow$ LtoInstr
brt: Tag $\rightarrow$ LtoInstr
seq: LtoInstr $\times$ LtoInstr $\rightarrow$ LtoInstr

and so on. However, complications arise for those instructions which refer to two locations, such as or: Loc $\times$ Loc $\rightarrow$ Instr. The options are to add a further sort representing [Loc $\rightarrow$ [Loc $\rightarrow$ Instr]] which is general, but long-winded, or make the observation that the compiler only ever produces instructions like or(nxt(ℓ),ℓ) for some ℓ, thus reducing it again to an abstraction on a single location. Another difficulty is evident from inspection of (for example) the compilation of e_1 **or** e_2 (§5.2), where we have a sequence of instructions where a location, say ℓ, is supplied to the first instruction, while nxt(ℓ) is supplied to the second instruction. The solution here is to add a further operator symbol next: LtoInstr $\rightarrow$ LtoInstr with the understanding that the application of next(α) to some location ℓ is the same as the application of α to nxt(ℓ). While the approach of including an operator symbol for each instruction type is possible, in view of the difficulties outlined above the extensions could become unnecessarily complicated.

For our purposes we take a less direct approach based on the fact that the extensions to Ω are required solely so that a morphism $\tau: \Sigma \to \Omega'$ can be established. Clearly then we may choose to identify as the constructors of LtoInstr only those objects that are to be mapped to by τ taking an operator of sort expression from Σ. This option is completely compiler-oriented and as such has some disadvantages over the more general scheme outlined above, not the least of which is the fact that the definition of τ no longer directly contains the definition of the translation algorithm. However this approach is much simpler, especially when we consider the translation of commands below and the requirements of more realistic and complex languages. The basic idea is to include an operator symbol of sort LtoInstr for each type of SL expression and we name them in a way that reflects this relation.

sort LtoInstr
  target: LtoInstr × Loc → Instr
  var-abstr: Loc → LtoInstr
  true-abstr: → LtoInst
  false-abstr: → LtoInst
  not-abstr: LtoInstr → LtoInstr
  and-abstr: LtoInstr × LtoInstr → LtoInstr
  or-abstr: LtoInstr × LtoInstr → LtoInstr.

Given this extension to Ω, we may immediately define at least part of the signature morphism $\tau: \Sigma \to \Omega'$. As mentioned in §5.1 we are assuming that there is some pre-determined relation between identifiers and the locations in the segment based on $\ell 0$. Thus we have $\tau(\text{Ide}) = \text{Loc}$ but we eschew further details. One possibility, assuming the presentation of Ide given in §3.1.1 is $\tau(\text{baseid}) = \ell 0$ and $\tau(\text{next}) = \text{nxt}$.

Sorts:
  $\tau(\text{Ide}) = \text{Loc}$
  $\tau(\text{Expr}) = \text{LtoInstr}$
Operator Symbols:
  $\tau(\text{var}) = \text{var-abstr}$
  $\tau(\text{true}) = \text{true-abstr}$
  $\tau(\text{false}) = \text{false-abstr}$
  $\tau(\text{not}) = \text{not-abstr}$

τ(and) = and-abstr
τ(or) = or-abstr

In itself, this definition of τ is far from enlightening so we immediately give the intended interpretation of the operators of sort LtoInstr by listing the associated equations to be added to TLP. The variable e is of sort LtoInstr, though it is obviously intended to be reminiscent of variables of sort Exp in Σ.

x1. target(var-abstr(ℓ_1),ℓ_2) = move(ℓ_1 ℓ_2)
x2. target(true-abstr,ℓ) = ldt(ℓ)
x3. target(false-abstr ℓ) = ldf(ℓ)
x4. target(not-abstr(e),ℓ) = seq(target(e,ℓ),com(ℓ))
x5. target(and-abstr(e_1,e_2),ℓ) = seq(target(e_1,ℓ),
                                    com(ℓ),
                                    target(e_2,nxt(ℓ)),
                                    com(nxt(ℓ)),
                                    or(nxt(ℓ),ℓ),
                                    com(ℓ))
x6. target(or-abstr(e_1,e_2),ℓ) = seq(target(e_1,ℓ),
                                    target(e_2,nxt(ℓ)),
                                    or(nxt(ℓ),ℓ))

To provide some intuitive insight, consider the translation of **x or y** embodied in τ(or)(τ(var)(τ(x)),τ(var)(τ(y))), assuming target location s0.

τ(or)(τ(var)(τ(x)),τ(var)(τ(y))) =
    or-abstr(var-abstr(τ(x)),var-abstr(τ(y)))
target(or-abstr(var-abstr(τ(x)),var-abstr(τ(y))),s0) =
  seq(target(var-abstr(τ(x)),s0),
    target(var-abstr(τ(y)),nxt(s0)),
    or(nxt(s0),s0))
= seq(move(τ(x),s0),
    move(τ(y),nxt(s0)),
    or(nxt(s0),s0))

Hence the translation of **x or y** consists of instructions to load the value of **x** into the location on top of the stack, load the value of **y** into the location above that and then "or" the two together into the lower location.

The initial top of stack for each complete expression (i.e. an expression occurring directly as part of a command) will be s0, as chosen in the above example, and this information will be provided when compiling a command. For example, an assignment statement i:=e could be treated as follows:

$$\tau(\text{assign})(\tau(i),\tau(e)) = \text{seq}(\text{target}(\tau(e),s0),\ \text{move}(s0),\tau(i))).$$

The compilation of commands is a little less straightforward than this and we shall see below that such a definition of τ(assign) is quite inadequate, though here it does serve the purpose of illustration.

We now turn our attention to the translation of the commands of SL. Clearly, since the only control structures in TL are conditional and unconditional branches to labels, the **if** and **while** commands of SL must be coded using such primitives. Hence the compiler must be supplied with labels with which to construct the instruction sequences for the translation of commands, in much the same way as the compiler is supplied with locations for the translation of expressions. This implies the need for a sort representing [Tag → Instr], but there is a fundamental difference between the supply of tags and the supply of locations: the locations can be **re-used** for "consecutive" expressions, whereas a new and different tag must be used each time. Therefore we need to maintain a record of which tags we have used and provided this is done in a regular manner (t0 first, nxt(t0) next and so on), the record of used tags can be achieved by associating a tag to tag map with each command. Putting the two requirements together, we appear to need a sort representing [Tag → [Instr × Tag]] for τ(Com). We achieve this by adding two sorts, IandT and TtoIandT respectively representing [Instr × Tag] and [Tag →[Instr × Tag]] to Ω (and TLP). Again, as was the case for translation of expression above, we allow the syntax of SL commands to suggest the operator symbols of sort TtoIandT.

```
sort IandT
   <_,_>: Instr × Tag → IandT
   1st: IandT → Instr
   2nd: IandT → Tag
sort  TtoIandT
   supply: TtoIandT × Tag → IandT
   continue-abstr: → TtoIandT
```

seq-abstr: TtoIandT $\times$ TtoIandT $\rightarrow$ TtoIandT
assign-abstr: Loc $\times$ LtoInstr $\rightarrow$ TtoIandT
if-abstr: LtoInstr $\times$ TtoIandT $\times$ TtoIandT $\rightarrow$ TtoIandT
while-abstr: LtoInstr $\times$ TtoIandT $\rightarrow$ TtoIandT

Given the (now completed) extension of Ω to Ω', we may complete the definition of $\tau: \Sigma \rightarrow \Omega'$:

Sorts:
τ(Com) = TtoIandT
τ(Program) = Program
Operator Symbols:
τ(continue) = continue-abstr
τ(seq) = seq-abstr
τ(assign) = assign-abstr
τ(if) = if-abstr
τ(while) = while-abstr
τ(prog)(τ(c)) = prog(1st(supply(τ(c),t0)))

Note that it is important that τ(Program) = Program as a general rule. Intuitively our compiler must produce a program in the target language given a program in the source language, otherwise we could not justify calling it a compiler! On a more technical note, if we did not insist on τ(Program) = Program it would be feasible to extend the target language presentation and define the signature morphism only in terms of the sorts and operators that had been added, with no reference to the actual target language. Clearly such a morphism could not be considered to be embodying a translation from the source to the target language.

The equation to be added to TLP to make TLPX (and give some meaning to the operators above) are as follows: -

x7. 1st(<i,t>) = i
x8. 2nd(<i,t>) = t
x9. supply(continue-abstr,t) = <dummy,t>
x10. supply(seq-abstr(c_1,c_2)t) =
    <seq(1st(supply(c_1,t)),
        1st(supply(c_2,2nd(supply(c_1,t))))),
    2nd(supply(c_2,2nd(supply(c_1,t))))>

```
x11. supply(assign-abstr(ℓ,e),t) = <seq(target(e,s0),move(s0,ℓ)), t>
x12. supply(if-abstr(e,c1,c2),t) =
          <seq(target(e,s0),
               com(s0),
               brt(s0,t),
               1st(supply(c1,nxt(nxt(t)))),
               brt(nxt(t)),
               lab(t),
               1st(supply(c2,2nd(supply(c1,nxt(nxt(t)))))),
               lab(nxt(t))),
           2nd(supply(c2,2nd(supply(c1,nxt(nxt(t))))))>
x13. supply(while-abstr(e,c),t) =
          <seq(lab(t),
               target(e,s0),
               com(s0),
               brt(s0,nxt(t)),
               1st(supply(c,nxt(nxt(t)))),
               br(t),
               lab(nxt(t))),
           2nd(supply(c,nxt(nxt(t))))>
```

By virtue of having defined a signature morphism $\tau: \Sigma \to \Omega'$ we have specified the intended translation algorithm of SL programs into TL programs. For the proof of correctness of this compiler we need to define a theory morphism SLP → TLPX based on the above definition of the signature morphism $\tau: \Sigma \to \Omega'$. The extension of τ to a full theory morphism (which we shall ambiguously but conveniently also denote τ) is clearly guided by the arity and sort of each of the operator symbols representing semantic valuations. For example, $\mathcal{P}_S$: Program → U and $\mathcal{P}_T$: Program → Store together with τ(Program) = Program suggests $\tau(\mathcal{P}_S) = \mathcal{P}_T$ and τ(U) = Store, and so on. The final part of the definition of $\tau: Th_{SLP} \to Th_{TLPX}$ is therefore as follows:

Sorts:

τ(U) = Store

τ(Bool) = Bool

Operator Symbols:

τ(arid) = empty

τ(bind) = set

τ(find) = val
τ(tt) = tt
τ(ff) = ff
$\tau(\neg) = \neg$
$\tau(\wedge) = \wedge$
$\tau(\vee) = \vee$
τ(err) = err
$\tau(\mathcal{P}_S) = \mathcal{P}_T$
$\tau(\mathcal{C})(\tau(c),\tau(\rho))$ = apply($/$(1st(supply(τ(c),t)),
$\mathcal{D}$(1st(supply(τ(c),t))),
null),$\tau(\rho)$)

Note that any tag t will do and that the use of $\mathcal{D}$ is valid only because we are providing the null continuation; otherwise a more complex formulation would be required.

$\tau(\mathcal{E})(\tau(e),\tau(\rho))$ = val(apply($/$(target(τ(e),ℓ),
$\mathcal{D}$(target(τ(e),ℓ)),
null),$\tau(\rho)$),ℓ)

Again note that any location ℓ will serve and that, a least for our simple language SL, $\mathcal{D}$(target(τ(e),ℓ)) will always return the arid environment by virtue of the fact that labels are never used in the coding of expressions.

To show τ to be a theory morphism, we need to show that Th_{TLPX} contains the τ-translation of the equations of SLP. We repeat them here but give no detailed proofs. Equations 1 to 11 are trivially satisfied and equations 12 to 23 (the semantic clauses) cause no difficulties though the proofs quickly become unwiedly, mainly due to the rather verbose notation. Also, structural induction over the syntactic sorts is required occasionally, for example τ(14) (see §3.3.5).

τ(12). $\mathcal{P}$(prog(1st(supply(c,t0))))
= apply($/$(1st(supply(c,t)),
$\mathcal{D}$(1st(supply(c,t))),
null),empty)

τ(13). apply($/$(1st(supply(continue-abstr,t)),
$\mathcal{D}$(1st(supply(continue-abstr,t))),
null),σ)
= σ

```
τ(14). apply( 𝓘(1st(supply(seq-abstr(c₁, c₂),t)),
                 𝓓(1st(supply(seq-abstr(c₁, c₂),t))),
                 null),σ)
       = apply( 𝓘(1st(supply(c₂,t)),
                  𝓓(1st(supply(c₂,t))),
                  null),
         apply( 𝓘(1st(supply(c₁,t)),
                  𝓓(1st(supply(c₁,t))),
                  null),σ))
τ(15). apply( 𝓘(1st(supply(assign-abstr(ℓ,e),t)),
                 𝓓(1st(supply(assign-abstr(ℓ,e),t))),
                 null),σ)
       = set(σ,ℓ,val(apply( 𝓘(target(e,ℓ'),
                               𝓓(target(e,ℓ')),
                               null),σ),ℓ'))
τ(16). apply( 𝓘(1st(supply(if-abstr(e,c₁,c₂),t)),
                 𝓓(1st(supply(if-abstr(e,c₁,c₂),t))),
                 null),σ)
       = if val(apply( 𝓘(target(e,ℓ),
                           𝓓(target(e,ℓ)),
                           null),σ),ℓ)
         then apply( 𝓘(1st(supply(c₁,t)),
                       𝓓(1st(supply(c₁,t))),
                       null),σ)
         else apply( 𝓘(1st(supply(c₂,t))
                       𝓓(1st(supply(c₂,t))),
                       null),σ)
τ(17). apply( 𝓘(1st(supply(while-abstr(e,c) t)),
                 𝓓(1s(supply(while-abstr(e,c),t))),
                 null),σ)
       = if val(apply( 𝓘(target(e,ℓ),
                           𝓓(target(e,ℓ)),
                           null),σ),ℓ)
         then  apply( 𝓘(1st(supply(while-abstr(e,c),t)),
                        𝓓(supply(while-abstr(e,c),t))),
                        null),
               apply( 𝓘(1st(supply(c,t)),
                        𝓓(1st(supply(c,t))),
                        null),σ))
```

else σ

$\tau(18)$. val(apply(/(target(var-abstr(ℓ),ℓ')
arid,null),σ),ℓ')
= val(σ,ℓ)

$\tau(19)$. val(apply(/(target(true-abstr,ℓ),
arid,null),σ),ℓ)
= tt

$\tau(20)$. val(apply(/(target(false-abstr,ℓ),
arid,null),σ),ℓ)
= ff

$\tau(21)$. val(apply(/(target(not-abstr(e),ℓ),
arid,null),σ),ℓ)
= $\neg$val(apply(target(e,ℓ),arid,null),σ),ℓ)

$\tau(22)$. val(apply(/(target(and-abstr(e_1,e_2),ℓ),
arid,null),σ),ℓ)
= val(apply(/(target(e_1,ℓ),arid,null),σ),ℓ)
$\wedge$ val(apply(/(target(e_2,ℓ),arid,null),σ),ℓ)

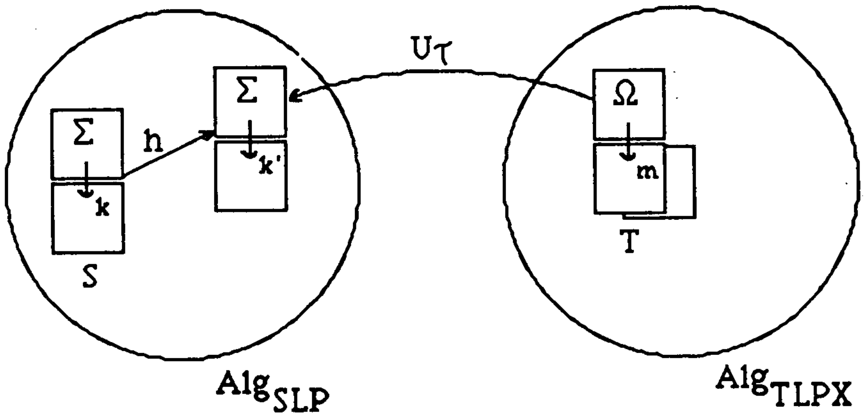
$\tau(23)$. val(apply(/(target(or-abstr(e_1,e_2),ℓ),
arid,null),σ),ℓ)
= val(apply(/(target(e_1,ℓ),arid,null),σ),ℓ)
$\vee$ val(apply(/(target(e_2,ℓ),arid,null),σ),ℓ)

The first expansions of supply(..., t) or target(..., ℓ) are the most enlightening since they replace the uninformative abstraction operators with (possibly incomplete) TL instruction sequences.

Diagrammatically, we may represent the stage we have now reached in the proof as follows:

$$\begin{tikzcd}\boxed{Th_{SLP}} \arrow[r, "\tau"] & \boxed{Th_{TLPX}}\end{tikzcd}$$

To finalise the proof we must fix an SLP-algebra S and a TLPX-algebra T and establish the existence of an SLP-homomorphism h: S $\to$ U_τ(T). This may be done directly (where possible) or by proving the initiality of S where possible. Note that it may be necessary in the case where S is given first, to tailor SLP such that S is initial. In the present case we have fixed the presentation first so we are free to choose any appropriate model, in particular the initial one.



Before any further discussion of particular choices of models we wish to point out two aspects of the example we are treating that have not featured prominently so far. The first is that the compilation homomorphism γ: (algebra of source programs) $\rightarrow$ (algebra of abstracted target programs) is not one to one. This was arranged so that the example was a more comprehensive illustration and simply involves implementing the source language **and** by invoking the identity
$a \wedge b = \neg(\neg a \vee \neg b)$, made necessary by the absence from TL of a corresponding and-instruction. Hence the two source language expressions **a and b** and **not(not(a) or not(b))** are identically coded. The second (and quite separate) point is that the use of the "homomorphism of restriction lemma" (Burstall & Landin, 1969) which is a major feature of earlier related work on compiler correctness has been circumvented in our approach. U_τ "subsumes" the restrictions since the semantic valuations of any TLPX-model (say T) are homomorphic, as must be the semantic valuations of $U_\tau(T)$, simply by virtue of the fact that $U_\tau(T)$ is an SLP-algebra and all the models of SLP have homomorphic semantic valuations (see §3.2).

At this point we intend to leave our example incomplete by not considering particular models and the establishment of an appropriate homomorphism on the grounds that little would be gained through such an exercise, particularly since we have investigated this area fairly thoroughly in Chapter 4. Instead, we shall discuss in more general terms some apparent benefits in the clear separation of the proof into two stages:

establishing a theory morphism and establishing a homomorphism. The underlying intuition is that the translation algorithm proper is expressed via the theory (or signature) morphism while the implementation of the semantic domains of the source language is expressed by means of the homomorphism. Again this idea closely follows a similar notion for semantic congruences, as discussed in §4.1.5.

The factoring of the proof into two stages therefore allows certain implementation details to be treated in isolation and hence they do not interfere with the actual translation algorithm. As a concrete example, a source language whose data types include integers would most likely presume the infinite semantic domain $\mathbb{Z}$. However the target machine may represent integers as strings of 16 bits using the usual 2's complement, that is -32768 .. 32767. Now both are models of the usual presentation (succ, pred, zero, etc) and therefore the theory morphism may be constructed *without* this mismatch in mind. However when attempting to establish a homomorphism from the source semantics to the derived model, the problem is highlighted by the fact that no such homomorphism exists and the domain of the source semantics model must be altered to correspond to the target model. Thus implementation restrictions are identified and may be treated in isolation. More complicated examples of mismatches between the "idealised" source language semantics and target language semantics that may arise in practice include problems with real arithmetic accuracy; limitations on the size of source programs, according to code segment size limitations or symbol table limitations; limitations on the depth of static nesting of blocks as on Burrough's B6700; and the tendency of many Pascal compilers to recognise only the first 8 characters of identifiers. Such difficulties clearly lie within the bounds of what would be called implementation restrictions rather than being major issues in the definition of translation algorithms.

As mentioned in §5.1 we have simplified our example by assuming a direct translation $\tau(\mathrm{Ide}) = \mathrm{Loc}$. It is interesting to consider briefly the requirements necessary for treating a compilation algorithm that takes the more usual approach of maintaining a symbol table. The rudimentary symbol table we have in mind consists of identifier, location pairs with a new pair being added whenever an identifier is met in a defining occurrence in the program. For our purposes we will consider the first (textually) assignment to an identifier to be its defining occurrence and the locations to be allocated in sequence ℓ0, nxt(ℓ0) and so on. Thus we wish to construct a (finite) function: Ide → Loc or its analogue as the compiler scans the program text. Clearly this notion closely corresponds to the environment structure commonly used in semantic definitions.

This compiler information is more complex than that we have so far considered: the location that is currently top of the stack and the next available tag. However it can be treated in much the same fashion by adding sorts and operators to the target language presentation that abstract on the symbol table concept. In outline, the requirements are firstly the addition to TLP of a sort Symtab representing [Ide → Loc] or [Ide × Loc]* with operators similar to those used for environments throughout this dissertation (ie arid, bind and find). Note that sort Ide must also be added to TLP therefore. Secondly, the symbol table is required for translating commands and may itself be altered in the process much as is the next available tag information. Hence we replace TtoIandT (representing Tag → [Instr × Tag] by SandTtoIandTandS, representing [Symtab × Tag] → [Instr × Tag × Symtab]. Similarly, since the symbol table is required for compiling expressions, LtoInstr should be replaced by SandLtoInstr representing [Symtab × Loc] → Instr. The definition of appropriate operators can be achieved in a straightforward manner by taking the same approach as was applied in the detailed extension of TLP to TLPX above.

A less ad hoc approach to the general problem could involve collecting all aspects of the information for the compiler under the umbrella of a single sort called (say) Compinfo. This sort would generally represent tuples; for our case Loc × Tag × Symtab. Given such a sort, the target language presentation could be further extended by a single abstraction sort representing Compinfo → [Instr × Compinfo] with operators based on each of the source language syntactic operators. Further consideration of such a methodology is outside the scope of our investigation but does provide a pointer for future work.

5.4 Compiler Correctness and the Commuting Square

It is pointed out in Thatcher et al (1979) that proving that a Morris square commutes does not necessarily always correspond to proof of compiler correctness. In the usual diagram of Σ-algebras and Σ-homomorphisms (§5.0), if T and U are chosen to be single point algebras, then commutation is ensured simply by the fact that T and U will thus be final Σ-algebras and γ, ψ *and* ε are hence unique (as is θ). Clearly there are many more subtle cases than the completely degenerate one outlined here.

To provide a more concrete example of how such a single point T and U can arise in practice we can define an appropriate "compiler" from SL to TL that always produces the same TL program. If we arrange for *every* SL program to be translated to the *same* TL code,

prog(ldt(f0))

then each SL command can be viewed as being translated into ldt(f0). Such a compiler leads to a Σ-algebra T with carriers defined as follows:

$T_{Program}$ = {prog(ldt(f0))}
T_{Com} = {ldt(f0)}
T_{Exp} = {ldt(s0)} (though there is no significance in such a choice)
T_{Ide} = {f0}

The corresponding semantic Σ-algebra U may be derived from the semantic Ω-algebra U_0 intrinsic in any particular model of TLP by the same means that T is derived from T_0 above, so that $U_{Program}$ = {set(empty,f0,tt)}. Since the carriers of T and U are all singleton sets, T and U are final objects in Alg_Σ implying that ψ and ε are unique and the square commutes, yet we are unlikely to consider such a translation algorithm to be a correct compiler.

Thatcher et al (1979) suggest that requiring ε to be injective is sufficient to avoid such degenerate cases and work their proof within such a framework. However they leave open the question whether the injectivity of ε is a necessary condition. The intuition underlying such a restriction on ε is that it prevents two different source program meanings (in M) from being identified in U. Thus if two programs in S have *different* meanings attached by the homomorphism $\theta: S \rightarrow M$, then requiring ε to be injective effectively prevents those two programs from being compiled into target language programs (in T) that have the *same* meaning. This circumvents the problems of degeneracy outlined above.

On the other hand the original diagram of Morris (1973) has a decode homomorphism δ: U $\rightarrow$ M, though for convenience he deals with ε: M $\rightarrow$ U. This requires the inclusion of a proof that ε has an inverse, at least for the part of U related to runnable programs. Without going into the details of that paper, this is equivalent to showing that ε has an inverse homomorphism δ(i.e. ε is an isomorphism) when restricted to a particular subalgebra of U. Clearly this is equivalent to requiring that ε be injective.

We suggest that using δ: U $\rightarrow$ M is inappropriate and that requiring ε to be injective is an excessive restriction. In the remainder of this section we intend to formulate necessary and sufficient conditions to ensure that a proof that a Morris square commutes constitutes a proof of compiler correctness. As a vehicle for our discussion we intend to use the rather unorthodox, but very simple notion of a completely trivial compiler that translates source programs into themselves. Also, rather than introducing any new languages we return to the lambda calculus as our source (and target) language and consider the semantic models treated in §4.2.2. One reason for choosing a compiler that translates a lambda expression into the same lambda expression is that it allows us to make the application of our work on semantic congruences more obvious. Another reason is that there can be no doubt that such a compiler is *correct* although we can imagine some resistance to the term "compiler" being applied.

To construct our Morris square, we intend to choose the left hand side of the diagram, θ: S $\rightarrow$ M, to be the LC-algebra Op and the right hand side of the diagram, ψ: T $\rightarrow$ U, to be the LC-algebra Den (§4.2.2). For completeness we will give specific definitions of the algebras S, M, T and U. Presume the syntactic signature to be the following portion of LC, denoted Λ.

sorts: Lambda, B, Ide
operators: constant: B $\rightarrow$ Lambda
var: Ide $\rightarrow$ Lambda
abstraction: Ide $\times$ Lambda $\rightarrow$ Lambda
application: Lambda $\times$ Lambda $\rightarrow$ Lambda

Then the semantic Λ-algebra M can be derived from Op as described in §3.2.3 to yield the following definition.

$M_{Lambda} = [E \to U]$
$M_B = B$
$M_{Ide} = Ide$
where $E = [Ide \times U]^*$, $U = [B + CLO]$ and $CLO = [M_{Lambda} \times Ide \times E]$
$constant_M(b) = \lambda e.(b \textit{ in } U)$
$var_M(x) = \lambda e.Lookup(x,e)$
$abstraction_M(x,\eta) = \lambda e.(\langle\eta,x,e\rangle \textit{ in } U)$
$application_M(\alpha,\beta) = \lambda e.apply(\alpha(e),\beta(e))$
where $apply(\langle\eta,x,e\rangle,b) = \eta(Extend(e,x,b))$

Similarly, the semantic Λ-algebra U can be derived from Den.

$U_{Lambda} = [ENV \to V]$
$U_B = B$
$U_{Ide} = Ide$
where $ENV = [Ide \to V]$, $V = [B + FUN]$ and $FUN = [V \to V]$
$constant_U(b) = \lambda\rho.(b \textit{ in } V)$
$var_U(x) = \lambda\rho.\rho(x)$
$abstraction_U(x,\eta) = \lambda\rho.(\lambda a.(\eta(\rho[x/a])) \textit{ in } V)$
$application_U(\alpha,\beta) = \lambda\rho.(\alpha(\rho)|FUN)(\beta(\rho))$

The particular Morris square we are dealing with is the following:

$$\begin{tikzcd} T_\Lambda \arrow[r, "\gamma"] \arrow[d, "\theta"'] & T_\Lambda \arrow[d, "\psi"] \\ M \arrow[r, "\epsilon"'] & U \end{tikzcd}$$

in which M and U are defined above, S and T are both instantiated as an initial Λ-algebra, θ and ψ are the associated unique homomorphisms and γ is the identity homomorphism. The existence of the encoding homomorphism ε is (indirectly) established in §4.2.

Thus we have defined a correct compiler and shown that its corresponding diagram does in fact commute. However ε is *not* injective, nor can a decode homomorphism $\delta: U \to M$ be found. If we consider the two lambda expressions $\lambda x.x$ and $\lambda y.y$, their respective meanings in M, attached by θ, are

$\lambda e.(\langle \lambda e'.\mathrm{Lookup}(x,e'),x,e\rangle$ *in* U)

and $\lambda e.(\langle \lambda e'.\mathrm{Lookup}(y,e'),y,e\rangle$ *in* U)

which are clearly different at least in the second item of the triple. The meanings in U attached by ψ are both

$\lambda\rho.(\lambda a.a$ *in* V).

Hence the Λ-homomorphism ε:M→U takes both the above objects of M to the same object of U and ε is immediately not injective. On the other hand, a decode homomorphism cannot possibly exist since clearly $\delta(\lambda\rho.(\lambda a.a$ *in* V)) must be single valued yet still satisfy $\theta = \gamma.\psi.\delta$.

It would seem that such blanket restrictions as "injective ε" have little to offer as general solutions to the problem we are addressing. We see a parallel here with the notion of an acceptable model of a semantic theory (ok-model) discussed in §3.3. It is noted there that the individual specifying the semantics of a particular language must be the one who decides whether two constructs may or may not be assigned the same meaning value. For instance, there can be no a priori reason for expecting that $\lambda x.x$ and $\lambda y.y$ may be given equivalent interpretations. In the same vein, there can be no justification for insisting in all cases that no more identification (i.e. confusion, §3.3) occurs in the semantics of (compiled) target programs than in the semantics of source programs, yet this is precisely what restricting ε to be injective ensures.

This leads us to suggest the following single requirement for a commuting square to represent compiler correctness: both $\theta: S \to M$ and $\gamma.\psi: S \to U$ should define acceptable semantics for the source language. If this condition holds we consider γ to be an ***acceptable translation***. With respect to our trivial compiler for the lambda calculus, γ is the identity and $\theta: S \to M$ and $\psi: T \to U$ are respectively the operational and denotational semantics of the lambda calculus and they are clearly acceptable semantic models according to our treatment in §3.3.

In terms of the semantic models we have been mainly dealing with in this dissertation, where syntactic domains, semantic domains and semantic valuations are treated as aspects of a single algebra, the notion of acceptable translation can be defined more exactly in terms of previously introduced concepts. Supposing we have source and target language semantic presentations SL and TL, together with an SL-algebra A, a TL-algebra B and a theory morphism ζ: $Th_{SL} \to Th_{TL}$, then establishing the existence of and SL-homomorphism h: $A \to U_{\zeta}(B)$ constitutes our version of proof that the related Morris square commutes. To ensure the acceptability of the compiler we further require $U_{\zeta}(B)$ to be an ok-model of Th_{SL}. Naturally, we are already presuming A and B to be ok-models of their respective theories and hence that appropriate sub-final models have been indicated.

We claim this requirement to be a suitable replacement for the rather excessive restriction that ε be injective on the grounds that the definition of ok-models has the effect of placing an upper limit on the allowable identification of SLP terms in $U_{\zeta}(B)$, irrespective of whether or not they are identified in A.

Chapter 6
Conclusion

In this thesis we have been concerned with developing an algebraically based technique for specifying the semantics of programming languages and examining the technique's utility and influence on formal proofs involving such specifications.

In Chapter 3, we laid the foundations by investigating the use of equational presentations for specifying programming language semantics. We noted that not all the algebras that are models of the presented theory necessarily provide acceptable semantics and suggested that a delineation of the subclass of models that are acceptable can be achieved by designating a particular algebra as being the one where as much identification of terms (i.e. confusion) as is admissible takes place. The ok-models are then those algebras with a particular homomorphic relation to this one such that *no further* identification is ensured.

It has been noted from the very beginning that a methodology and modularisation technique must be developed to control the level of complexity in any enterprise aimed at producing a sound semantic specification of any realistic language. Insisting on a modular approach leads naturally to the notion of keeping a library of standard types and type constructors and the situation may well develop wherein semantic definitions begin to resemble the denotational style, at least in surface appearance. In fact, Ehrich & Lipeck (1983) have already made some progress in this direction, though they only manage to treat domains of *finite* functions. An important aspect is the consideration of the usefulness and flexibility of the idea of *higher-order* algebras. The choices here are to follow the work of Parsaye-Ghomi (1981) or Poigne (1984) or perhaps to stick with standard universal algebra and develop a standard *notation* for sorts representing functional domains. This latter option seems more attractive at present since it would seem advantageous to avoid any insistence on carriers being functional and consequently disallowing closures as operational equivalents.

The relation between the models of our semantic presentations and the initial algebra approach is established on both a conceptual and formal level. An interesting opportunity for further research arises here. Given some operational semantic model, an initial algebra semantics may be systematically derived from it. Now the initial algebra semantics is clearly denotational by virtue of the fact that it is in terms of a homomorphism from syntax to semantics and homomorphisms are by definition compositional. The exact connection between such "pairs" of operational and denotational definitions is not immediately evident, nor is it clear whether such connections can be fruitfully exploited.

In Chapter 4 we examined the notion of the congruence of semantic models from an intuitive standpoint and developed a rigorous algebraic formulation corresponding to the natural idea. Expressed simply, two semantic models are congruent provided there is a homomorphism from one to an algebra derived from the other. Such relationships cannot always be established by the traditional inductive approaches, however the concept of initiality and some straightforward related results allow us to fill this gap. The detailed examples treated in Chapter 4 are sufficient to give us confidence that our formal notion of congruence is both useful and tractable. The fixed-point construction developed when dealing with those examples would appear to have applications for operational semantics beyond the fairly narrow range of uses given here.

The size and level of detail of some of the proofs in Chapter 4 clearly indicates the need for some form of automation or mechanical assistance when developing such proofs. Though it is outside the bounds of this dissertation, there would appear to be some challenging problems involved with the development of a fully general system based on equational rewrite rules (O'Donnell, 1977).

In Chapter 5 we reformulated the advice of Morris (1973) on proving the correctness of compilers to suit our style of semantic definition, applying much of the work we had done on congruences to this related problem. In treating a somewhat more realistic example than those appearing in the literature, with the target languages being described in the same style as the source language, it was made clear that the notion of a compiler being a homomorphism from the language to an algebra derived from the target language was overly simplistic. If the compiler expects to maintain any record of the symbols being used (eg the location that is current top of stack, the next unused target label) or any relation between source and target objects (eg a symbol table), then the compiler appears to correspond more closely to a homomorphism from the source language to *abstractions* of the target language, so that the derived algebra cannot always be generated from the target language alone.

The status of the homomorphism that forms the part of the commuting square diagram connecting the two semantic algebras was investigated in tems of a trivial example. Both the *decode* option, being a homomorphism from target program meanings of source program meanings or requiring the *encode* homomorphism ε from source program meanings to be injective were found to be too restrictive and hence unsatisfactory. An alternative is presented wherein the compiler homomorphism is required to constitute an *acceptable translation*, a notion closely related to that of an ok-model introduced in §3.3. In fact, closer examination leads to more fundamental questions than the injectivity or otherwise of ε, calling into doubt the appropriateness of the whole commuting square approach.

It seems natural that if $\gamma.\psi: S \to U$ (the semantics of S given by the composition of the compiler and target semantics homomorphisms) is an acceptable semantic definition of the source language S, then the translation γ is *correct* as well as being acceptable. For certain choices of models, such a situation can exist without a homomorphism $\varepsilon: M \to U$ necessarily existing. We are actively pursuing this line of investigation at the moment, with obvious influence on a re-development of the concept of an ok-model, both in terms of our semantic definitions and also within the framework of initial algebra semantics.

The future will tell whether algebraic foundations will allow the development of programming methodologies (or even programming languages) emphasising correctness, that are accessible to and useful for programmers at large. What is clear however, is that the formal basis of any such work must be clearly detailed and confirmed if such an enterprise is to succeed.

Bibliography

Arbib, M.A. and Manes, E.G. (1975):
Arrows, Structures and Functors: The Categorical Imperative, Academic Press.

Baker-Finch, C.A. (1984a):
Acceptable Models of Algebraic Semantics, 7th Annual Australian Computer Science Conference. Australian Computer Science Communications 6,1, pp. 5.1-5.10.

Baker-Finch, C.A. (1984b):
Algebraic, Operational and Denotational Semantics of the Lambda Calculus, Australian Computer Journal, 16,3, pp. 96-101.

Birkhoff, G. (1935):
On the Structure of Abstract Algebras, Proceedings, Cambridge Philosophical Society 31, pp 433-454.

Birkhoff, G and Lipson, J.D. (1970):
Heterogeneous Algebras, Journal of Combinatorial Theory 8, pp 115-133.

Bjorner, D. (editor) (1983):
Formal Description of Programming Concepts - II. North-Holland.

Broy, M. and Wirsing, M. (1980):
Programming Languages as Abstract Data Types, 5eme Colloque de Arbres en Algebre et en Programmation, Lille. M. Dauchet (ed.), pp 160-177.

Broy, M., Dosch, W., Moller, B. and Wirsing, M. (1981):
GOTOs - A Study in the Algebraic Specification of Programming Languages, Internal Report CSR-89-81. Department of Computer Science, University of Edinburgh.

Burstall, R.M. and Goguen, J.A. (1979):
The Semantics of Clear, A Specification Language, Abstract Software Specifications. D. Bjorner (ed.), Lecture Notes in Computer Science 86, pp 292-332. Springer-Verlag.

Burstall, R.M. and Goguen, J.A. (1982):
Algebras, Theories and Freeness: An Introduction for Computer Scientists, Theoretical Foundations of Programming Methodology, M. Broy and G. Schmidt (eds.), pp 329-348. D. Reidel.

Burstall, R.M. and Landin, P.J. (1969):
Programs and Their Proofs: An Algebraic Approach, Machine Intelligence 4, pp 17-43. Edinburgh University Press.

Church, A. (1941):
The Calculi of Lambda - Conversion. Princeton University Press.

Cohn, P.M. (1982):
Universal Algebra. D. Reidel.

Dennis, J.B. (1974):
On Storage Management for Advanced Programming Languages, MIT Computation Structures Group Memo 109-1.

Dennis, J.B. (1976):
Semantic Theory for Computer Systems, Course Notes 6.841, Massachussets Institute of Technology.

Ehrich, H-D. and Lipeck, U. (1983):
Algebraic Domain Equations, Theoretical Computer Science 27, pp 167-196. North-Holland.

Elgot, C. C. (1973):
Monadic Computation and Iterative Algebraic Theories, Logic Colloquium '73. H.E. Rose and J.C. Shepherdson (eds), pp 175-230. North-Holland.

Fasel, J.H. (1980):
Programming Languages as Abstract Data Types: Definition and Implementation. Ph.D. Thesis, Purdue University.

Gaudel, M-C., Deschamp, Ph. and Mazaud, M. (1978):
Semantics of Procedures as an Algebraic Abstract Data Type, IRIA Laboria Report No. 334.

Gaudel, M-C. (1980):
Specification of Compilers as Abstract Data Type Representations, Semantics-Directed Compiler Generation. N.D. Jones (ed.), Lecture Notes in Computer Science 94, pp 140-164. Springer-Verlag.

Goguen, J.A. (1975):
Correctness and Equivalence of Data Types, Mathematical Systems Theory. Lecture Notes in Economics and Mathematical Systems 131, pp 352-358. Springer-Verlag.

Goguen, J.A. (1978):
Abstract Errors for Abstract Data Types, Formal Description of Programming Concepts. E.J. Neuhold (ed.). North-Holland.

Goguen, J.A. (1980):
How to Prove Algebraic Inductive Hypotheses Without Induction, 5th Conference on Automated Deduction. W. Bibel and R. Kowalski (eds.), Lecture Notes in Computer Science 87, pp 356-373. Springer-Verlag.

Goguen, J.A. and Burstall, R.M. (1984a):
Some Fundamental Algebraic Tools for the Semantics of Computation. Part 1: Comma Categories, Colimits, Signatures and Theories, Theoretical Computer Science 31, pp 175-209. North Holland.

Goguen, J.A. and Burstall, R.M. (1984b):
Some Fundamental Algebraic Tools for the Semantics of Computation. Part 2: Signed and Abstract Theories, Theoretical Computer Science 31, pp 263-295. North-Holland.

Goguen, J.A. and Meseguer, J. (1981):
Completeness of Many-Sorted Equational Logic, ACM SIGPLAN Notices 16, 7, pp 24-32.

Goguen, J.A. and Meseguer, J. (1983):
An Initiality Primer. (Draft Report).

Goguen, J.A. and Parsaye-Ghomi, K. (1981):
Algebraic Denotational Semantics Using Parameterised Abstract Modules, International Colloquium on Formalisation of Programming Concepts. J. Diaz and I. Ramos (eds.), Lecture Notes in Computer Science 107, pp 292-309. Springer-Verlag.

Goguen, J.A. and Tardo, J.J. (1979):
An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications, Specification of Reliable Software, IEEE, pp 170-189.

Goguen, J.A., Thatcher, J.W. and Wagner, E.G. (1977):
Initial Algebra Semantics and Continuous Algebras, Journal of the ACM 24, 1, pp 68-95.

Goguen, J.A., Thatcher, J.W. and Wagner, E.G. (1978):
An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types, Current Trends in Programming Methodology, Vol. 4, R.T. Yeh (ed.), pp 80-149. Prentice-Hall.

Goguen, J.A., Thatcher, J.W., Wagner, E.G. and Wright, J.B. (1973):
A Junction Between Computer Science and Category Theory, I: Basic Concepts and Examples (Part 1), Technical Report, IBM T.J. Watson Research Centre, Yorktown Heights, New York. Research Report RC 4526.

Goguen, J.A., Thatcher, J.W., Wagner, E.G. and Wright, J.B. (1975):
An Introduction to Categories, Algebraic Theories and Algebras, Technical Report, IBM T.J. Watson Research Centre, Yorktown Heights, New York. Research Report RC 5369.

Goguen, J.A., Thatcher, J.W., Wagner, E.G. and Wright, J.B. (1976):
A Junction Between Computer Science and Category Theory, I: Basic Concepts and Examples (Part 2), Technical Report, IBM T.J. Watson Research Centre, Yorktown Heights, New York. Research Report RC 5908.

Guessarian, I. (1981):
Algebraic Semantics, Lecture Notes in Computer Science 99. Springer-Verlag.

Guessarian, I. (1983):
Survey on Classes of Interpretations and Some of Their Applications, ACM SIGACT News 15, 3, pp 45-71.

Guttag, J.V. (1975):
The Specification and Application to Programming of Abstract Data Types. Ph.D. Thesis, University of Toronto, Technical Report CSRG-59.

Guttag, J.V. and Horning, J.J. (1978):
The Algebraic Specification of Abstract Data Types, Acta Informatica 10, pp 27-52.

Henson, M.C. (1983):
Extending Advice on Structuring Compilers and Proving Them Correct, Proceedings 3rd Conference on Foundations of Software Technology and Theoretical Computer Science. (Bangalore). Also: Technical Report CSM-56, Department of Computer Science, University of Essex.

Henson, M.C. and Turner, R. (1982):
Completion Semantics and Interpreter Generation, 9th Annual Symposium on Principles of Programming Languages, ACM, pp 242-254.

Huet, G. and Oppen, D.C. (1980:
Equations and Rewrite Rules: A Survey, Formal Language Theory - Perspectives and Open Problems, R.V. Book (ed.), pp 349-405.

Kamin, S. (1979):
Rationalizing Many-Sorted Algebraic Theories, Technical Report, IBM T.J. Watson Research Centre, Yorktown Heights, New York. Research Report RC 7574.

Kamin, S. (1980):
Final Data Type Specifications: A New Data Type Specification Method, 7th Annual Symposium on Principles of Programming Languages, ACM, pp 131-138.

Kamin, S. (1983):
Final Data Types and Their Specification, ACM Transactions on Programming Languages and Systems 5, 1, pp 97-123.

Kaphengst, H. and Reichel, H. (1977):
Initial Algebraic Sematnics for Non Context-Free Languages, Fundamentals of Computation Theory. Proceedings 1977. M. Karpinski (ed.), Lecture Notes in Computer Science 56, pp 120-126. Springer-Verlag.

Kutzler, B. and Lichtenberger, F. (1983):
Bibliography on Abstract Data Types, Informatik - Fachberichte 68. Springer-Verlag.

Landin, P.J. (1964):
Mechanical Evaluation of Expressions, Computer Journal, 6, 4, pp 308-320.

Lawvere, F.W. (1963):
Functorial Semantics of Algebraic Theories. Ph.D. Thesis, University of Columbia.

Liskov, B. and Zilles, S.N. (1975):
Specification Techniques for Data Abstractions, IEEE Transactions on Software Engineering SE-1, 1, pp 7-19.

MacLane, S. (1972):
Categories for the Working Mathematician. Springer-Verlag.

MacQueen, D.B. and Sanella, D.T. (1984):
Completeness of Proof Systems for Equational Specifications, Internal Report CSR-160-184. Department of Computer Science, University of Edinburgh.

Manes, E.G. (1976):
Algebraic Theories. Springer-Verlag.

Manna, Z. (1974):
Mathematical Theory of Computation. McGraw-Hill.

McCarthy, J. (1962):
Towards a Mathematical Science of Computation, Information Processing, pp 21-28. North-Holland.

McCarthy, J. and Painter, J. (1967):
Correctness of a Compiler for Arithmetic Expressions, Proceedings of Symposia in Applied Mathematics 19, pp 33-41.

Milne, R.E. (1974):
The Formal Semantics of Computer Languages and Their Implementations. Ph.D. Thesis, Cambridge University.

Milne, R.E. and Strachey, C. (1976):
A Theory of Programming Languages Semantics. Chapman and Hall.

Milner, R. (1977):
Fully Abstract Models of Typed λ-Calculi, Theoretical Computer Science 4, pp 1-22.

Milner, R. and Wehrauch, R. (1972):
Proving Compiler Correctness in a Mechanized Logic, Machine Intelligence 7, pp 51-70. Edinburgh University Press.

Morris, F.L. (1972):
Correctness of Translations of Programming Languages -- An Algebraic Approach, Stanford Artificial Intelligence Project Memo AIM - 174. Computer Science Department Report CS-303, Stanford University.

Morris, F.L. (1973):
Advice on Structuring Compilers and Proving Them Correct, ACM Symposium on Principles of Programming Languages, pp 144-152.

Mosses, P. (1980):
A Constructive Approach to Compiler Correctness, Automata, Languages and Programming. Proceedings 1980. J. de Bakker and J. van Leeuwen (eds.), Lecture Notes in Computer Science 85, pp 449-469. Springer-Verlag.

Manes, E.G. (1976):
Algebraic Theories. Springer-Verlag.

Manna, Z. (1974):
Mathematical Theory of Computation. McGraw-Hill.

McCarthy, J. (1962):
Towards a Mathematical Science of Computation, Information Processing, pp 21-28. North-Holland.

McCarthy, J. and Painter, J. (1967):
Correctness of a Compiler for Arithmetic Expressions, Proceedings of Symposia in Applied Mathematics 19, pp 33-41.

Milne, R.E. (1974):
The Formal Semantics of Computer Languages and Their Implementations. Ph.D. Thesis, Cambridge University.

Milne, R.E. and Strachey, C. (1976):
A Theory of Programming Languages Semantics. Chapman and Hall.

Milner, R. (1977):
Fully Abstract Models of Typed λ-Calculi, Theoretical Computer Science 4, pp 1-22.

Milner, R. and Wehrauch, R. (1972):
Proving Compiler Correctness in a Mechanized Logic, Machine Intelligence 7, pp 51-70. Edinburgh University Press.

Morris, F.L. (1972):
Correctness of Translations of Programming Languages -- An Algebraic Approach, Stanford Artificial Intelligence Project Memo AIM - 174. Computer Science Department Report CS-303, Stanford University.

Morris, F.L. (1973):
Advice on Structuring Compilers and Proving Them Correct, ACM Symposium on Principles of Programming Languages, pp 144-152.

Mosses, P. (1980):
A Constructive Approach to Compiler Correctness, Automata, Languages and Programming. Proceedings 1980. J. de Bakker and J. van Leeuwen (eds.), Lecture Notes in Computer Science 85, pp 449-469. Springer-Verlag.

Mosses, P. (1981):
A Semantic Algebra for Binding Constructs, International Colloquium on Formalization of Programming Concepts, J. Diaz and I. Ramos (eds.), Lecture Notes in Computer Science 107, pp 408-418. Springer-Verlag.

Mosses, P. (1983):
Abstract Semantic Algebras!, Formal Description of Programming Concepts - II, D. Bjorner (ed.), pp 45-70. North-Holland.

O'Donnell, M.J. (1977):
Computing in Systems Described by Equations, Lecture Notes in Computer Science 58. Springer-Verlag.

Padawitz, P. and Wirsing, M. (1984):
Completeness of Many-Sorted Equational Logic Revisited, Bulletin of the European Association for Theoretical Computer Science, 24, pp 88-94.

Pair, C. (1982):
Abstract Data Types and Algebraic Semantics of Programming Languages, Theoretical Computer Science 18, pp 1-31. North-Holland.

Parsaye-Ghomi, K. (1981):
Higher Order Abstract Data Types. Ph.D. Thesis, University of California, Los Angeles.

Poigne, A. (1984):
Higher Order Data Structures - Cartesian Closure Versus λ-Calculus, Symposium of Theoretical Aspects of Computer Science. M. Fontet and K. Melhorn (eds.), Lecture Notes in Computer Science 166, pp 174-185. Springer-Verlag.

Reichel, H. (1980):
Initially - Restricting Algebraic Theories, Mathematical Foundations of Computer Science 1980. P. Dembinski (ed.), Lecture Notes in Computer Science 88, pp 504-514. Springer-Verlag.

Sanella, D. and Wirsing, M. (1983):
A Kernel Language for Algebraic Specification and Implementation, Interal Report CSR-131-83. Department of Computer Science, University of Edinburgh.

Scott, D.S. and Strachey, C. (1971):
Toward a Mathematical Semantics for Computer Languages, PRG-6, Programming Research Group, Oxford University.

Stoy, J.E. (1977):
Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press.

Stoy, J.E. (1981):
The Congruence of Two Programming Language Definitions, Theoretical Computer Science 13, pp 151-174.

Strachey, C. and Wadsworth, C.P. (1974):
Continuations: A Mathematical Semantics for Handling Full Jumps, PRG-11, Programming Research Group, Oxford University.

Tarski, A. (1955):
A Lattice-Theoretical Fixpoint Theorem and its Application, Pacific Journal of Mathematics 5, pp 285-309.

Tennett, R.D. (1977):
A Denotational Definition of the Programming Language Pascal, Technical Report 77-47, Queens University, Ontario.

Thatcher, J.W., Wagner, E.G. and Wright, J.B. (1979):
More On Advice on Structuring Compilers and Proving Them Correct, Automata, Languages and Programming. Proceedings 1979. H.A. Maurer (ed.), Lecture Notes in Computer Science 71, pp 596-615. Springer-Verlag.

Turner, R. (1979):
The Operational and Denotational Semantics of the Lambda Calculus, (unpublished manuscript). University of Essex.

Wagner, E.G., Thatcher, J.W. and Wright, J.B. (1978):
Programming Languages as Mathematical Objects, Mathematical Foundations of Computer Science. J. Winkowski (ed.), Lecture Notes in Computer Science 64, pp 84-101. Springer-Verlag.

Wagner, E.G., Wright, J.B., Goguen, J.A. and Thatcher, J.W. (1976):
Some Fundamentals of Order - Algebraic Semantics, Mathematical Foundations of Computer Science, 1976. A. Mazurkiewicz (ed.), Lecture Notes in Computer Science 45, pp 153-168. Springer-Verlag.

Wand, M. (1979):
Final Algebra Semantics and Data Type Extensions, Journal of Computer and System Sciences 19, 1, pp 27-44.

Wand, M. (1980a):
Different Advice on Structuring Compilers and Proving Them Correct, Technical Report No. 95. Computer Science Department, Indiana University.

Wand, M. (1980b):
First-Order Identities as a Defining Language, Acta Informatica 14, pp 337-357.

Wand, M. (1982):
Deriving Target Code as a Representation of Continuation Semantics, ACM Transactions on Programming Languages and Systems, 4,3, pp 496-517.

Wirsing, M. and Broy, M. (1982):
An Analysis of Semantic Models for Algebraic Specifications, Theoretical Foundations of Programming Methodology. M. Broy and G. Schmidt (eds.), pp 351-412. D. Reidel.

Zilles, S.N. (1979):
An Introduction to Data Algebras, Abstract Software Specifications. D. Bjorner (ed.), Lecture Notes in Computer Science 86, pp 248-272. Springer-Verlag.

b