A STRUCTURED APPROACH TO STATIC SEMANTICS CORRECTNESS

R. BARBUTI
Dipartimento di Informatica, Università di Pisa, 56100 Pisa, Italy

A. MARTELLI
Dipartimento di Scienze dell' Informazione, Università di Torino, 10125 Torino, Italy

Communicated by U. Montanari
Received January 1982
Revised March 1983

Abstract. An approach to the correctness proof of static semantics with respect to the standard semantics of a programming language is presented, where correctness means that the properties of the language described by the static semantics, such as type checking, are consistent with the standard semantics. The standard and static semantics are given in a denotational style in terms of some basic domains and domain constructors, which, together with suitable operations, are used to describe fundamental semantic concepts. The domains have different meaning in the two semantics and the static semantics correctness proof is carried out by devising a set of suitable functions between them. We show that the correctness proof can be greatly simplified by structuring the semantics definitions, and we illustrate that by applying the methodology to a simple imperative language. In the example the derivation of a static checking algorithm from the static semantics is described.

1. Introduction

Static checking can be seen in general as the proof of a program property [8], which usually refers to scope rules or type consistency. For instance, a static type checking algorithm checks that a program will be correct with respect to type consistency rules for every possible execution. Static checking can be considered as a first step towards a program correctness proof; for this reason, recently designed programming languages (CLU [17], Alphard [29], Russell [9], ADA [1], ML [13]) emphasize statically checkable features.

The increasing complexity of static aspects of languages has raised the need of a rigorous definition of static checking. This was done for several languages by translating a program into an expression in a suitable formalism characterizing only those aspects of the language relevant for static checking, while disregarding all others [3, 11, 15, 16]. Static checking is carried out by evaluating that expression.
This approach can be rephrased by saying that static checking is carried out by evaluating the program in non-standard domains [8, 24], or alternatively that the program is given a meaning according to a non-standard (static) semantics of the language. The static semantics can be formally defined by means of the same techniques used to define the standard semantics, such as the well-known denotational technique [12].

A denotational approach has been used, for instance, to give the formal definition of ADA [2, 6]. According to this approach the semantics is defined in two separate parts, the static and the dynamic semantics, both given denotationally. The static semantics defines which syntactically correct programs are correct with respect to statically checkable features; the dynamic semantics is defined only for statically correct programs and describes the run time aspects of the language.

This approach is suitable for compiled languages, such as ADA, since the two parts of the semantics closely correspond to two main components of the compiler, and thus the implementor’s task of deriving these components from the formal definition of the language is made easier. On the other hand this approach excludes the possibility of building an interpreter for the language performing all checks at run time. Furthermore the definition of the semantics in two parts can raise some doubts on the consistency between them, especially when complex features, such as polymorphism or aliasing, are involved.

In this paper we take a different approach, although based on the same denotational technique. We assume the formal definition of the language to be given by a unique standard denotational semantics, where all checks (on types, scope rules, . . .) are performed dynamically. Thus an interpreter could be easily derived from it. Then the static semantics will be defined separately and proved correct with respect to the standard one, where correct means that the static semantics describes correctly a wanted program property. A similar approach to static semantics correctness was taken by Milner [19] who proved the correctness of a static type checking algorithm.

Note that, once the static semantics has been proved correct, we might transform the standard semantics by eliminating all checks which are performed statically. In this way we can have the static and dynamic semantics as in the previously described approach, but with the two parts of the semantics certainly consistent.

In this paper we will express the static semantics denotationally. Thus, to prove its correctness, we might use techniques analogous to those used by other authors to prove the equivalence of various standard and non-standard semantic definitions [10, 18, 23, 27]. However we show that the complexity of the correctness proof can be substantially reduced by giving the denotational semantics in a structured way. The approach is described in Sections 2 and 3. In Section 4 the approach is illustrated with the definition of the standard and static semantics of a simple imperative language with integer and boolean expressions, blocks, input and output, recursive procedures. The example shows that, by suitably structuring the definitions, the correctness proof becomes almost trivial. Furthermore we show how to derive a static checking algorithm from the denotational static semantics.
2. An overview of the approach

Let us assume that one of the possible static meanings of a program may be a value "P-correct". Then we say that the static semantics is correct with respect to a given property \( P \) if, whenever the static meaning of a program is "P-correct", then the standard meaning of that program satisfies property \( P \).

A possible approach to the correctness proof of static semantics is that of finding, if it exists, a function \( h \) such that the following diagram:

\[
\begin{array}{c}
L \\
\downarrow s \\
\downarrow s' \\
M \\
\uparrow h \\
SM
\end{array}
\]

commutes, that is \( h(s(p)) = s'(p) \) for every \( p \in L \). Furthermore, \( h \) must satisfy the above definition of correctness, that is \( h \) must map in "P-correct" only values which satisfy property \( P \).

From an algebraic point of view, \( L, M \), and \( SM \) are many-sorted \( G \)-algebras, where \( G \) is the abstract grammar of the source language, \( s \) and \( s' \) are homomorphisms and \( L \) is the word algebra defined by \( G \). It is well known that \( L \) is the initial \( G \)-algebra, that is there is a unique homomorphism from it to any other \( G \)-algebra. Thus to prove that the diagram commutes it suffices to prove that \( h \) is a homomorphism, that is for every operator \( op \) of the \( G \)-algebras

\[
(\text{See [28] for a similar approach to compiler correctness.})
\]

However in most cases it is impossible to give a function \( h \) which is a homomorphism. Let us consider for instance a program \( p1 \) containing the statement

\[
\text{if } B \text{ then } S1 \text{ else } S2,
\]

where the value of the boolean expression \( B \) is always true, the statement \( S2 \) contains a type error and every other construct of \( p1 \) is type correct; and a program \( p2 \) obtained from \( p1 \) by replacing the above if statement with statement \( S1 \). As pointed out in the introduction, the standard semantics assumes all checks to be done dynamically. Thus a branch which is never executed has no effect on the meaning of a program and the two programs \( p1 \) and \( p2 \) have the same standard meaning. However a static semantics would usually give the meaning "type-correct" to program \( p2 \), but not to program \( p1 \) because of the error in statement \( S2 \). Thus there cannot be a function \( h \) since the same standard value should be mapped into two different static values.
The above algebraic approach can be made less restrictive by assuming that the carriers of algebra $SM$ be partially ordered sets, and by requiring that, for every program $p$,

$$h(s(p)) \sqsubset s'(p)$$  \hspace{1cm} (2.1)

where $\sqsubset$ is the ordering relation over the carriers of $SM$.

This is still a correctness proof of the static semantics, if $h$ maps only standard values satisfying property $P$ into static values less than or equal to "$P$-correct". By referring to the above example, $h$ will map the standard meaning of $p1$ and $p2$ into the value "type-correct". Then $s'(p2)$ will be "type-correct", whereas $s'(p1)$ will be an error value greater than "type-correct".

It easy to prove by induction on $p$ that if every operator $op$ of the $G$-algebras has the following properties:

$$h(op_M(x_1, \ldots, x_n)) \sqsubseteq op_{SM}(h(x_1), \ldots, h(x_n)),$$

(2.2)

$$op_{SM} \text{ is monotone},$$

(2.3)

then (2.1) holds.

We assume now the standard and static semantics to be given in a denotational style, and all carriers of $M$ and $SM$ to be Scott's domains, that is continuous lattices. We use lattices instead of c.p.o.s for a technical reason, to be explained in Section 3.3. This property of carriers obviously satisfies the previous requirement on carriers of $SM$ of being partially ordered sets. Every domain can be either a primitive domain or it can be obtained by applying a domain constructor, such as $+$ (union), $\times$ (cartesian product), $\rightarrow$ (functions), to other domains. These auxiliary domains are introduced to describe basic semantic concepts, such as environment, store or continuations.

This structured definition of the semantic domains allows to simplify the correctness proof for these domains which have the same structure in $M$ and $SM$. More precisely, to prove static semantic correctness we have to find a function $h$ for every pair of corresponding (i.e. with the same name) domains in $M$ and $SM$, and to prove (2.2) and (2.3) for every operation. There are three possible cases:

(i) if corresponding domains are equal in $M$ and $SM$ (e.g. Types, Identifiers, . . .) then $h$ is the identity function;

(ii) if corresponding domains are defined in $M$ and $SM$ as the application of the same constructor to the corresponding domains in $M$ and $SM$ (e.g. the Environment defined as Identifiers $\rightarrow$ Denotations), then function $h$ can be derived in a standard way from the functions $h$ of the component domains. The rules for deriving $h$ for each constructor are described in the next section, where it is also shown that (2.2) and (2.3) hold for the operations associated with the constructors.

(iii) if corresponding domains are defined in two different ways in $M$ and $SM$ (e.g. Denotations), then a suitable function $h$ must be devised satisfying condition (2.2).
A structured approach

Usually, as it will be shown with the example of Section 4, most of the corresponding domains are defined in the same way, and, as pointed out above, the proof has to be actually carried out only for some auxiliary domains which are defined differently in $M$ and $SM$. These auxiliary domains, together with the associated operations, are used to describe quite general basic semantic concepts, and thus a further advantage of this approach is that these domains can be used either to define new constructs of the language or to give the semantics of other languages, based on the same concepts, without requiring any new proof. This fact was pointed out by Mosses in [21], where an approach to compiler correctness is presented, which allows to add completely new features, like non-determinism or concurrency without changing the original semantic equations.

3. Function $h$ for domain constructors

In this section we give the rules for deriving the function $h$ for commonly used domain constructors, namely $+$, $\times$, $\rightarrow$. As pointed out before the carriers of $M$ and $SM$ are Scott’s domains, that is continuous lattices. A (complete) lattice is a partially ordered set, with ordering relation $\leq$, where every subset has a least upper bound. For the definition of continuous lattices see [25], where it is also proved that the application of the above mentioned domain constructors yield continuous lattices as well.

We give the definition of function $h$ for these constructors and we prove property (2.2) for the operators associated with them; property (2.3) is certainly satisfied because these operators are monotonic and continuous.

3.1. Disjoint union

Let $D^1$ and $D^2$ be two domains. The domain $D = D^1 + D^2$ is the domain

$$\{(1, d^1) | d^1 \in D^1\} \cup \{(2, d^2) | d^2 \in D^2\} \cup \{\bot\} \cup \{\top\}$$

with $\bot \leq d | d \in D$, $d \leq \top | d \in D$ and $\langle n_1, d_1 \rangle \leq \langle n_2, d_2 \rangle$ iff $n_1 = n_2$ and $d_1 \leq d_2$.

Let $D^1_M$ and $D^2_M$ be domains of the algebra $M$ and $D^1_SM$ and $D^2_SM$ the corresponding domains in $SM$.

We define function $h_D$ from $D_M = D^1_M + D^2_M$ to $D_SM = D^1_SM + D^2_SM$ as follows:

$$h_D((1, d^1_M)) = (1, h_{D^1}(d^1_M)),$$

$$h_D((2, d^2_M)) = (2, h_{D^2}(d^2_M)),$$

$$h_D(\bot_M) = \bot_SM,$$

$$h_D(\top_M) = \top_SM$$

where $h_{D^1}: D^1_M \rightarrow D^1_SM$ and $h_{D^2}: D^2_M \rightarrow D^2_SM$. 


The operations of disjoint union are

(i) injection: \( d' \) in \( D \), where \( d' \in D', i = 1, 2 \), denotes the corresponding element \((i, d')\) in \( D \);

(ii) Projection: \( d|D^i \), where \( d \in D \), \( i = 1, 2 \), denotes

\[
\begin{align*}
\perp_{D'} & \quad \text{if } d = \perp_{D}, \\
\top_{D'} & \quad \text{if } d = \top_{D}, \\
d' & \quad \text{if } d = (i, d'), \\
\perp_{D'} & \quad \text{if } d = (j, d') \text{ with } j \neq i;
\end{align*}
\]

(iii) \( d \in D' \), where \( d \in D \), denotes the following truth values:

\[
\begin{align*}
\perp_T & \quad \text{if } d = \perp_{D}, \\
\top_T & \quad \text{if } d = \top_{D}, \\
\text{true} & \quad \text{if } d = (i, d'), \\
\text{false} & \quad \text{if } d = (j, d') \text{ with } j \neq i.
\end{align*}
\]

It is easy to see that \( h_D \) and \( h_{D'} \) satisfy (2.2) for each operation if \( h_{D'} \) \((i = 1, 2)\) are strict, that is

\[
h_{D'}(\perp_{D'_M}) = \perp_{D'_M}, \quad (i = 1, 2).
\]

3.2. Cartesian product

The domain \( D = D^1 \times D^2 \) is the domain

\[
\{(d^1, d^2) \mid d^1 \in D^1, d^2 \in D^2\}
\]

with \((d^1, d^2) \equiv (\bar{d}^1, \bar{d}^2)\) iff \( d^1 \equiv \bar{d}^1 \) and \( d^2 \equiv \bar{d}^2 \).

We define function \( h_D \) from \( D_M = D^1_M \times D^2_M \) to \( D_SM = D^1_SM \times D^2_SM \) as follows:

\[
h_D((d^1_M, d^2_M)) = (h_{D^1}(d^1_M), h_{D^2}(d^2_M)).
\]

The operations are

(i) pair construction: \((d^1, d^2)\), where \( d^1 \in D^1, d^2 \in D^2 \),

(ii) selection: \( d|i \), where \( d \in D \), \( i = 1, 2 \).

Functions \( h_D \) and \( h_{D'} \) obviously satisfy (2.2) for the above operations.

3.3. Continuous functions

In order to define the domain \( D^1 \rightarrow D^2 \) we need the following definitions:

A set \( X \) (ordered by \( \equiv \)) is a chain iff for all \( x_1, x_2 \in X \), \( x_1 \equiv x_2 \) or \( x_2 \equiv x_1 \).

A function \( f \) from \( D^1 \) to \( D^2 \) is monotonic iff \( f(d^1) \equiv f(\bar{d}^1) \) whenever \( d^1 \equiv \bar{d}^1 \).

A function \( f \) from \( D^1 \) to \( D^2 \) is continuous iff \( f \) is monotonic and

\[
f(\bigcup X) = \bigcup \{f(x) \mid x \in X\} \text{ for every chain } X \subseteq D.
\]
The domain $F = D^1 \to D^2$ is the domain of the continuous functions from $D^1$ to $D^2$ with
\[ f \succeq g, f \in F, g \in F \iff f(d^1) \sqsubseteq g(d^1) \text{ for all } d^1 \in D^1. \]

We define function $h_F$ from $F_M = D^1_M \to D^2_M$ to $F_{SM} = D^1_{SM} \to D^2_{SM}$ as follows:
\[ (h_F(f_M))(d^2_{SM}) = \bigsqcup \{ d^2_{SM} \mid d^2_{SM} = h_{D^2}(f_M(d^1_M)) \text{ where } h_{D^1}(d^1_M) \sqsubseteq d^1_{SM} \}. \]

(Remind that because all domains we use are continuous lattices, the existence of a least upper bound for every subset is guaranteed.)

We have to prove that this function $h_F$ actually maps values of $F_M$ into values of $F_{SM}$, that is $h_F(f_M)$ is a continuous function.

To prove this it is not sufficient that $h_{D^1}$ and $h_{D^2}$ are continuous but function $h_{D^1}$ must satisfy a further property defined as follows.

**Property 3.1. Backward continuity.** Let $X$ and $Y$ be two domains. A function $f$ from $X$ to $Y$ is backward continuous if, for every chain in $Y$,
\[ y^1 \sqsubseteq y^2 \sqsubseteq \ldots \sqsubseteq y = \bigsqcup_{i=1}^{\infty} y^i, \quad y, y^i \in Y. \]

and for all $x \in X$ such that $f(x) \sqsubseteq y$ we have
\[ x = \bigsqcup_{i=1}^{\infty} x^i, \quad \text{where } x^i \sqsubseteq x^{i+1} \text{ and } f(x^i) \sqsubseteq y^i. \]

The usefulness of this property is made clear in Appendix A, where we prove that, if $h_{D^1}$ and $h_{D^2}$ satisfy the following constraints:
- $h_{D^1}$ and $h_{D^2}$ are continuous,
- $h_{D^1}$ and $h_{D^2}$ are backward continuous,
then $h_F$ maps continuous functions in continuous functions (that is $h_F(f_M)$ is continuous for all $f_M \in F_M$) and $h_F$ is continuous and backward continuous as well.

Finally, we prove that the operations associated to continuous function domains satisfy condition (2.2).

The operations are
(i) **Functional application**: $f(d^1)$, where $f \in F$, $d^1 \in D^1$.

Condition (2.2), in this case, is
\[ h_{D^2}(f_M(d^1_M)) \sqsubseteq (h_F(f_M))(h_{D^1}(d^1_M)), \]
which is directly verified by the definition of $h_F$.

(ii) **Functional composition**: $f' \circ f''$, where $f' \in F'$, $f'' \in F''$, $F': D^1 \to D^3$, $F'': D^3 \to D^2$.

Condition (2.2), in this case, is
\[ (h_F(f'_M \circ f''_M))(d^1_{SM}) \sqsubseteq (h_F(f'_M) \circ h_F(f''_M))(d^1_{SM}) \text{ for all } d^1_{SM} \in D^1_{SM}; \]

by using the definition of $h_F$ it is easy to see that the above inequality is verified.
(iii) **Functional abstraction:** \( \text{LAM } x^1. E(x^1) \), where \( x^1 \) is a variable in \( D^1 \). \( E(x^1) \) is an expression in \( D^2 \) which contains the variable \( x^1 \) and \( \text{LAM } x^1. E(x^1) \) denotes the function which maps any argument in \( D^1 \) to the value in \( D^2 \) obtained by evaluating \( E \) when \( x^1 \) equals the argument value.

Condition (2.2), in this case, is

\[
\text{fix}(F) = \bigcup_{n=1}^{\infty} F^n(\bot).
\]

We prove in Appendix B (Theorem B.1) that if all component operations of \( E \) satisfy condition (2.2) and (2.3), then (3.1) holds.

(iv) **Least fixed point:** for any domain \( D \) we define

\[
\text{fix}: (D \rightarrow D) \rightarrow D \text{ by } \text{fix}(f) = \bigcup_{n=1}^{\infty} f^n(\bot_D).
\]

In Appendix B (Theorem B.2) we prove that condition (2.2) holds for the least fixed point, that is

\[
\text{fix}_D(f_M) = \text{fix}_{SM}(h_F(f_M)) \text{ where } F = D \rightarrow D
\]

if \( h_D \) is strict and continuous.

3.4. Reflexive domains

Scott’s domains can be also defined recursively, e.g.

\[
D^2 = D^1 + D^2 \rightarrow D^2.
\]

In this case the function \( h_D \) will also be defined recursively in terms of the functions \( h \) of the component domains, according to the rules given in the previous section.

It easy to see that the properties of continuity and backward continuity of functions \( h \) are preserved by disjoint union and cartesian product. Therefore we can summarize the results of this section by saying that, if functions \( h \) of basic domains are strict, continuous and backward continuous, then functions \( h \) of the derived domains (obtained by applying Scott’s constructors in any order to basic domains) satisfy condition (2.2) for all operations.

In particular, if in the basic domains in \( M \) and \( SM \) there are no infinite chains of distinct elements, then continuity of \( h \) reduces to monotonicity and backward continuity holds trivially for every \( h \).

4. An example

4.1. The language \( L \)

The grammar \( G \) of the language \( L \) is given in Fig. 1. \( L \) is a strongly typed imperative language with integer and boolean types, and with a block structure a
program \( \langle \text{prog} \rangle := \langle \text{block} \rangle \)

\text{mk-block} \hspace{1cm} \langle \text{block} \rangle := \text{begin} \langle \text{decl} \rangle ; \langle \text{stat} \rangle \text{ end} \)

\text{emptydecl} \hspace{1cm} \langle \text{decl} \rangle := \text{emptydecl} \)

\text{compdecl} \hspace{1cm} \langle \text{decl} \rangle := \langle \text{decl} \rangle ; \langle \text{decl} \rangle \)

\text{vardecl} \hspace{1cm} \langle \text{decl} \rangle := \text{var} \langle \text{ide} \rangle : \langle \text{type} \rangle \)

\text{procdecl} \hspace{1cm} \langle \text{decl} \rangle := \text{procedure} \langle \text{ide} \rangle \langle \langle \text{param} \rangle \rangle \langle \text{block} \rangle \)

\text{mk-param} \hspace{1cm} \langle \text{param} \rangle := \langle \text{ide} \rangle : \langle \text{type} \rangle \)

\text{inttype} \hspace{1cm} \langle \text{type} \rangle := \text{integer} \)

\text{booltype} \hspace{1cm} \langle \text{type} \rangle := \text{boolean} \)

\text{assign} \hspace{1cm} \langle \text{stat} \rangle := \langle \text{ide} \rangle := \langle \text{exp} \rangle \)

\text{ifthenelse} \hspace{1cm} \langle \text{stat} \rangle := \text{if} \langle \text{exp} \rangle \text{ then} \langle \text{stat} \rangle \text{ else} \langle \text{stat} \rangle \)

\text{wiledo} \hspace{1cm} \langle \text{stat} \rangle := \text{while} \langle \text{exp} \rangle \text{ do} \langle \text{stat} \rangle \)

\text{compound} \hspace{1cm} \langle \text{stat} \rangle := \langle \text{stat} \rangle ; \langle \text{stat} \rangle \)

\text{proccall} \hspace{1cm} \langle \text{stat} \rangle := \langle \text{ide} \rangle \langle \langle \text{ide} \rangle \rangle \)

\text{mk-read} \hspace{1cm} \langle \text{stat} \rangle := \text{read} \langle \text{ide} \rangle \)

\text{mk-write} \hspace{1cm} \langle \text{stat} \rangle := \text{write} \langle \text{exp} \rangle \)

\text{emptystat} \hspace{1cm} \langle \text{stat} \rangle := \text{emptystat} \)

\text{mk-plus} \hspace{1cm} \langle \text{exp} \rangle := \langle \text{exp} \rangle + \langle \text{exp} \rangle \)

\text{mk-minus} \hspace{1cm} \langle \text{exp} \rangle := \langle \text{exp} \rangle - \langle \text{exp} \rangle \)

\text{mk-times} \hspace{1cm} \langle \text{exp} \rangle := \langle \text{exp} \rangle \ast \langle \text{exp} \rangle \)

\text{mk-and} \hspace{1cm} \langle \text{exp} \rangle := \langle \text{exp} \rangle \text{ and} \langle \text{exp} \rangle \)

\text{mk-or} \hspace{1cm} \langle \text{exp} \rangle := \langle \text{exp} \rangle \text{ or} \langle \text{exp} \rangle \)

\text{mk-lessthan} \hspace{1cm} \langle \text{exp} \rangle := \langle \text{exp} \rangle \text{ <} \langle \text{exp} \rangle \)

\text{mk-equal} \hspace{1cm} \langle \text{exp} \rangle := \langle \text{exp} \rangle \text{ =} \langle \text{exp} \rangle \)

\text{mk-not} \hspace{1cm} \langle \text{exp} \rangle := \text{not} \langle \text{exp} \rangle \)

\text{mk-true} \hspace{1cm} \langle \text{exp} \rangle := \text{true} \)

\text{mk-false} \hspace{1cm} \langle \text{exp} \rangle := \text{false} \)

\text{mk-constant} \hspace{1cm} \langle \text{exp} \rangle := \langle \text{const} \rangle \)

\text{mk-ident} \hspace{1cm} \langle \text{exp} \rangle := \langle \text{ide} \rangle \)

\text{digit-const} \hspace{1cm} \langle \text{const} \rangle := \langle \text{digit} \rangle \)

\text{comp-const} \hspace{1cm} \langle \text{const} \rangle := \langle \text{const} \rangle \langle \text{digit} \rangle \)

\text{letter-ide} \hspace{1cm} \langle \text{ide} \rangle := \langle \text{letter} \rangle \)

\text{comp-ide} \hspace{1cm} \langle \text{ide} \rangle := \langle \text{ide} \rangle \langle \text{letter} \rangle \)

\text{zero} \hspace{1cm} \langle \text{digit} \rangle := 0 \)

\text{nine} \hspace{1cm} \langle \text{digit} \rangle := 9 \)

\text{a} \hspace{1cm} \langle \text{letter} \rangle := A \)

\text{z} \hspace{1cm} \langle \text{letter} \rangle := Z \)

Fig. 1.
la Pascal, i.e. where blocks are procedure bodies. Procedure parameters are variables passed by reference.

To allow to see grammar $G$ as a many sorted algebra, each production has been given a name. Then the sorts of the $G$-algebra are the non-terminals of the grammar, and the names of the productions are operator symbols. For instance, $\text{procdecl}$ is an operator symbol denoting a function which takes three arguments of sorts $(\text{ide})$, $(\text{param})$, $(\text{block})$ and yields a result of sort $(\text{decl})$, and $\text{emptydecl}$ is a constant symbol of sort $(\text{decl})$.

4.2. The first level of the semantic definitions

According to our approach, the standard and static semantic domains, that is the carriers of algebras $M$ and $SM$, will be defined in terms of auxiliary domains. These domains correspond to usual basic semantic domains, and the carriers of $M$ and $SM$ will be obtained by applying the domain constructors described in Section 3 to them.

As pointed out before, some of the domains are defined in the same way both in $M$ and in $SM$, and the correctness proof has to be carried out only for the remaining domains, which are listed in Fig. 2 together with their operations. Their definition will be given in the next section. In Fig. 3 we give the definitions of the domains which are common to the standard and static semantics. In particular, the domains which are carriers of $M$ or $SM$ have the same name as the corresponding non-terminal of grammar $G$.

The semantics of the language is given in Fig. 4 itself. Here we define all operations of the $G$-algebra, except for the operations which already appear in Fig. 3, in terms of the operations of Fig. 2 and of Fig. 3, of the operations of the domain constructors described in Section 3, and of some auxiliary operations whose definition appears at the end of Fig. 4. By adding one of the two different definitions of the operations of Fig. 2 given in the next section, we obtain alternatively the standard and static semantics of the language.

A few comments on the notation. A $\text{LET}$ construct as been used as syntactic sugar to simplify the understanding of expressions. The form of a $\text{LET}$ expression is

\[ \text{LET } x = E_1 \text{ in } E_2 \]

and its meaning is

\[ (\text{LAM } x. E_2) E_1. \]

For every domain $D$ an $\text{IF} \cdots \text{THEN} \cdots \text{ELSE} \cdots$ function (strict in its first argument) is assumed to be defined

\[ \text{IF} \cdots \text{THEN} \cdots \text{ELSE} \cdots : (\text{Boolean} \times D \times D) \to D. \]
**Intval**

\[
\text{plus} : (\text{Intval} \times \text{Intval}) \to \text{Intval} \\
\text{minus} : (\text{Intval} \times \text{Intval}) \to \text{Intval} \\
\text{times} : (\text{Intval} \times \text{Intval}) \to \text{Intval} \\
\text{const-value} : \text{Const} \to \text{Intval}
\]

**Boolval**

\[
\text{and} : (\text{Boolval} \times \text{Boolval}) \to \text{Boolval} \\
\text{or} : (\text{Boolval} \times \text{Boolval}) \to \text{Boolval} \\
\text{less-than} : (\text{Intval} \times \text{Intval}) \to \text{Boolval} \\
\text{equal} : (\text{Intval} \times \text{Intval}) \to \text{Boolval} \\
\text{not} : \text{Boolval} \to \text{Boolval} \\
\text{true} : \text{Boolval} \\
\text{false} : \text{Boolval}
\]

**Ans** the domain of program outputs

\[
\text{ok} : \text{Ans} \\
\text{static-error} : \text{String} \to \text{Ans} \\
\text{run-error} : \text{String} \to \text{Ans} \\
\text{add-and} : (\text{Intval} \times \text{Ans}) \to \text{Ans} \\
\text{if-then-else} : (\text{Boolval} \times \text{Ans} \times \text{Ans}) \to \text{Ans}
\]

**In** the domain of program inputs

**Intloc**

**Boolloc**

**Store**

\[
\text{init-store} : \text{In} \to \text{Store} \\
\text{empty-input} : \text{Store} \to \text{Boolval} \\
\text{read-input} : \text{Store} \to (\text{Intval} \times \text{Store}) \\
\text{new-loc} : (\text{Store} \times \text{Type}) \to (\text{Loc} \times \text{Store}) \\
\text{write-store} : (\text{Store} \times \text{Loc} \times \text{Value}) \to \text{Store} \\
\text{undef-value} : (\text{Store} \times \text{Loc}) \to \text{Boolval} \\
\text{read-store} : (\text{Store} \times \text{Loc}) \to \text{Value}
\]

where \[
\text{Loc} = \text{Intloc} + \text{Boolloc} \\
\text{Value} = \text{Intval} + \text{Boolval}
\]

---

Fig. 2.
Boolean = the flat domain of truth values
String = the flat domain of strings
Digit = the flat domain of digits
Letter = the flat domain of letters
Const = the flat domain of sequences of digits, whose elements are obtained through the operations
digit-const: Digit → Const
comp-const: (Const × Digit) → Const
Ide = the flat domain of sequences of letters (identifiers), whose elements are obtained through the operations
letter-ide: Letter → Ide
comp-ide: (Ide × Letter) → Ide
Type = the flat domain containing the constants
Inttype : Type
booltype : Type
Param = Ide × Type
Loc = Intloc + Boolloc
Value = Intval + Boolval
Cont = Store → Ans statement continuations
Procden = Proc × Type
Proc = Loc → Cont → Cont
Den = Loc + Procden denotable values
Undefenv = a singleton domain with the constant undefenv
Envvalue = Den + Undefenv
Env = Ide → Envvalue the environment
Dcont = Enc → Cont declaration continuations
Econt = Value → Cont expression continuations
Prog = In → Ans
Block = Enc → Cont → Cont
Decl = Enc → Dcont → Cont
Stat = Enc → Cont → Cont
Exp = Enc → Econt → Cont

Fig. 3.

A CASE construct is used for values belonging to a disjoint union.
If \( d \in D = D^1 + D^2 + \cdots \), the notation

\[
\text{CASE } d \\
D^1 \rightarrow \cdots d \cdots \\
D^2 \rightarrow \cdots d \cdots \\
\vdots \\
\text{ENDCASE}
\]
A structured approach

program (block) =
    LAM in. block init-env init-cont (init-store in)

mk-block (decl, stat) =
    LAM env. LAM cont. decl env (LAM env1. stat env1 cont)

empty-decl =
    LAM env. LAM dcont. dcont env

comp-decl (decl1, decl2) =
    LAM env1. LAM dcont. decl1 env1
    (LAM env2. decl2 env2 dcont)

var-decl (ide, type) =
    LAM env. LAM dcont. LAM store
    LET loc-store = new-loc (store, type)
    IN dcont (add-env (env, ide, loc-store \downarrow 1 in Den))
    loc-store \downarrow 2

proc-decl (ide, param, block) =
    LAM env. LAM dcont.
    LET proc-den = (fix (LAM proc. LAM loc. LAM cont.
        block (add-env (add-env (env, ide, proc, param \downarrow 2) in Den).
        param \downarrow 1, loc in Den)) cont),
        param \downarrow 2)
    IN dcont (add-env (env, ide, proc-den in Den))

mk-param (ide, type) =
    \langle ide, type \rangle

assign (ide, exp) =
    LAM env. LAM cont.
    LET env-value = env ide
    IN CASE env-value
        Den \rightarrow CASE env-value
            Loc \rightarrow exp env (LAM value. LAM store.
            IF (ltypeof env-value) = (vtypeof value) THEN cont (write-store
            (store, env-value, value)) ELSE static-error "assign"
        Proc-den \rightarrow LAM store. static-error "assign"
    ENDCASE
    Undef-env \rightarrow LAM store. static-error "undefined-env"
ENDCASE

Fig. 4.
ifthenelse \( (\text{exp}, \text{stat}_1, \text{stat}_2) = \)
\[ \text{LAM env. LAM cont. exp env (LAM value. LAM store.} \]
\[ \text{CASE value} \]
\[ \text{Boolval} \rightarrow \text{if-then-else (value,} \]
\[ \text{stat}_1 \text{ env cont store,} \]
\[ \text{stat}_2 \text{ env cont store) } \]
\[ \text{Intval} \rightarrow \text{static-error} \text{ "if-expr"} \]
\[ \text{ENDCASE}) \]
\[ \text{whiledo (exp, stat) =} \]
\[ \text{LAM env. LAM cont. fix (LAM cont1. exp env} \]
\[ \text{(LAM value. LAM store.} \]
\[ \text{CASE value} \]
\[ \text{Boolval} \rightarrow \text{if-then-else (value,} \]
\[ \text{stat env cont store,} \]
\[ \text{cont store) } \]
\[ \text{Intval} \rightarrow \text{static-error} \text{ "while-expr"} \]
\[ \text{ENDCASE}) \]
\[ \text{compound (stat}_1, \text{stat}_2) = \]
\[ \text{LAM env. LAM cont. stat}_1 \text{ env (stat}_2 \text{ env cont) } \]
\[ \text{proccall (ide}_1, \text{ide}_2) = \]
\[ \text{LAM env. LAM cont.} \]
\[ \text{LET envvalue}_1 = \text{env ide}_1 \text{ IN} \]
\[ \text{LET envvalue}_2 = \text{env ide}_2 \text{ IN} \]
\[ \text{CASE envvalue}_1 \]
\[ \text{Den} \rightarrow \]
\[ \text{CASE envvalue}_1 \]
\[ \text{Procden} \rightarrow \]
\[ \text{CASE envvalue}_2 \]
\[ \text{Den} \rightarrow \]
\[ \text{CASE envvalue}_2 \]
\[ \text{Loc} \rightarrow \text{IF (ltypeof envvalue}_2) = \]
\[ \text{envvalue}_1 \downarrow \text{2} \]
\[ \text{THEN envvalue}_1 \downarrow \text{1} \]
\[ \text{envvalue}_2 \]
\[ \text{cont} \]
\[ \text{ELSE LAM store.} \]
\[ \text{static-error "call"} \]
\[ \text{Procden} \rightarrow \text{LAM store.} \]
\[ \text{static-error "call"} \]
\[ \text{ENDCASE} \]
\[ \text{Undefenv} \rightarrow \text{LAM store.} \]
\[ \text{Fig 4 (cont.)} \]
A structured approach

\[ \text{static-error} \, \text{"under-env"} \]

\[ \text{Loc} \rightarrow \text{LAM} \, \text{store. static-error} \, \text{"call"} \]

\[ \text{ENDCASE} \]

\[ \text{Undefenv} \rightarrow \text{LAM} \, \text{store. static-error} \, \text{"undef-env"} \]

\[ \text{ENDCASE} \]

\[ \text{mk-read (ide)} = \]
\[ \text{LAM} \, \text{env. LAM cont. LAM store.} \]
\[ \text{LET} \, \text{envvalue} = \text{env ide IN} \]
\[ \text{CASE envvalue} \]
\[ \text{Den} \rightarrow \]
\[ \text{CASE envvalue} \]
\[ \text{Loc} \rightarrow \]
\[ \text{CASE envvalue} \]
\[ \text{Intloc} \rightarrow \text{if-then-else (empty-input store, run-error "input"}, \]
\[ \text{cont (write-store} \]
\[ \text{((read-input store)↓2, envvalue in Loc,} \]
\[ \text{read-input store)↓1 in Value))} \]
\[ \text{Boolloc} \rightarrow \text{static-error "read"} \]

\[ \text{ENDCASE} \]

\[ \text{Procden} \rightarrow \text{static-error "read"} \]

\[ \text{ENDCASE} \]

\[ \text{Undefenv} \rightarrow \text{static-error "undef-env"} \]

\[ \text{ENDCASE} \]

\[ \text{mk-write (exp)} = \]
\[ \text{LAM} \, \text{env. LAM cont. exp env (LAM value. LAM store.} \]
\[ \text{CASE value} \]
\[ \text{Intval} \rightarrow \text{add-ans (value, cont store)} \]
\[ \text{Boolval} \rightarrow \text{static-error "write"} \]

\[ \text{ENDCASE} \]

\[ \text{emptystat} = \]
\[ \text{LAM} \, \text{env. LAM cont. cont} \]

\[ \text{mk-plus (exp1, exp2)} = \]
\[ \text{LAM} \, \text{env. LAM econt. exp1 env (LAM value1.} \]
\[ \text{CASE value1} \]
\[ \text{Intval} \rightarrow \text{exp2 env (LAM value2.} \]
\[ \text{CASE value2} \]
\[ \text{Intval} \rightarrow \text{econt ((plus (value1, value2)) in Value)} \]
\[ \text{Boolval} \rightarrow \text{LAM store. static-error "plus"} \]

\[ \text{ENDCASE} \]

Fig. 4 (cont.)
Boolval → Lam store. static-error “plus”
ENDCASE)

mk-minus (exp1, exp2) = · · ·
mk-times (exp1, exp2) = · · ·
mk-and (exp1, exp2) = · · ·
mk-or (exp1, exp2) = · · ·
mk-lessthan (exp1, exp2) = · · ·
mk-equal (exp1, exp2) = · · ·
mk-not (exp) = · · ·
mk-true = · · ·
mk-false = · · ·
mk-constant (const) =
   Lam env. Lam econt. econt ((const-value const) in Value)
mk-ident (ide) =
   Lam env. Lam econt. Lam store.
   Let envvalue = env ide in
   Case envvalue
   Den →
   Case envvalue
   Loc → if-then-else (undef-value (store, envvalue),
                         run-error “undef-value”,
                         econt (read-store (store, envvalue)))
   Procden → static-error “exp-ide”
ENDCASE
Undefenv → static-error “env-error”
ENDCASE

init-env: Env = Lam ide. undefenv in Envvalue
add-env (env, ide, den): Env =
   Lam ide1. If ide1 = ide THEN (den in Envvalue)
       ELSE env ide1

init-cont: Cont = Lam store. ok

ltypeof (loc): Type =
   Case loc
   Intloc → inttype
   Boolloc → booltype
ENDCASE

Fig. 4 (cont.)
A structured approach

\[
\text{typeof (value)} : \text{Type} = \\
\text{CASE value} \\
\text{Intval} \rightarrow \text{inttype} \\
\text{Boolval} \rightarrow \text{booltype} \\
\text{ENDCASE}
\]

Fig. 4 (cont.)

means

\[
\text{IF } dE^D1 \text{ THEN } \cdots (d|D^1) \cdots \\
\text{ELSE IF } dE^D2 \text{ THEN } \cdots (d|D^2) \cdots \\
\cdots
\]

that is, in every clause of the **CASE** construct labelled by \( D^1 \), the type of variable \( d \) is implicitly translated from \( D \) to \( D^1 \).

Function application associates to the left, that is

\[
a \ b \ c \ d
\]

means

\[
((a(b))(c))(d).
\]

Finally, variables are implicitly typed. In fact a variable denoting a value in a certain domain has the same name as the domain, beginning with a lower case letter and possibly followed by an integer number to distinguish several variables of the same type.

The semantics of the language is given by means of the well-known semantic concepts of environment, store and continuations. Continuations are used in this language only to describe abnormal terminations due to errors, but they could be also used to describe jump statements.

The semantics of a program is a function from input to answers. As we will describe precisely in the next section, the input is a sequence of integers and is part of the store, and the answer is a sequence of integers ending with a string which can be either “ok”, if the program terminates correctly, or an error message. Errors are divided into two categories, static and run-time errors. Static errors, of course, are the errors which can be detected by the static semantics, and, in this language, they deal with the environment (when an undefined identifier is used) and with the types. Run-time errors instead deal with store operations, such as reading an empty input or accessing an undefined location.

Two different ifthenelse operations are used in the semantic definition. The first one, **IF \cdots THEN \cdots ELSE \cdots**, has the first argument belonging to the domain **Boolean** and it has the standard meaning. The second operation if-then-else, will be defined in the next section, and its first argument belongs to the domain **Boolval**, which will also be defined in the next section. In the standard semantics the meanings of the two operations will be the same, whereas they will differ in the static semantics.
Procedure denotations are pairs consisting of a procedure value and the type of the argument, which is used in the type checking of procedure calls. Errors occurring in a procedure body are not detected when the procedure is declared, but only when it is called. That is, a wrong procedure which is never used does not give rise to any error. In fact, a procedure value is always a function, which, in case of an error occurring in the procedure body, returns as answer the corresponding error message.

4.3. The last level of semantic definitions

In this section we define the domains and operations of Fig. 2 for the standard and static semantics, and we prove the correctness of static semantics. In Fig. 5 we give the definitions for algebra $M$, that is for the standard semantics. As we mentioned before, the answers are finite sequences of integers, built by the $write$ statement, ending with either "ok" or an error message. The input is a sequence of integers and it is a component of the store. The other component of the store is the memory which is divided into two typed memories, one for integers and one for booleans. Each memory is a mapping from locations (integers) to values, and it contains also a pointer to the last created location.

In Fig. 6 we give the definition for the static semantics. All the domains are singletons except for the answers which are sets of static error messages. Note that, the domains defined as singleton, also contain a top element $T$, and then they are actually two elements domains. The operations defined on such domains are strict on the top.

We can now prove the correctness of static semantics by defining the functions $h$ from the domains of $M$ to the domains of $SM$. Function $h_{Ans}$ is defined as follows:

$$h_{Ans}(T_M) = T_{SM},$$

$$h_{Ans}(\bot_M) = \emptyset,$$

$$h_{Ans}(ok_M) = \{\text{"ok"}\},$$

$$h_{Ans}(\text{static-error}_M(\text{string})) = \{\text{string}\},$$

$$h_{Ans}(\text{run-error}_M(\text{string})) = \{\text{"ok"}\},$$

$$h_{Ans}(\text{add-ans}_M(\text{intval}_M, \text{ans}_M)) = h_{Ans}(\text{ans}_M).$$

This function obviously satisfies all requirements on continuity of Section 3, because neither $Ans_M$ nor $Ans_{SM}$ possess infinite chains.

All other functions $h$ are defined trivially since they map every (non-top) element of a domain in $M$ into the (non-top) element of the corresponding domain in $SM$, and $T_M$ into $T_{SM}$.

Finally we have to prove that condition (2.2) holds for all operations listed in Fig. 2, as defined in Figs. 5 and 6. The proof is trivial for all operations of $Intval$, $Boolval$ and $Store$, and for operations $ok$, $static-error$, $run-error$, $add-ans$ of $Ans$. 
A structured approach

\[ \text{Intval} = \text{Integer with the operations: plus, minus, times, const-value (which converts a constant into an integer)} \]

\[ \text{Boolval} = \text{Boolean with the operations: and, or, less-than, equal, not, true, false.} \]

\[ \text{Ans} = \text{the flat domain of elements obtained through the constructors: ok, static-error, run-error, add-ans.} \]

Furthermore the following operation is defined

\[ \text{if-then-else} \quad (\text{boolval}, \text{ans1}, \text{ans2}) = \]

\[ \text{IF boolval THEN ans1 ELSE ans2} \]

\[ \text{In} = \text{the flat domain of elements obtained through the constructors} \]

\[ \text{empty-in : In} \]

\[ \text{add-in : (Intval} \times \text{In} \rightarrow \text{In} \]

and with operations

\[ \text{head-in : In} \rightarrow \text{Intval} \]

\[ \text{tail-in : In} \rightarrow \text{In} \]

\[ \text{is-empty-in : In} \rightarrow \text{Boolean} \]

\[ \text{Intloc} = \text{Integer} \]

\[ \text{Undefint} = \text{a singleton domain with the constant} \]

\[ \text{undef-int: Undefint} \]

\[ \text{Intmemval} = \text{Intval} + \text{Undefint} \]

\[ \text{Intmem} = (\text{Intloc} \rightarrow \text{Intmemval}) \times \text{Intloc} \]

with the operations

\[ \text{init-int-mem} : \text{Intmem} = (\text{LAM intloc. undef-int in Intmemval, 0)} \]

\[ \text{new-loc-int-mem} (\text{intmem}) : \text{Intmem} = (\text{intmem} \downarrow 1, \text{intmem} \downarrow 2 + 1) \]

\[ \text{write-int-mem} (\text{intmem}, \text{intloc}, \text{intval}) : \text{Intmem} = (\text{LAM intloc1. if intloc1 = intloc \then intval in Intmemval \else intmem \downarrow 1 intloc1, intmem \downarrow 2}) \]

\[ \text{undef-value-int-mem} (\text{intmem}, \text{intloc}) : \text{Boolval} = \]

\[ \text{CASE intmem} \downarrow 1 \text{ intloc} \]

\[ \text{Intval} \rightarrow \text{false} \]

\[ \text{Undefint} \rightarrow \text{true} \]

\[ \text{ENDCASE} \]

Fig. 5
read-int-mem \((\text{intmem}, \text{intloc})\): \(\text{Intval} = (\text{intmem}\downarrow1 \text{intloc})\downarrow\text{Intval}\)

\[
\begin{align*}
\text{Boolloc} & = \text{Integer} \\
\text{Undefbool} & = \text{a singleton domain with the constant} \\
& \quad \text{undef-boot} : \text{Undefbool} \\
\text{Boolmemval} & = \text{Boolval} + \text{Undefbool} \\
\text{Boolmem} & = (\text{Boolloc} \rightarrow \text{Boolmemval}) \times \text{Boolloc} \\
& \text{with operations analogous to those of } \text{Intmem} \\
\text{Mem} & = \text{Intmem} \times \text{Boolmem} \\
& \text{with the operations}
\end{align*}
\]

\[
\begin{align*}
\text{init-mem}: \text{Mem} = \\
& \langle \text{init-int-mem}, \text{init-bool-mem} \rangle \\
\text{new-loc-mem} \ (\text{mem}, \text{type}) : (\text{Loc} \times \text{Mem}) = \\
& \text{if } \text{type} = \text{inttype} \\
& \text{then let } \text{intmem} = \text{new-loc-int-mem} \ \text{mem}\downarrow1 \\
& \quad \text{in } \langle \text{intmem}\downarrow2 \ \text{in Loc}, \\
& \quad \langle \text{intmem}, \ \text{mem}\downarrow2 \rangle \rangle \\
& \text{else let } \text{boolmem} = \text{new-loc-bool-mem} \ \text{mem}\downarrow2 \\
& \quad \text{in } \langle \text{boolmem}\downarrow2 \ \text{in Loc}, \\
& \quad \langle \text{mem}\downarrow1, \ \text{boolmem} \rangle \rangle
\end{align*}
\]

\[
\begin{align*}
\text{write-mem} \ (\text{mem}, \text{loc}, \text{value}) : \text{Mem} = \\
& \text{case } \text{loc} \\
& \quad \text{Intloc} \rightarrow \langle \text{write-int-mem} \ (\text{mem}\downarrow1, \ \text{loc}, \\
& \quad \text{value} \downarrow\text{Intval}), \\
& \quad \text{mem}\downarrow2 \rangle \rangle \\
& \quad \text{Boolloc} \rightarrow \langle \text{mem}\downarrow1, \\
& \quad \text{write-bool-mem} \ (\text{mem}\downarrow2, \ \text{loc}, \\
& \quad \text{value} \downarrow\text{Boolval}) \rangle \rangle \\
& \text{endcase}
\end{align*}
\]

\[
\begin{align*}
\text{undef-value-mem} \ (\text{mem}, \text{loc}) : \text{Boolval} = \\
& \text{case } \text{loc} \\
& \quad \text{Intloc} \rightarrow \text{undef-value-int-mem}(\text{mem}\downarrow1, \ \text{loc}) \\
& \quad \text{Boolloc} \rightarrow \text{undef-value-bool-mem}(\text{mem}\downarrow2, \ \text{loc}) \\
& \text{endcase}
\end{align*}
\]

\[
\begin{align*}
\text{read-mem} \ (\text{mem}, \text{loc}) : \text{Value} = \\
& \text{case } \text{loc} \\
& \quad \text{fig. 5 (cont.)}
\end{align*}
\]
Intval \rightarrow \text{read-int-mem}(\text{mem} \downarrow 1, \text{loc}) \in \text{Value}

Boolval \rightarrow \text{read-bool-val}(\text{mem} \downarrow 2, \text{loc}) \in \text{Value}

\text{ENDCASE}

\text{Store} = \text{Mem} \times \text{In}

\text{with the operations}

\text{init-store} (\text{in}) = (\text{init-mem}, \text{in})

\text{empty-input} (\text{store}) = \n\
\text{is-empty-in} \; \text{store} \downarrow 2

\text{read-input} (\text{store}) = \langle \text{head-in} \; \text{store} \downarrow 2, \langle \text{store} \downarrow 1, \text{tail-in} \; \text{store} \downarrow 2 \rangle \rangle

\text{new-loc} (\text{store}, \text{type}) = \n\
\text{LET} \; \text{locmem} = \text{new-loc-mem} (\text{store} \downarrow 1, \text{type}) \n\
\text{IN} \langle \text{locmem} \downarrow 1, \langle \text{locmem} \downarrow 2, \text{store} \downarrow 2 \rangle \rangle

\text{write-store} (\text{store}, \text{loc}, \text{value}) = \langle \text{write-mem} (\text{store} \downarrow 1, \text{loc}, \text{value}), \text{store} \downarrow 2 \rangle

\text{undef-value} (\text{store}, \text{loc}) = \langle \text{undef-value-mem} (\text{store} \downarrow 1, \text{loc}) \rangle

\text{read-store} (\text{store}, \text{loc}) = \langle \text{read-mem} (\text{store} \downarrow 1, \text{loc}) \rangle

\text{Fig. 5 (cont.)}

For \text{if-then-else} we have

\[ h_{\text{Ans}}(\text{if-then-else}_{\text{SM}}(\text{boolval}_{\text{M}}, \text{ans}_1, \text{ans}_2)) = h_{\text{Ans}}(\text{if} \; \text{boolval}_{\text{M}} \; \text{then} \; \text{ans}_1 \; \text{else} \; \text{ans}_2). \]

\[ \text{if-then-else}_{\text{SM}}(h_{\text{Boolval}}(\text{boolval}_{\text{M}}), h_{\text{Ans}}(\text{ans}_1), h_{\text{Ans}}(\text{ans}_2)) = h_{\text{Ans}}(\text{ans}_1) \cup h_{\text{Ans}}(\text{ans}_2). \]

This is the only operation for which (2.2) does not hold with the equal sign.

Having defined functions \( h \) from the domains of the standard semantics to the corresponding domains of the static semantics, we can now define precisely the properties which are described by the static semantics. In fact we have proved that, for every program \( p \),

\[ h(s(p)) \subseteq s'(p) \]

where \( s \) and \( s' \) are the standard and static semantics. The standard meaning \( s(p) \) is a function in the domain \( \text{In}_M \rightarrow \text{Ans}_M \). According to the definition of function \( h \) for the domain of functions given in Section 3.3, \( h(s(p)) \) is a function in \( \text{In}_{\text{SM}} \rightarrow \text{Ans}_{\text{SM}} \) mapping the single value of \( \text{In}_{\text{SM}} \) into the set of all static error messages which can be returned by \( s(p) \) for all possible input values. According to the above inequality, the static semantics \( s'(p) \) may give a larger set of error messages. Thus if the static
\textit{Intval} = a singleton domain consisting of the element \textit{single-int}, and with the operations
\begin{align*}
\text{plus} \ (\text{intval1, intval2}) &= \text{single-int} \\
\end{align*}

\textit{Boolval} = a singleton domain consisting of the element \textit{single-bool}, and with operations
\begin{align*}
\text{and} \ (\text{boolval1, boolval2}) &= \text{single-bool} \\
\end{align*}

\textit{Ans} = the domain of finite sets of strings corresponding to all possible static errors plus a string "ok" ordered with set inclusion, with the operations
\begin{align*}
\text{ok} &= \{\text{"ok"}\} \\
\text{static-error} \ (\text{string}) &= \{\text{string}\} \\
\text{run-error} \ (\text{string}) &= \{\text{"ok"}\} \\
\text{add-ans} \ (\text{intval, ans}) &= \text{ans} \\
\text{if-then-else} \ (\text{boolval, ans1, ans2}) &= \text{ans1} \cup \text{ans2} \\
\end{align*}

\textit{In} = a singleton domain consisting of the element \textit{single-in}

\textit{Intloc} = a singleton domain consisting of the element \textit{single-intloc}

\textit{Boolloc} = a singleton domain consisting of the element \textit{single-boolloc}

\textit{Store} = a singleton domain consisting of the element \textit{single-store}, and with the operations
\begin{align*}
\text{init-store} \ (\text{in}) &= \text{single-store} \\
\text{empty-input} \ (\text{store}) &= \text{single-bool} \\
\text{read-input} \ (\text{store}) &= \langle\text{single-int, single-store}\rangle \\
\text{new-loc} \ (\text{store, type}) &= \\
\text{IF} \ \text{type} = \text{inttype} \\
\text{THEN} \ \langle\text{single-intloc in Loc, single-store}\rangle \\
\text{ELSE} \ \langle\text{single-boolloc in Loc, single-store}\rangle \\
\text{write-store} \ (\text{store, loc, value}) &= \text{single-store} \\
\text{undef-value} \ (\text{store, loc}) &= \text{single-bool} \\
\text{read-store} \ (\text{store, loc}) &= \\
\text{CASE} \ \text{loc} \\
\text{Intloc} \rightarrow \text{single-int in Value} \\
\text{Boolloc} \rightarrow \text{single-bool in Value} \\
\text{ENDCASE} \\
\end{align*}

Fig. 6.
A structured approach

4.4. Simplifying the static semantics

By putting together Figs. 4 and 6 we obtain the complete definition of the static semantics. However this definition is rather redundant since it contains several singleton domains which do not carry any information. For instance the semantics of a program could be simply \( \text{Ans}_{\text{SM}} \) instead of \( \text{In}_{\text{SM}} \rightarrow \text{Ans}_{\text{SM}} \), and similarly a continuation could be simply an answer. In this way, the domain \( \text{Input} \) and \( \text{Store} \) with their operations would disappear from the static semantics as usually happens with the static semantics of languages.

To obtain the simplified static semantics \( \text{SSM} \) we can use the following general transformation. Let

\[
F_{\text{SM}} = A \rightarrow B
\]

be a domain of \( \text{SM} \) where \( A \) is a singleton domain with the value \( \text{sing-a} \), and let

\[
F_{\text{SSM}} = B
\]

be the corresponding domain of \( \text{SSM} \).

The operations of \( F_{\text{SSM}} \) corresponding to the usual operations of functions of \( F_{\text{SM}} \) are defined as follows:

- functional application: \( f_{\text{SSM}}(a) = f_{\text{SSM}} \)
- functional abstraction: \( \text{LAM } a. E(a) = E(\text{sing-a}) \)

It is easy to see that the domains \( F_{\text{SM}} \) and \( F_{\text{SSM}} \) with their operations are isomorphic.

The simplified static semantics \( \text{SSM} \) is obtained from \( \text{SM} \) by modifying the definition of \( \text{Prog} \) and \( \text{Cont} \) and their operations according to above transformations. For instance, operation \( \text{mk-read} \), defined in Fig. 4, becomes, by substituting \( \text{if-then-else} \) with its definition,

\[
\text{mk-read}(\text{id}) = \\
\text{LAM } \text{env}. \text{LAM } \text{cont}. \\
\text{LET } \text{envvalue} = \text{env } \text{id} \text{ IN } \\
\text{CASE } \text{envvalue} \\
\text{Den } \rightarrow \\
\text{CASE } \text{envvalue} \\
\text{Loc } \rightarrow \\
\text{CASE } \text{envvalue} \\
\text{Intloc } \rightarrow \{\text{"ok"} \} \cup \text{cont} \\
\text{Boolloc } \rightarrow \{\text{"read"} \} \\
\text{ENDCASE} \\
\text{Procden } \rightarrow \{\text{"read"} \} \\
\text{ENDCASE} \\
\text{Undefenv } \rightarrow \{\text{"undef-env"} \} \\
\text{ENDCASE}
\]
Note that, by defining in Section 4.2 the domains Prog and Cont as primitive
domains with suitable operations, we might have defined directly the static semantics
SSM in Section 4.3. However, by introducing SM as an intermediate step, we have
a simpler correctness proof because the definition and correctness of function h for
Prog and Cont comes automatically from the general rules of Section 3 and from
the isomorphism of SM and SSM.

4.5. Static checking algorithm

Given the static denotational semantics of language L, we want now to give an
algorithm for evaluating the static meaning of a program, that is for performing
static checking. The problem of finding this algorithm is analogous to the problem
of deriving an interpreter, or a compiler, from the standard denotational semantics
of a language. Both problems are trivially solved if the denotational semantics is
expressed in a executable formalism.

Our example has been implemented in sis [20], a system designed by Mosses for
defining the denotational semantics of programming languages. The definition of a
language in sis is given in two parts. The first part deals with syntactic aspects and
provides an interface between concrete and abstract syntax. In the second part the
semantics is associated with every construct of the abstract syntax by means of a
language called DSL. DSL is a typed applicative language whose primitive types are
Scott's domains, and which allows to define new types by means of the domain
constructors of Section 3. Thus the standard and static semantics can be trivially
translated from the formalism used in this paper to DSL syntax.

Furthermore, DSL allows to give the semantics in levels, by leaving some domains
with their operations unspecified and by defining them at the lower level. Using
this feature we can give to the semantic definition exactly the same structure as in
this paper, by defining in the first level the part of the semantics given in Fig. 4,
and by defining alternatively in the second level the domains and operations of
Figs. 5 and 6.

Once the semantics of language L has been expressed in DSL we have immediately
an interpreter of the language, which, in the case of static semantics, is the static
checker. However there is a point which must be considered with care, the evaluation
of the fixed point operator. In fact, using the standard evaluation rule would cause
the non termination of the static checker, and thus a different rule must be used.

The domains of static semantics on which the fixed point is computed are finite.
Thus the computation of the fixed point according to its definition, i.e. as the limit
of the chain \( f^n (\bot) \) always terminates in a finite number of steps. A more careful
analysis of these domains allows to find the maximum number of steps.

Let us consider first the fixed point operator appearing in the definition of whiledo.
The fixed point operator is applied to a function from Cont to Cont, which in algebra
SSM is simply a function from \( \text{Ans} \) to \( \text{Ans} \). It is easy to see that each function
\( f : \text{Ans} \rightarrow \text{Ans} \) in algebra SSM is such that

\[
f(\text{ans}) = c \quad \text{or} \quad f(\text{ans}) = c \cup \text{ans}
\]
where $c$ is an element of $\text{Ans}$, because the only operations defined on $\text{Ans}$ are some constants and the union operation. Then, $f(\Phi) = f(f(\Phi))$, and the least fixed point is obtained in just one step. The definition of \texttt{whiledo} in algebra $\text{SSM}$ can be rewritten as

\[
\texttt{whiledo (exp, stat)} = \text{LAM } \text{env. LAM } \text{ans. exp env (LAM value. CASE value \texttt{Boolval}} \rightarrow (\text{stat env } \Phi) \cup \text{ans} \texttt{Intval } \rightarrow \{\text{"while-expr"\}} \text{ENDCASE)}
\]

This definition corresponds to the way the checking of this statement is usually carried out. In fact, the condition $\text{exp}$ is first evaluated and, if it has the correct type, the body $\text{stat}$ is evaluated once with continuation $\Phi$. If the body is correct we have $(\text{stat env } \Phi) = \Phi$ and the whole while statement is correct.

The other fixpoint operator appears in procedure declaration, and it is applied to a function from $\text{Proc}$ to $\text{Proc}$, where $\text{Proc} = \text{Loc} \rightarrow \text{Ans} \rightarrow \text{Ans}$ in algebra $\text{SSM}$. It is easy to see that in this case one step is not sufficient to reach the least fixed point starting from the bottom $\text{LAM } \text{loc. LAM } \text{ans. } \Phi$. In fact, if the procedure body contains a recursive call, the first step does not propagate the continuation of the recursive call; that is, an error occurring after the recursive call is not detected in the first step. However can be seen that two steps are sufficient to reach the least fixed point. In fact, because functions on answers return either $c$ or $c \cup \text{ans}$ as pointed out before, then at the second step all possible answers are propagated.

The checking of procedure declarations is slightly more complicated than expected by requiring two applications of the functional. A more intuitive approach would be to prove inductively the correctness of the procedure body by assuming correct all recursive calls, that is to apply the functional once to the procedure $\text{LAM } \text{loc. LAM } \text{ans. ans. ans}$ (greater than the bottom $\text{LAM } \text{loc. LAM } \text{ans. } \Phi$). However this approach not always yields the least fixed point; in fact if the body of a recursive procedure does not contain any conditional statement, then the least fixed point will be $\text{LAM } \text{loc. LAM } \text{ans. } \Phi$ (reachable starting from the bottom), which is smaller than the obtained fixed point $\text{LAM } \text{loc. LAM } \text{ans. ans.}$ Both these fixed points are semantics of correct procedures. The advantage of the least fixed point approach is to allow the distinction between correct and terminating procedures ($\text{LAM } \text{loc. LAM } \text{ans. ans})$ and correct and non-terminating ones ($\text{LAM } \text{loc. LAM } \text{ans. } \Phi$).

Besides $\text{DSL}$, other languages have been used in the literature to express denotational semantics. For instance, in $\text{OBJ}$ [14], an executable algebraic specification language, the semantics of a language can be described by means of a set of modules, such that each module corresponds to some important features of that language. This approach has the advantage, with respect to $\text{DSL}$, of presenting the semantics in a more structured way. On the other hand, $\text{OBJ}$, and similar algebraic specification languages, do not have higher order capabilities, that is do not allow to treat
operations as values. Of course this gives rise to problems in the description of higher order features of languages, such as procedures or continuations.

A higher order algebraic approach is described in [22] where the semantics of languages is given in terms of algebras of operations (actions).

5. Conclusions

In this paper we have presented an approach to proving static semantics correctness, that is proving that the static semantics correctly describes the desired properties of a given language. The approach is based on a structured description of the semantics, which is widely recognised to be the right technique to present the semantics of a language (see for instance [7]). More specific advantages of this approach are the following:

- The correctness proof is greatly simplified. As shown in the example, it is possible to describe the static and standard semantics in such a way that they are identical up to a certain level of refinement. Then the correctness has to be carried out only for the definitions given in lower levels, and it will propagate automatically through the upper level.

- In many cases it is easy to extend or modify the language. The domains defined at lower levels and their use describe basic semantic concepts, and therefore it is very easy to add new construct to the language, or to define new languages with other constructs, as long as these constructs are based on the same semantic concepts. For instance, it would be possible to add to the language of Section 4 procedure parameters passed by value, or functions, or some kind of jump statement. Note that this extensions would not require any proof at all, since their definitions would be inserted into the first level of the semantic definitions.

The reader might find unsatisfactory the way the static semantics of procedures is defined in our example, because the static checking algorithm has to re-evaluate the procedure body for every call. However it wouldn't be too difficult to define another static semantics, allowing a more classical static checking, and to prove its correctness with respect to the previous one. For instance, in [4] we have used this approach to give the static semantics, and prove its correctness, for a language where procedures can be passed as parameters. First we have defined a static semantics $s': L \rightarrow SM$ similar to the one given in this paper. Then we have defined another static semantics $s'': L \rightarrow SSM$ where the meaning of a procedure is simply its type. The function $h'$ from $SM$ to $SSM$ maps a procedure denotation into its type if, whenever it is applied to a statically correct argument it gives a statically correct answer, or into an error value if this does not happen.

We conclude with a few remarks on the metalanguage used to give the semantics. First the metalanguage should allow to express in a natural way the semantic definitions according to the denotational approach; in particular, it should allow to define domains (even recursive domains) and higher order functions. Furthermore
the metalanguage should provide a module construct to express the semantics in a structured way. Finally it should be an executable language; in this case the definition of the static semantics immediately provides a static checking algorithm, which can be considered as a non standard interpreter of the language, in the same way as the standard semantics provides a standard interpreter. The two interpreters will share most of their structure and will differ only in the implementation of some modules [5].

Appendix A

Let X and Y be two domains and f a function from X to Y; we define the following functions:
- \( \bar{f} : \mathcal{P}(X) \to \mathcal{P}(Y) \) is the function from subsets of X to subsets of Y defined in the obvious way:
  \[ \bar{f}(X') = \{ y \mid y = f(x'), x' \in X' \} \]
  where \( X' \in \mathcal{P}(X) \), \( y \in Y \);

- \( \bar{f} : \mathcal{P}(X) \) is defined as follows:
  \[ \bar{f}(y) = \{ x \mid f(x) \subseteq y, x \in X, y \in Y \} ; \]

- \( \sqcup : \mathcal{P}(Y) \to Y \) is the usual least upper bound function.

By using these definitions we can define \( h_F \) (from \( F_M = X_M \to Y_M \) to \( F_{SM} = X_{SM} \to Y_{SM} \)) according to the definition of Section 3.3 as follows:

\[ h_F(f_M) = h_X \circ \bar{f}_M \circ h_Y \circ \sqcup \]

(where the notation \((f \circ g)(x)\) means \( g(f(x)) \) for \( f : X \to Y, g : Y \to Z, x \in X \))

To prove continuity of \( h_F(f_M) \) we obviously need some continuity property of \( h_X \) that can be derived from backward continuity of \( h_X \). We introduce two definitions and two lemmas. The two definitions are an extension of the ones given in [26] to construct power domains.

**Definition.** An infinitary tree is a tree whose nodes may have an infinite number of sons.

**Definition.** Let \( X \) be a domain, and \( T \) a (node-) labeled infinitary tree satisfying

(i) for each node \( t \) the label \( l(t) \in X \),

(ii) \( T \) has no terminating branches, and

(iii) if \( t' \) is a descendant of \( t \) in \( T \), then \( l(t) \subseteq l(t') \).

Let \( L \) be a function which assigns to each (infinite) path \( \pi \) through \( T \) the least upper bound of the labels occurring along \( \pi \).

We say that \( T \) is an infinitary generating tree over \( X \), which generates the set

\[ S = \{ L(\pi) \mid \pi \text{ is a path through } T \} \]
We define $S_n \subseteq X$ as the cross section of $T$ at depth $n$ (that is, the set of labels of nodes at depth $n$). It is obvious that, each element $s \in S$ is the least upper bound of a chain whose elements $s_n$ belong to $S_n$ ($n = 1, 2, \ldots$).

**Lemma A.1.** Let $X$ and $Y$ be two domains and $f$ a continuous function from $X$ to $Y$. If $S_n \subseteq X$ ($n = 1, 2, \ldots$) are the cross sections of an infinitary generating tree $T_X$ over $X$ (which generates the set $S$), then $f(S_n)$ ($n = 1, 2, \ldots$) are cross sections of an infinitary generating tree $T_Y$ over $Y$ and $f(S)$ is the set generated by $T_Y$.

**Proof.** Obvious. $\square$

**Lemma A.2.** Let $T$ be an infinitary generating tree over a domain $X$, which generates the set $S \subseteq X$. Let $S_n$ ($n = 1, 2, \ldots$) be the cross sections of $T$, then

$$\bigcup_{n=1}^{\infty} (\bigcup S_n) = \bigcup S.$$

**Proof.** By the definition of generated set we know that for every $s \in S$ there is a chain $s_1 \subseteq s_2 \subseteq \cdots$ such that $s = \bigcup_{n=1}^{\infty} s_n$ and $s_n \in S_n$. We have

$$s_n \subseteq \bigcup S_n, \quad n = 1, 2, \ldots$$

and

$$s \subseteq \bigcup_{n=1}^{\infty} (\bigcup S_n), \quad n = 1, 2, \ldots.$$

Then, from the definition of least upper bound, we have

$$s \subseteq \bigcup_{n=1}^{\infty} (\bigcup S_n) \quad \text{for all } s \in S$$

and

$$\bigcup S \subseteq \bigcup_{n=1}^{\infty} (\bigcup S_n).$$

On the other hand, we have

$$s_n \subseteq \bigcup S, \quad s_n \in S_n, \quad n = 1, 2, \ldots$$

Then, from the definition of least upper bound

$$\bigcup S_n \subseteq \bigcup S, \quad n = 1, 2, \ldots$$

and

$$\bigcup_{n=1}^{\infty} (\bigcup S_n) \subseteq \bigcup S.$$

Thus the lemma is proved. $\square$
Now we can prove that, if \( h_x \) and \( h_y \) satisfy the following constraints:
- \( h_x \) and \( h_y \) are continuous,
- \( h_x \) and \( h_y \) are backward continuous (Property 3.1),
then \( h_F \) maps continuous functions in continuous functions and it is continuous and backward continuous.

**Theorem A.1.** \( h_F \) maps continuous functions in continuous functions, that is \( h_F(f_M) \) is continuous for all \( f_M \in F_M \).

**Proof.** \( h_F(f_M) \) is obviously monotonic. We must prove that for every chain in \( X_SM \),

\[
x_SM \subseteq x_SM^2 \subseteq \cdots \subseteq x_SM = \bigsqcup_i x_SM^i,
\]

then

\[
\bigsqcup_i (h_F(f_M))(x_SM^i) = (h_F(f_M))(x_SM).
\]

By applying \( h_x \) to the elements of the chain we obtain the sequence of sets

\[
h_x(x_SM^1), h_x(x_SM^2), \ldots, h_x(x_SM).
\]

From Property 3.1 of \( h_x \) we have

\[
X_M = \bigsqcup_i x_M^i, \quad x_M^i \subseteq x_M^{i+1}, \quad x_M^i \in h_x(x_SM^i)
\]

for all \( x_M \in h_x(x_SM) \),

and, by definition of \( h_x \), we have that every set of the sequence is contained in the next one.

Then, according to the above definitions, it is easy to see that, \( h_x(x_SM) \) is a set generated by an infinitary tree and \( h_x(x_SM) \) are cross sections at depth \( i \) (the root (depth 0) of the tree is labeled by \( 1 \)).

By using functions \( f_M \) and \( h_x \) and by applying twice Lemma A.1 we have that

\[
h_y(f_M(h_x(x_SM))), h_y(f_M(h_x(x_SM))), \ldots
\]

is a sequence of cross sections of an infinitary tree with labels in \( Y_SM \) generating the set \( h_y(f_M(h_x(x_SM))) \).

Then, from Lemma A.2, we have

\[
\bigsqcup_i (h_y(f_M(h_x(x_SM)))) = (h_y(f_M(h_x(x_SM)))),
\]

that is

\[
\bigsqcup_i (h_F(f_M))(x_SM^i) = (h_F(f_M))(x_SM).
\]
Theorem A.2. \( h_F \) is a continuous function.

Proof. \( h_F \) is monotonic by definition. Given a chain in \( F_M \) with its least upper bound
\[
\bigcup_{i=1}^{\infty} f_M^i = f_M^1 \subseteq f_M^2 \subseteq \cdots \subseteq f_M^i = \bigcup_{i=1}^{\infty} f_M^i,
\]
we must prove that for all \( x_{SM} \in X_{SM} \),
\[
\left( h_F \left( \bigcup_{i=1}^{\infty} f_M^i \right) \right)(x_{SM}) = (h_F(f_M))(x_{SM}),
\]
that is, by definition of \( h_F \),
\[
\bigcup_{i=1}^{\infty} \left( h_X(\bar{f}_M^i(\bar{h}_X(x_{SM}))) \right) = h_Y(\bar{f}_M(\bar{h}_X(x_{SM}))). \tag{A.1}
\]
By applying \( h_X \) to \( x_{SM} \) we obtain a set in \( X_M \).

It is easy to see that
\[
\bar{f}_M^i(\bar{h}_X(x_{SM})), \bar{f}_M^1(\bar{h}_X(x_{SM})), \ldots
\]
is a sequence of cross sections of an infinitary tree (with root labeled by \( \bot \)) generating the set \( \bar{f}_M(\bar{h}_X(x_{SM}))) \) in \( Y_M \).

By applying Lemmas A.1 and A.2 as in the proof of Theorem A.1, we trivially prove (A.1). \( \square \)

Theorem A.3. \( h_F \) is backward continuous.

Proof. We must prove that, given a chain of functions in \( F_SM \) with their least upper bound
\[
\bigcup_{i=1}^{\infty} f_SM^i = f_SM^1 \subseteq f_SM^2 \subseteq \cdots \subseteq f_SM^i = \bigcup_{i=1}^{\infty} f_SM^i,
\]
then, for every \( f_M \in F_M \) such that \( h_F(f_M) \subseteq f_SM \), we can find a chain of functions in \( F_M \)
\[
f_M^1 \subseteq f_M^2 \subseteq \cdots
\]
with the following properties:
\[
f_M = \bigcup_{i=1}^{\infty} f_M^i \tag{A.2}
\]
and
\[
h_F(f_M^i) \subseteq f_SM^i, \quad i = 1, 2, \ldots \tag{A.3}
\]
First we show how to obtain the functions \( f_M^i \) and then we prove that the above properties holds for these functions.
Let \( x_M \) be any value in \( X_{SM} \), and let us consider the chain in \( Y_{SM} \)

\[
f_{SM}^1(h_X(x_M)) \subseteq f_{SM}^2(h_X(x_M)) \subseteq \cdots \subseteq f_{SM}(h_X(x_M)) = \bigcup_{i=1}^{\infty} f_{SM}^i(h_X(x_M)).
\]

From the definition of \( f_M \) it is easy to derive that

\[
h_Y(f_M(x_M)) \subseteq f_{SM}(h_X(x_M)).
\]

Then, from the backward continuity of \( h_Y \), we know that we can find a chain in \( Y_M : y_M^1 \equiv y_M^2 \equiv \cdots \) such that

\[
f_{M}(x_M) = \bigcup_{i=1}^{\infty} y_M^i
\]

and

\[
h_Y(y_M^i) \subseteq f_{SM}^i(h_X(x_M)), \quad i = 1, 2, \ldots.
\]  \hspace{1cm} (A.4)

Now, we define \( f_M' \) as follows:

\[
f_M'(x_M) = y_M.
\]

This construction obviously satisfies property (A.2). We show now that (A.3) holds as well.

Let \( x_{SM} \) be any value in \( X_{SM} \), and let \( x_M \) be a value in \( X_M \) such that \( x_M \in \bar{h}_X(x_{SM}) \), that is \( h_X(x_M) \equiv x_{SM} \).

From property (A.4) we know that

\[
h_Y(x_{SM}) \subseteq \bar{h}_X(x_{SM}).
\]

Because \( h_X(x_M) \subseteq x_{SM} \) and \( f_{SM}^i \) is a monotonic function we have

\[
h_Y(f_M'(x_M)) \subseteq f_{SM}^i(x_{SM}) \quad \text{for all } x_M \in \bar{h}_X(x_{SM}).
\]

But \( (h_Y(f_M'))(x_{SM}) \) is the least upper bound of all \( h_Y(f_M'(x_M)) \) such that \( x_M \in \bar{h}_X(x_{SM}) \), and thus (A.3) holds. \( \square \)

Appendix B

**Theorem B.1.** If all component operations of \( E \) satisfy conditions (2.2) and (2.3), then (3.1) holds.

**Proof.** From the hypothesis we have:

\[
h_D^2(E_M(d_M^{1})) \subseteq E_{SM}(d_{SM}^{1}) \quad \text{for all } d_M^{1} \in D_M^{1} \text{ and } d_{SM}^{1} \in D_{SM}^{1} \text{ such that } h_D^2(d_M^{1}) \subseteq d_{SM}^{1}.
\]

Let \( d_{SM}^{1} \) be a value of \( D_{SM}^{1} \). By applying the left member of (3.1) to \( d_{SM}^{1} \) we have

\[
\sqcup \{ d_{SM}^{2} | d_{SM}^{2} = h_D^2(E_M(d_M^{1})) \text{ where } h_D^2(d_M^{1}) \subseteq d_{SM}^{1} \}.
\]  \hspace{1cm} (B.2)
On the other hand, by applying the right member of \((3.1)\) to \(d_{SM}^1\) we have
\[
E_{SM}(d_{SM}^1). \tag{B.3}
\]
By (B.1) we have that each element of the set in (B.2) is \(\subseteq\) the value (B.3) and the same is true for the least upper bound.

This proof holds for every \(d_{SM}^1 \in D_{SM}^1\) and then (3.1) is true. \(\square\)

**Theorem B.2.** If the function \(h_D\) is strict and continuous, then condition (2.2) holds for least fixed point, that is
\[
h_D(\text{fix}_M(f_M)) \sqsubseteq \text{fix}_SM(h_M(f_M)) \quad \text{where } F : D \to D.
\]

**Proof.** By induction we prove
\[
h_D(f_M^n(\bot_{D_M})) \sqsubseteq (h_F(f_M))^n(\bot_{D_{SM}}). \tag{B.4}
\]
This condition is true for \(n = 0\) because \(h_D\) is strict.

Let us assume (B.4) true for \(n - 1\). From properties (2.2) and (2.3) for functional application and from the inductive hypothesis we have
\[
h_D(f_M^n(\bot_{D_M})) = h_D(f_M(f_M^{n-1}(\bot_{D_M})))
\]
\[
\sqsubseteq (h_F(f_M))((h_D(f_M))^{n-1}(\bot_{D_M}))
\]
\[
= (h_F(f_M))^n(\bot_{D_{SM}}).
\]
Thus (B.4) holds.

From (B.4) we obtain
\[
h_D(f_M^n(\bot_{D_M})) \sqsubseteq \bigcup_{n=0}^{\infty} (h_F(f_M))^n(\bot_{D_{SM}}) = \text{fix}_SM(h_M(f_M)). \tag{B.5}
\]
From the continuity of \(h_D\) and from \(B.5\) we derive
\[
h_D(\text{fix}_M(f_M)) = h_D\left(\bigcup_{n=0}^{\infty} f_M^n(\bot_{D_M})\right)
\]
\[
= \bigcup_{n=0}^{\infty} (h_D(f_M^n(\bot_{D_M}))) \sqsubseteq \text{fix}_SM(h_M(f_M)). \quad \square
\]

**References**


