THE META-LANGUAGE: A REFERENCE MANUAL

Cliff B.Jones

Abstract:

The recent work of the Vienna Laboratory on the sub= ject of semantic definitions has used the "denotational semantics" approach. Although this is a clear break with the earlier abstract interpreter approach, the newer meta-language has tried to preserve and even improve upon the readability of the earlier "*VDL*" no= tation. The meta-language described here has been used in the definitions of large programming languages and systems. This paper is not a tutorial; rather it pro= vides a reference document for the meta-language.

CONTENTS

0. INTRODUCTION

This paper is intended to provide a reference document which describes the meta-language used throughout this volume. It is equally important to appreciate which objectives are not being aimed for here. On the one hand this document is definitely not a tutorial, neither for denotational semantics nor for the specific meta-language used. In much of this paper no motivation at all is provided, the reader who is not familiar with the general ideas is recommended to read Bjørner 78b before attempting to use this paper. On the other hand this document does not provide a formal "foundation" for the meta-language in the style of, say, Mosses 75.

The aim of this "reference manual" is to provide a document where readers or writers of the meta-language can find a description of its constructs in terms of some other notation. As such, this document does not necessarily introduce all terms and notation in a strict left-to-right order.

The notation to be introduced is primarily intended for the purpose of defining languages or systems using the approach known as "denotational semantics"; although it can also be used as a sort of programming language this is not its purpose. A general "denotational" approach can be expected to yield a definition which can be viewed in four parts:

i) Abstract syntax of the language to be studied
ii) Context Conditions
iii) Semantic Objects
iv) Meaning function(s)

Before embarking on defining the meaning of a language it is obviously essential to fix what ("object") language is to be studied: this is the task of the first two parts of a definition. The abstract syntax defines a class of objects which are abstractions of the actual texts of the language. (The class of texts would be described by a concrete syntax: this subject is widely documented and in consequence need not occupy space here). The abstract syntax of a language is "context free" and it is thus impossible to restrict the class of abstract texts to those which are obviously meaningful. For example, in a programming language it is impossible to specify that all valid programs must use only declared identifiers. Rather than leave such checking to cloud the meaning functions, a separate section defines predicates over the objects of the abstract syntax. Such predicates define a sub-set of the objects which are specified by the abstract syntax; those objects which satisfy the context conditions are said to be "well-formed". The meaning functions provide denotations only for well-formed objects.

The third part of a definition defines sets of semantic objects. Both the denotations chosen for a language (e.g. continuous functions over states) and any auxiliary objects such as environments are described. In a sense, this part of a definition is unnecessary! The abstract syntax is the only way in which the class of source objects is delimited (unless a translator function from concrete syntax is written); the denotations and auxiliary objects which can be created by the meaning functions could, in fact, be deduced from those functions. However, the separate description of semantic objects is both an invaluable aid to the reader of a definition and an important tool to be used during the development of the meaning functions.

The first three parts of a definition, then, have defined a class of well-formed abstract objects and a class of semantic objects: meaning is defined by a function mapping the former into the latter. This mapping will be defined by a family of (recursive) functions. An objective of a denotational definition is that the meaning of a composite object should be created from the meanings of its components. Given this objective there will normally be one semantic function for each sub-class of abstract objects.

Having outlined the "denotational method" in general, a few comments can now be made on the spefic meta-language used in this volume. (The notation was referred to within the Vienna Lab as "Meta-IV"). It is important to appreciate that the authors do not regard the meta-language as any sort of fixed standard. Rather, it is offerred as a basis which has been shown to be adequate for a variety of definitions, and which can be expected to be adequate for defining other related systems. Even for such systems the existance of a body of notation does not eliminate the difficulties of choosing suitable abstractions for use in their definition. Moreover, it must be realized that an attempt to apply the ideas of denotational semantics to systems of an entirely different nature will require the invention of suitable extensions to the notation presented here. There does, however, appear to be some virtue in a degree of standardisation in the basic notation.

The notation used by the Oxford University group (cf. Stoy 74) for denotational semantics definitions is rather different in appearance from that presented here and it may be useful to speculate as to the cause. The Oxford group have been principally concerned with questions of foundations and have worked with relatively small languages. They have, in fact, chosen languages which illustrate the key problems with a minimum of unnecessary detail. In contrast, the Vienna group have tended to tackle larger (given) languages and systems. Furthermore, the languages have usually been defined "warts and all". For a definition of a certain size it becomes virtually impossible to remember enough conventions to permit use of single character names, type clauses given implicitly by Greek letters chosen for parameters etc. Thus, for large languages a different style is adopted to provide a readable definition. For example, a syntax is employed which is as abstract as possible, (longer) function names are used in the hope that they are suggestive to the reader etc. It is interesting to compare the relative merits of succinctness and readability on Mosses 74 and Henhapl 78.

The various topics are distributed throughout this paper as follows. Section 4 is concerned with the abstract syntax notation. The actual objects (and their operators) which have been found to be of use in earlier applications of this meta-language, are described in sections 2 and 3. Sections 1 and 5 briefly outline the logic notation used. The heart of the meta-language is the means for defining and combining meaning functions: this is described in section 6. The subject of arbitrary order or merging of operations is completely omitted from this paper for reasons given in section 6.

An outline concrete syntax for the meta-language is given in appendix I and appendix II lists some conventions which have been used in earlier definitions. Appendix III contains a definition of a small language, close study of which should provide the reader with a clear understanding of the use of the notation.

1. LOGIC NOTATION

In defining any language something has to be taken as a basis. For the description of the operators of the meta-language, the first-order predicate calculus with equality is chosen. There are two reasons for this choice. Firstly, a consistent and complete axiomatic definition can be provided (e.g. Kleene 67). Secondly, it is widely enough understood that the presentation here can be restricted to providing "readings" for the particular symbols chosen.

Fig. 1-1 displays the notation adopted throughout this paper. The constraints on bounded quantifiers are used in preference to implications (Thus:

$(\forall x \in Nat^0)(\neg is\text{-}prime(4 \cdot x))$, rather than
$(\forall x)(x \in Nat^0 \Rightarrow \neg is\text{-}prime(4 \cdot x))$)

in order to reduce the number of "undefined" operands. However, the problem of the meaning of the logical operators with undefined values must be faced and is discussed in section 5. The constraints on quantifiers may be omitted where the context makes them obvious.

The (iota) description operator yields the unique object satisfying a predicate. It is an error to use the operator if no, or more than one, value satisfies the predicate. Thus:

$$(\exists! x_0 \in X)(p(x_0)) \Rightarrow p((\iota x \in X)(p(x)))$$

Here again, the constraint may be omitted if it is clear from the context.

Symbol	Reading
<u>*TRUE*</u>	*truth values*
<u>*FALSE*</u>	
$\&$	*and*
$\vee$	*or*
$\Rightarrow$	*implies*
$\Leftrightarrow$	*equivalence*
$\neg$	*not*
$\forall$	*for all*
$\exists$	*there exists*
$\exists!$	*there exists exactly one*
ι	*the unique object*
$(\forall x \in X)(p(x))$	*for all members of set X, p(x)*
$(\forall x \mid c(x))(p(x))$	*for all x satisfying c, p(x)*

similarly for other quantifiers

fig. 1-1: Logic notation

2. ELEMENTARY OBJECTS

In writing a definition it will normally be necessary to use objects which are structured (i.e. composite). Some types of composite objects, together with their operators, are described in section 3. A definition will, however, also have to employ certain basic objects whose structure is of no interest for the system being defined. For example, many definitions will require natural numbers but will wish to treat them as elementary objects.

Two standard objects, the truth values, have already been introduced:

TRUE, *FALSE*

Another object which will be explained below (roughly, it is a place holder for an omitted branch in a tree) is:

NIL

An author may also enumerate any other objects required for his definition in a notation explained in section 4. The only property which is assumed about elementary objects is that they are distinguishable. Thus two meaningful operators are the equality operators ($=,\neq$). Among the elementary objects to be enumerated for a definition are "references" (see section 6.1).

In addition to those objects which can be enumerated, a definition will usually also require known objects like natural numbers, integers, reals etc. With such familiar sets one may also wish to adopt some of their standard operators (e.g. $<,\leq,\surd$). Any definition should include a list of such assumed objects and their operators. Section 3.1 lists some suggested names for sets of such objects. All definitions will be assumed to use the standard propositional connectives on the truth values.

3. COMPOSITE OBJECTS

A definition will have to define a number of classes of objects, not least the texts of the language or system being defined. The style of defining classes of objects is described in section 4. This section will introduce a number of classes of objects which are useful in building abstractions appropriate for most systems. The objects given here, in distinction to those discussed in section 2, have structure. Thus operations will be introduced for manipulating (building), decomposing and interrogating the objects of each class. The test for equality is available for each class and will be defined.

The objects are defined in the first place in terms of the logic notation introduced in section 1; there is then a layer by layer construction of each more complex class of objects. Because of this construction, certain objects of different classes are formally identical:

in fact definitions will ensure that the types are not mixed.

<u>3.1</u> <u>Sets</u>

Sets are characterized by the members they contain. Testing whether an object is a member of a set is achieved by a two place infix operator:

$e \in S$

This is a propositional expression and thus yields <u>*TRUE*</u> or <u>*FALSE*</u>.

Its converse is written:

$e \notin S \Leftrightarrow \neg(e \in S)$

Fig. 3-1 uses the test for membership to define the set operators. Notice that the distributed <u>*union*</u> is defined only for sets of sets.

Set Operator	Definition
$S = T$	$e \in S \Longleftrightarrow e \in T$
$S \cup T$	$\{x \mid x \in S \vee x \in T\}$
<u>*union*</u> S	$\{e \mid (\exists s \in S)(e \in s)\}$
$S \cap T$	$\{x \mid x \in S \;\&\; x \in T\}$
$S - T$	$\{x \mid x \in S \;\&\; x \notin T\}$
$S \subseteq T$	$e \in S \Rightarrow e \in T$
$s \subset T$	$S \subseteq T \;\&\; S \neq T$
<u>*power*</u>S	$\{s \mid s \subseteq S\}$

Fig. 3-1: Set operators

The basic way of constructing sets is by implicitly defining, via a predicate, those elements of some other set which are to be included:

$$x_0 \in \{x \in S \mid p(x)\} \Leftrightarrow (x_0 \in s \ \& \ p(x_0))$$

Where some set is clearly implied as the range by the context in which a formula appears, the "$\in S$" can be omitted.

It is possible to view the explicit enumeration of the elements of a (finite) set as an abbreviation:

$$\{x_1, x_2, \ldots, x_n\} \triangleq \{x \mid x=x_1 \vee x=x_2 \vee \ldots \vee x=x_n\}$$

In particular, for the empty set:

$$\{\} \triangleq \{x \mid x \neq x\}$$

Thus:

$$\neg(\exists x_0)(x_0 \in \{\})$$

For two integers a set of integers can be defined by the abbreviation:

$$\{i:j\} \triangleq \{x \in Int \mid i \leq x \leq j\}$$

Where context makes clear which variable(s) is (are) to be considered bound:

$$\{f(x) \mid p(x)\} \triangleq \{z \mid (\exists x)(p(x) \ \& \ z=f(x))\}$$

The number of elements in a finite set can be determined by:

$$\underline{card}\ \{\} = 0$$
$$x_0 \notin S \Rightarrow \underline{card}\ (S \cup \{x_0\}) = \underline{card}\ S + 1$$

It is well known that an unconstrained view of sets will permit the possibility of paradoxical sets (e.g. $\{S \mid S \notin S\}$): starting from non-paradoxical sets and using the operators given here this danger does not exist.

The following standard sets will be used:

$$Bool = \{\underline{TRUE}, \underline{FALSE}\}$$
$$Nat \ = \{1, 2, \ldots\}$$

$Nat^0 = \{0, 1, 2, \ldots\}$
$Int = \{\ldots, -1, 0, 1, \ldots\}$

3.2 Pairs

In building the subsequent notion of *MAP* it will be necessary to have a notation for ordered pairs: such a notation is introduced in this section but will not be used other than within section 3 (a general tuple notation is developed in section 3.4.)

The objects considered here will be best understood as ordered pairs:

$$pair(e_1, e_2) \in PAIR$$

They, and their operations, are defined via sets:

$$pair(e_1, e_2) = \{e_1, \{e_1, e_2\}\}$$

Selection of elements is achieved by:

$$first(pr) = (\iota e_1)(\exists e_2)(pr = \{e_1, \{e_1, e_2\}\})$$
$$second(pr) = (\iota e_2)(\exists e_1)(pr = \{e_1, \{e_1, e_2\}\})$$

Notice that it is an immediate consequence of the definition that:

$$pair(e_1, e_2) = pair(e_1', e_2') \Leftrightarrow (e_1 = e_1' \ \& \ e_2 = e_2')$$

3.3 Maps

The *Maps* to be introduced here are a restiction of the general functions to be covered in section 3.6. The restriction is that the domain of a map is finite and constructable (i.e. every way of generating a map also shows how to compute the domain). The usefulness of this restricted class of functions (along with its separate notation) comes from their use in a definition.

The basic model is a set of pairs which pair a unique second element with any first element:

for $m \in Map$:

$$p_1, p_2 \in m \;\&\; first(p_1) = first(p_2) \Rightarrow p_1 = p_2$$

For some *MAP m*, the operations of domain, application and range are defined in terms of the set model:

$$\underline{dom}m = \{d \mid (\exists p \in m)(d = first(p))\}$$
$$m(d) = (\iota r)(pair(d,r) \in m)$$
$$\underline{rng}m = \{m(d) \mid d \in \underline{dom}m\}$$

The actual computation of the domain of a map can be shown for each of the map generating expressions below. (The range operation is also recursive because domain is). Notice that applying a map to an element outside its domain is undefined.

The map constructors and operators are defined in *fig. 3-2*. The basic way of constructing maps is by implicitly defining the pairs they should contain. Strictly, the set from which *d* is chosen should be specified but this will be assumed to be given by the context. As with sets a finite enumeration of the elements in a map is regarded as an abbreviation.

Notation	Meaning	Domain
$[d \mapsto r \mid p(d,r)]$	$\{pair(d,r) \mid p(d,r)\}$	$\{d \mid (\exists r)(p(d,r))\}$
$[d_1 \mapsto r_1, \ldots, d_n \to r_n]$	$[d \mapsto r \mid (d=d_1 \;\&\; r=r_1) \vee \ldots (d=d_n \;\&\; r=r_n)]$	$\{d_1, \ldots, d_n\}$
for: $\underline{dom}m \cap \underline{dom}n = \{\}$ $m \cup n$	$m \cup n$	$\underline{dom}m \cup \underline{dom}n$
$m+n$	$[d \mapsto r \mid d \in \underline{dom}n \;\&\; r=n(d) \vee d \in (\underline{dom}m - \underline{dom}n) \;\&\; r=m(d)]$	$\underline{dom}m \cup \underline{dom}n$
$m \mid s$	$[d \mapsto m(d) \mid d \in (\underline{dom}m \cap s)]$	$\underline{dom}m \cap s$
$m \setminus s$	$[d \mapsto m(d) \mid d \in (\underline{dom}m - s)]$	$\underline{dom}m - s$
for: $\underline{rng}n \subseteq \underline{dom}m$ $m \cdot n$	$[d \mapsto m(n(d)) \mid d \in \underline{dom}n]$	$\underline{dom}n$

fig. 3-2: Maps

Notice that the union operator is defined between maps whose domains are disjoint. The normal (set) union symbol will be used because it is clear that maps are being combined.

3.4 Tuples

The objects described here will be familiar as unbounded (finite) lists, but following Reynolds 76 we yield in the face of the established use in computing of this term. (Although the term "list" will be used in informal discussions).

Tuples are modelled on maps and are either empty or have a head and a tail component. The tail component of a tuple is always a tuple. Tuples are finite, that is after a finite number of selections of the tail component an empty tuple will be located.

$$\begin{aligned} t \in TUPLE \Rightarrow (&\underline{dom}\, t = \{\} \vee \\ &\underline{dom}\, t = \{\underline{HD}, \underline{TL}\} \mathbin{\&} t(\underline{TL}) \in TUPLE) \end{aligned}$$

The tuple notation (including explicit enumeration) is defined in *fig. 3-3*. Unlike sets, tuples provide an ordered access to their elements (by $\underline{hd}$ or indexing). For this reason care must be taken in the choice of an implicit list notation. In order to ensure that an order is defined for the created list, generation is defined only for (modification of) a sub-list of a given list. Thus:

$$<f(tup(i)) \mid p(tup(i))>$$

Notice that the distributed concatenation ("*conc*") is only defined for tuples of tuples.

Notation	Definition
<>	[]
$<e_1,\ldots,e_n>$	$[\underline{HD}\rightarrow e_1,\ \underline{TL}\rightarrow[\ldots,\underline{TL}\rightarrow[\underline{HD}\rightarrow e_n,\underline{TL}\rightarrow[\,]]\ldots]]$
for tup≠<>: $\underline{hd}tup$	$tup(\underline{HD})$
for tup≠<>: $\underline{tl}tup$	$tup(\underline{TL})$
$\underline{len}tup$	$(\underline{HD}\notin\underline{dom}tup\rightarrow 0, T\rightarrow\underline{len}\,\underline{tl}tup+1)$
for $1\le i\le\underline{len}tup$: $tup(i)$	$(i=1\rightarrow\underline{hd}tup,\quad T\rightarrow(\underline{tl}tup)(i-1))$
$tup1\frown tup2$	$(\iota\, tup)(\underline{len}tup = \underline{len}tup1+\underline{len}tup2\ \&\ (1\le i\le\underline{len}tup1 \Rightarrow tup(i)=tup1(i))\ \&\ (1\le i\le\underline{len}tup2 \Rightarrow tup(i+\underline{len}tup1)=tup2(i)))$
$\underline{conc}\ tt$	$(tt=<> \rightarrow <>,\ T\rightarrow\underline{hd}tt\frown\underline{conc}\,\underline{tl}tt)$
$\underline{elems}tup$	$\{tup(i) \mid 1\le i\le\underline{len}tup\}$
$\underline{inds}tup$	$\{1:\underline{len}tup\}$

fig. 3-3: tuple notation

3.5 Trees

In order to define structures which correspond to (abstract forms of) programs etc., it will be necessary to have a way of combining instances of objects into new objects and these combinations must be recognisable and decomposable. Such objects will be built by " constructor functions" the only essential property of which is their uniqueness.

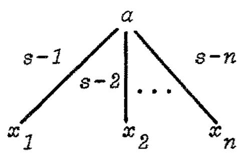
$$mk\text{-}a(x_1,x_2,\ldots,x_n) = mk\text{-}a'(x_1',x_2',\ldots,x_n')$$
$$\Longleftrightarrow (a=a'\ \&\ x_1=x_1'\ \&\ x_2=x_2'\ \&\ \ldots\ \&\ x_n=x_n')$$

(The names of constructor functions are always formed by the prefix "*mk-*" and the name of the relevant abstract syntax rule: for the relation to abstract syntax see section 4).

One technique for decomposing a constructed object will be to define "selector functions" in the constructor:

$$mk\text{-}a(s\text{-}1{:}x_1,\ s\text{-}2{:}x_2,\ \ldots,\ s\text{-}n{:}x_n) = a$$
$$\Rightarrow s\text{-}1(a) = x_1\ \&\ s\text{-}2(a) = x_2\ \&\ \ldots\ \&\ s\text{-}n(a) = x_n$$

Notice that, although the selector functions do not play a part in the distinction between objects, the rules about object description (see section 4) prevent any confusion. A convenient intuitive view of the result of a constructor function is a labelled tree, thus:



Apart from using selector functions, an object may be decomposed by writing the constructor in a defining position (e.g. parameter name, left-hand-side of a "<u>*let*</u>"). This achieves a complete decomposition in one step. Using the <u>*let*</u> construct of section 3.6:

$$\underline{let}\ mk\text{-}a(s\text{-}1{:}n_1,\ s\text{-}2{:}n_2,\ \ldots,\ s\text{-}m{:}n_m) = a \quad \triangleq \left\{\begin{array}{l} \underline{let}\ n_1 = s\text{-}1(a) \\ \underline{let}\ n_2 = s\text{-}2(a) \\ \quad\vdots \\ \underline{let}\ n_m = s\text{-}m(a) \end{array}\right.$$

<u>3.6 Functions</u>

We shall be interested in defining functions over defined domains which deliver results in defined ranges. This, so called type, information will be written:

$$f\colon D \to R$$

If the domain is, in fact, a cartesian product of sets, this is written (without "$\times$" symbols):

$$f\colon D_1\ D_2\ \ldots\ D_n \rightarrow R$$

Similar extensions hold for the range. (Strictly all functions take one argument which is a tuple, but the actual tupling and decomposition will be systematically omitted).

The simpler functions to be defined will define a result for any value in their domain: they are total and the undecorated arrow will be used to separate the domain and range. Functions which may be undefined for some input values are said to be partial and will use the symbol $\tilde{\rightarrow}$, for example:

$$Tr = \Sigma \tilde{\rightarrow} \Sigma$$

It is assumed that the reader is familiar with the standard ways of writing and naming functions even where recursion is involved, thus:

$$f(x) = \ldots\ x\ \ldots\ f\ \ldots$$

A notation will be required which enables the creation of functions without having to provide names. A brief introduction to the lambda notation is given here, for further information consult Stoy 74.

Instead of defining f by some expression in terms of its arguments (e.g.

$$f(x) = e(x)\ ),$$

the lambda notation provides a way of defining instances of functions in the form "λ" followed by argument list followed by "." followed by defining expression, thus:

$$\lambda x.e(x)$$

Named functions can be applied to arguments and a simple symbol substitution for the parameters can be used to determine the value. In simple cases, the same rule will provide an evaluation mechanism for lambda expression application, thus:

$(\lambda x.5)(7) = 5$

$(\lambda x.x)(7) = 7$

$(\lambda x.(\lambda y.x+y))(7) = \lambda y.7+y$

Notice that this last application yields a function as result. We can now define the identity function:

$I = \lambda x.x$

The first operator to be considered for functions is functional composition:

$f^{\bullet}g = \lambda x.f(g(x))$

The "$\underline{let}$" notation can now be introduced. The purpose of this notational device is to permit the introduction of local abbreviations which can then be used in an expression. Thus:

$$\left.\begin{array}{ll} & \underline{let}\ x = \ldots\ \underline{in}\ e(x) \\ \text{or} & \underline{let}\ x = \ldots \\ & \quad e\ (x) \end{array}\right\} \triangleq (\lambda x.\,e(x))(\ldots)$$

i.e. "$\underline{in}$" can be dropped if a new line is used!

Beyond this basic use, the $\underline{let}$ notation will be used to introduce recursive definitions:

$\underline{let}\ f = \ldots\ f\ \ldots$

with the meaning that f is to name the least fixed point (see Park 69) of the defining equation.

Several notational devices will permit the specification of different ways of computing a result. The basic conditional expression form should be familiar:

$\underline{if}\ p\ \underline{then}\ e_1\ \underline{else}\ e_2$

A multiple (two or greater) split can be achieved by the case construct:

$$\left.\begin{array}{l} \underline{cases}\ v: \\ v_1 \rightarrow e_1 \\ v_2 \rightarrow e_2 \\ \\ \\ \\ v_{n-1} \rightarrow e_{n-1} \\ T \quad \rightarrow e_n \end{array}\right\} \triangleq \left\{\begin{array}{l} \underline{if}\ v = v_1\ \underline{then}\ e_1 \\ \underline{else}\ \underline{if}\ v = v_2\ \underline{then}\ e_2 \\ \underline{else}\ \underline{if} \\ \quad\vdots \\ \underline{else}\ \underline{if}\ v = v_{n-1}\ \underline{then}\ e_{n-1} \\ \underline{else}\qquad\qquad e_n \end{array}\right.$$

Alternatively a multiple split can be achieved by the "McMarthy conditional expression" form:

$$p_1 \rightarrow v_1, p_2 \rightarrow v_2, \ldots, p_{n-1} \rightarrow v_{n-1}, T \rightarrow v_n$$
$$\triangleq \left\{\begin{array}{l} \underline{if}\ p_1\ \underline{then}\ v_1 \\ \underline{else}\ \underline{if}\ p_2\ \underline{then}\ v_2 \\ \underline{else}\ \underline{if} \\ \quad\vdots \\ \underline{else}\ \underline{if}\ p_{n-1}\ \underline{then}\ v_{n-1} \\ \underline{else}\ v_n \end{array}\right.$$

In either the case construct or conditional expressions the "T" clause can be omitted if the preceding cases cover all possibilities. If, however, none of the tests are satisfied the result is undefined.

The subject functions, and their definition, is returned to in section 6.

3.7 Implicit Definition

As has been pointed out earlier, the meta-language presented here should not be considered to be closed in any way. One particular way in which it is "open-ended" is in the ability to use objects whose characterisation is implicit.

Such definitions are usually difficult (see Bekić 70b) and in languages where they can be avoided a constructive form is preferred (cf. appendix III and Henhapl 78). For general approaches to the problem of implicit definition see Guttag or Liskov 75.

4. OBJECT DESCRIPTION

If a definition consisted solely of functions creating objects, the only use for describing classes of objects would be as a comment for the reader. This is, in fact, the case for many objects and, for example, one could deduce from a definition the class of possible states which could be generated. There is, however, another use for object descriptions which amounts to a necessity. When a definition is to be presented, one must have a way of defining the exact class of objects which is to be considered. Such a definition will not normally be practical via its generation function and it is then a requirement to have a notation for describing classes of objects.

In a simple language it would be possible to associate meaning directly with concrete forms of the object language texts. Some languages offer a "rich" choice of representations which are semantically equivalent. An abstract syntax not only offers a way of defining some "normal form" for such expressions, it also omits the details which are present only to ensure unambiguous parsing. Judicious choice of objects used (e.g. sets of declarations) can also shorten the semantic descriptions.

Objects will be described by an abstract syntax. (cf. concrete syntaxes which define a set of strings). The members of the set of rules comprising an abstract syntax will each define and name a set of objects. Each abstract syntax class will be defined as follows:

$$NAME\ \{=|::\}RULE$$

By convention the first character of a *NAME* is an upper case character. The choice between the "*::*" and "*=*" definition symbols and the form of the *RULE* dictate the set of objects which is to be associated with *NAME*. The majority of this section is concerned with defining the sets given by the various possibilities. Before embarking on this we note that a predicate which tests for membership of a particular class of objects is implicitly defined for each *NAME*. Thus

$$is\text{-}\theta(t) \triangleq t \in \Theta$$

where Θ is a *NAME* defined within an abstract syntax.

A *RULE* consists of one or more *TERMS* separated by vertical bars (i.e. "|", read as "or").

TERMS are built up by juxtaposed *ITEMS*. Unless a term contains exactly one item it is considered to be bracketed. *ITEMS* are either *NAME*s, modified *NAMES*, lists of underscored symbols, or bracketed *TERM*s.

Within a *TERM*, the meaning of a *NAME* is the set of objects defined by its rule; that of a modified *NAME* is explained later; that of a list of underscored symbols is the unit set containing an elementary object distinct from that denoted by any other string of underscored characters; a bracketed *TERM* denotes a constructed (i.e. tree-like) object. This last case requires more explanation. Section 3.5 has introduced constructor functions and the possibility to view them as trees. A bracketed term denotes the set of objects obtained by applying a constructor to elements of the cartesian product of the sets denoted by the *ITEMS* within the brackets. If the term occurs alone to the right of a "*::*" symbol, the name of the constructor is the *NAME*, found on the left hand side of the "*::*" symbol, preceded by "*mk-*". Thus:

$$N :: N_1 \; N_2 \; \ldots \; N_m$$

defines

$$N = \{mk\text{-}n(n_1,n_2,\ldots,n_m) \mid n_1 \in N_1 \;\&\; n_2 \in N_2 \;\&\; \ldots \;\&\; n_m \in N_m\}$$

The other contexts in which a bracketed *TERM* has been allowed above are as an option (separated by "or" form other *TERMS*) or as an *ITEM* within a *TERM*. In either of these cases the name of the constructor is "*mk*-α" (i.e. α is a name not used for any rule). Notice that this can introduce (sub-)classes of objects which are not distinguishable.

The set of objects denoted by an "*=rule*" (which contains zero or more "or"s) is the union of the sets denoted by the *TERMS* (which are separated by the "or"s). *Fig. 4-1* provides some examples of rules and the set of objects satisfying them.

Rule	set of objects
$A=B$	B
$A::B$	$\{mk\text{-}a(b) \mid b \in B\}$
$A=B \mid C$	$B \cup C$
$A=(BC) \mid D$	$\{mk\text{-}\alpha(b,c) \mid b \in B \ \& \ c \in C\} \cup D$
$A::C \mid D$	not defined
$A=B \mid (C \mid D)$	erroneous

fig. 4-1: examples of rules

"Modified names" and the sets of objects they denote are given in *fig. 4-2*.

Modified Name	Set denoted
$N\text{-}set$	$\underline{power}\ N$
N^*	$\{l \in TUPLE \mid \underline{1 \leq i \leq len l} \Rightarrow l(i) \in N\}$
N^+	$\{l \in N^* \mid \underline{len l \geq 1}\}$
$[B]$	$B \cup \{\underline{NIL}\}$
$B \underset{m}{\rightarrow} C$	$\{m \in MAP \mid \underline{dom} m \subseteq B \ \& \ \underline{rng} m \subseteq C\}$
$B \underset{m}{\leftrightarrow} C$	$\{m \in (B \underset{m}{\rightarrow} C) \mid m(d_1)=m(d_2) \Rightarrow d_1=d_2\}$
$B \rightarrow C$	continuous functions
$B \overset{\sim}{\rightarrow} C$	partial functions

where: N, B and C can be *TERMS* (bracketed by implication)

fig. 4-2: Meaning of modified names

There are occasions where a class of map objects is to be defined in terms of other map types. If the new map is defined over the union of the domains a freedom is introduced which may be unwanted. This can be avoided by using:

$$M = M1 \, \underline{\cup} \, M2$$

which defines M to be a set of maps:

$$\begin{aligned}\{m\in MAP \mid\ & \underline{dom}m = \underline{dom}M1 \cup \underline{dom}M2 \ \& \\ & \underline{rng}m = \underline{rng}M1 \cup \underline{rng}M2 \ \& \\ & (a\in\underline{dom}M1 \Rightarrow m(a)\in\underline{rng}M1) \ \& \\ & (a\in\underline{dom}M2 \Rightarrow m(a)\in\underline{rng}M2)\}\end{aligned}$$

A used constructor can be compared with abstract syntax rules when the syntax described is of internal (semantic) objects. Because constructors can also be used to decompose texts of the object language, a comparison is also possible between the abstract syntax of the texts being defined and their use in the semantic rules. In all cases where such a comparison is possible the used constructors, their selectors and their arguments must conform to the rules describing the class of objects.

5. THE RÔLE OF "UNDEFINED"

The treatment of logic in section 2 is somewhat over-simplified in that (in common with most text books) the possibility that expressions might be undefined has been ignored. Although the frequency has been reduced by the use of bounded quantifiers, expressions like:

$$d\in\underline{dom}m \ \& \ m(d)\in c$$
$$x = 0 \lor y/x>10$$

will be found. For such expressions the "conditional expression" forms (cf. Walk 69) are assumed. For example:

$$a \ \& \ b \triangleq \underline{if} \ a \ \underline{then} \ b \ \underline{else} \ \underline{FALSE}$$

This minimizes the danger of expressions being made undefined by their operands. (Notice that Jones 72 and Dijkstra 76 adopt separate symbols for the conditional forms.)

6. DEFINING SEMANTICS

As explained in the introduction, the general approach to defining the semantics of a language is to define a function which maps its (well-formed) elements to some class of understood denotations. For the languages which are most commonly considered interesting these denotations will have to be functions. Because most computer languages embody some form of assignment operation the functions will normally be state transformations (functions from states to states). In languages which have a procedure concept in its full generality the state transformations will, in fact, have to be very general functions which can be applied to functions (even themselves) and yield functions as results. If such general functions were permitted without restriction, it would be possible to generate the sort of paradoxes which Russell observed in set theory. However, it has been shown (cf. Scott 71) that the restriction to monotone and continuous constructions ensures that a model can be constructed and thus guarantees that inconsistencies are avoided. As this result is both of considerable importance and difficulty, it would be inappropriate to repeat that work here (for an excellent tutorial see Stoy 74). This paper will not, then, provide a complete treatment of the foundations of the meta-language. Rather, the approach taken here is to accept (gratefully) that models of the lambda calculus exist and to define meta-language expressions by specifying the closed lambda expressions into which an object program is mapped. In fact, a few extra combinators are adopted from Stoy 74: these are discussed in section 6.4.

The meaning function, then, is a way of mapping the object language to lambda expressions which, in turn, denote functions. For an infinite language (i.e. one with a recursive syntax) this function can obviously not be given extensionally. It is, indeed, very important to construct the meaning function in a way which associates (functional) denotations with parts of the constructs of the object language. Furthermore, the meaning of compound constructs of the object language should depend only on the meaning of the components of such compounds. This approach provides a natural way of categorising the meta-language constructs used in defining the meaning functions. Those parts of the meta-language discussed in section 6.2 can be viewed as a macro-expansion scheme for (abstract) programs of the object language. What is created from this expansion is an expression in an enriched ("sugared") lambda notation. Section 6.3 explains most of the extra notation which is used to aid readability in terms of pure lambda notation plus two combinators which are discussed in section 6.4. The basic ideas of states and state transition functions are reviewed in the first subsection below.

The whole treatment of arbitrary order has been excised from this volume. This problem had been tackled in the framework of abstract interpreters (see Walk 69) and was recognised in Bekić 74. The general problem of how to define the merging of co-operating processes using a denotational sematics approach is, however, very complex and still to some extent "open". In view of the aim of this volume to present a safe basis for definition work it was decided that the problems involved in merging should be dropped. There is some discussion of apparently unordered constructs in section 6.5.

6.1 States and State Transitions

Most language definitions use a basic semantic object which is a state. Such a state is a mapping from some class of names or their surrogates (e.g. locations) to whatever values can be manipulated. Such a state provides the vehicle for defining the denotations of names. Constructs which have an assignment nature (nearly all interesting languages contain some) will naturally be granted denotations which are functions from states to states. Such functions are usually called state transitions. Thus for some class of constructs Θ:

$$m\text{-}\Theta\colon\ \Theta \to Tr$$
$$Tr = \Sigma \overset{\sim}{\to} \Sigma$$

The notation specific to states and transitions can now be introduced.

It will normally be necessary to have some structure within states (e.g. one component for storage, one for files). This is reflected by introducing a class of elementary objects called references. References can be explicitly enumerated by writing them as strings of underlined upper-case characters. The only operators defined over references are the tests for (in)equality. A state will be a mapping from such references to any sort of objects, thus:

$$\Sigma = REF \underset{m}{\rightarrow} OBJECT$$

The basic way of defining a new state is by specifying a difference from an existing state. A state transition is specified by writing a meta-language assignment statement which uses the reference (which is to have a different value) on the left and a value on the right. Thus the meta-language assignment defines a transition:

$$(r := v): \Sigma \overset{\sim}{\rightarrow} \Sigma$$

and is defined as:

$$r := v \overset{\Delta}{=} \lambda\sigma.\sigma+[r\mapsto v]$$

Unlike most programming languages, a reference always denotes itself (rather than its value) even if it is written on the right-hand-side of an assignment. If the value is required the contents operator "$\underline{c}$" must be used (see section 6.3 for further details).

If a semantic function maps elements of some class Θ into transformations, the type will be specified as:

$$m\text{-}\theta: \Theta \Rightarrow$$

with the meaning:

$$m\text{-}\theta: \Theta \rightarrow (\Sigma \overset{\sim}{\rightarrow} \Sigma)$$

Further, with value returning transformations (see below):

$$m\text{-}\theta: \Theta \Rightarrow R$$

stands for:

$$m\text{-}\theta: \Theta \rightarrow (\Sigma \overset{\sim}{\rightarrow} \Sigma\ R)$$

Having introduced the ideas of states and state transitions, it would be possible to build up ways of describing increasingly more complex transitions. This is not the approach taken here. Rather, transitions have been introduced only to provide motivation for the succeeding three sub-sections. The treatment is now top-down in that the means of creating extended expressions precedes the description of the extended expressions in terms of basic combinators, so that only in section 6.4 is the treatment back to the level of lambda expressions.

6.2 A Macro-expansion Process

The assignment concept in the meta-language together with the combinators to be introduced in subsequent sub-sections provide ways of writing complex tranformations. Given some object language it is not difficult to write for any particular program a corresponding meta-language expression which denotes the function which captures the meaning of that particular program. Since the task in hand is defining whole languages, a way must be provided for generating the corresponding meta-language expressions for any (well-formed) program. The generation itself, which is the subject of this sub-section, can best be understood as a macro-expansion process. By this is meant that the parts of the meta-language described in this section rely on the text and environment alone. This expansion is rather obvious and will be explained with a minimum of formalism.

In a definition there will be (notionally) for each defined class of the abstract syntax one semantic rule. This semantic rule maps objects of the class to their denotations. The most basic part of the macro-expansion process is the application of the relevant rules to the text components. The qualification "notionally" has been added above for two reasons. Partly, it would be unfortunate to apply any rule which dictated a structure on the semantic functions. Thus very short semantic functions may be better conbined with that corresponding to the syntax class which uses them. More importantly, syntax rules which simply list options would have corresponding semantic rules which were simply case distinctions selecting other semantic rules. These "splitting" (or " routing " rules) are omitted and only syntax rules which define constructors (i.e. "*::rules*") will normally have corresponding semantic rules.

The syntax for an infinite language will be recursive. In consequence the corresponding semantic rules must also be a family of mutually recursive functions. However, it was pointed out in the discussion of objects that valid tree objects will always be finite. This guarantees that the macro-expansion process envisaged will terminate.

A trivial language can be used to illustrate this basic expansion. The language to be considered is that of binary numerals. It is, of course, so simple that the corresponding denotations are not transformations but the natural numbers. The syntax of the language is:

$$\begin{aligned} &\mathit{Bin\text{-}digit} &&= \underline{0}\,|\,\underline{1} \\ &\mathit{Bin\text{-}numeral} &&= [\mathit{Bin\text{-}numeral}]\ \mathit{Bin\text{-}digit} \end{aligned}$$

Given the choice of denotations semantic functions are required of types:

$$\begin{aligned} &m\text{-}d\text{: } \mathit{Bin\text{-}digit} &&\rightarrow \mathit{Nat}^0 \\ &m\text{-}n\text{: } \mathit{Bin\text{-}numeral} &&\rightarrow \mathit{Nat}^0 \end{aligned}$$

Appropriate functions might be:

$$\begin{aligned} &m\text{-}d(d) = \underline{\mathit{if}}\ d{=}\underline{0}\ \underline{\mathit{then}}\ 0\ \underline{\mathit{else}}\ 1 \\ &m\text{-}n(mk\text{-}bin\text{-}numeral(n,d)) = \underline{\mathit{if}}\ n{=}\underline{NIL}\ \underline{\mathit{then}}\ m\text{-}d(d) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \underline{\mathit{else}}\ 2.m\text{-}n(n){+}m\text{-}d(d) \end{aligned}$$

For any element of the object language (*Bin-numeral*) the meaning function will create a finite expression. In this simple language the expression will contain only arithmetic objects and it can be simplified by the laws of arithmetic to a natural number (the required denotation). A number of points have, however, been illustrated. It was observed in section 3.5 that objects can be decomposed by writing the constructor "on the left of a *let*": the use of the constructor in the parameter position of *m-n* is a further application of this idea and is equivalent to:

$$\begin{aligned} &m\text{-}n(nm) = (\underline{\mathit{let}}\ mk\text{-}bin\text{-}numeral(n,d) = nm \\ &\qquad\qquad\quad \ldots) \end{aligned}$$

Furthermore the conditional statement used in the definition of *m-n* is entirely dependent on the text being analyzed and is a part of the macro-expansion process.

Of the three forms of conditional expression in the meta-language, *case* is always dependant on the text alone, "McCarthy" conditionals are always dependant on dynamic values and *if then else* can be used in either way. The tests which depend on dynamic values are discussed in sub-section 6.3. The meaning of the text-dependant *if then else* should be obvious: either one or the other expansion is chosen depending on the result of the test. The need for the *case* construct has been reduced somewhat by the adoption, in large definitions, of the practice of omitting semantic rúles for syntax classes which are lists of options. But an artificial example can be constructed for the syntax:

```
A  = B|C|D

B :: X
C :: X Y
D :: X Y Z
```

The definition:

```
fn-a(a) = cases a:
              mk-b(x)      → f(x)
              mk-c(x,y)    → g(x,y)
              mk-d(x,y,z)  → h(x,y,z)
```

has the meaning:

```
fn-a(a) =
   if (∃x)(mk-b(x)=a) then (let mk-b(x)=a in f(x))
   else if (∃x,y)(mk-c(x,y)=a) then (let mk-c(x,y)=a in g(x,y))
   else if (∃x,y,z)(mk-d(x,y,z)=a) then (let mk-d(x,y,z)=a in h(x,y,z))
```

A case clause with a final "*T*→" clause will use this last option if none of the preceding predicates were true. If no such clause is present it is an error for none of the predicates to be satisfied.

A similar static expansion from the text is given by the *for* construct. If a textual object which is a list is to be given a meaning, the semantic rule is likely to be formed from a *for* construct, thus:

```
for i=l to u do f(list(i))
```

this can be statically expanded into:

$f(list(l));f(list(l+1));\ldots;f(list(u))$

where the "$;$" combinator between transformations is that explained in section 6.3. Similarly a value returning transformation can be used to define a list to be created from the list in the text by:

$\underline{let}\ vl\colon <vt(list(i))\,|\,l\leq i\leq u>$

with the meaning (again expressed in terms of combinators to be defined in the next sub-section):

$$\begin{array}{lll}
\underline{let}\ vll & : & vt(list(l)); \\
\underline{let}\ vlsl & : & vt(list(l+1)); \\
\ldots & & \\
\underline{let}\ vlu & : & vt(list(u)); \\
\underline{let}\ vl & = & <vll,vlsl,\ldots,vlu>
\end{array}$$

Definitions of programming languages have universally adopted the concept of an environment to handle the problems related to block structure. Construction of explicit environments is achieved by standard use of the meta-language; the rôle of the explicit environment in the macro-expansion process is discussed at the end of this sub-section. There is, however, another topic to be discussed first and that is the exit mechanism of the meta-language.

The exit-mechanism is discussed in detail elsewhere in this volume (Jones 78b) and a step-by-step motivation of the approach has been presented in Jones 75. Here, only a brief review of the idea is given. For a simple language (i.e. one without constructs which cause abnormal termination) the transformations to be used as denotations will be:

$T = \Sigma \overset{\sim}{\rightarrow} \Sigma$

The exit approach to languages which permit abnormal termination is to use denotations which are transformations of the type:

$E = \Sigma \overset{\sim}{\rightarrow} \Sigma\ [Abn]$

Here, just as Σ is chosen to fit the particular language, the class

Abn is a set of objects which can be used for the language in question to indicate what sort of abnormal termination has occured. Of importance for the meta-language is only the distinction between NIL (*Abn* omitted) and an actual *Abn* value. A particular element of E which takes an element of Σ into a pair which contains a (possibly different) element of Σ and NIL is a "normal" transition. In other words no abnormal situation is to be resolved. On the other hand, if the second element of the output pair is not NIL, it provides not only the knowledge that an abnormal situation is to be resolved, but also some information which aids resolution. In a simple language with goto statements, for example, the set *Abn* might be identical with label identifiers. An unresolved goto is then denoted by an element of E which yields the label as the second component of the result. It is "resolution" which is of concern in this section because it employs a sort of implicit environment. The *exit* and normal successor (i.e. ";") combinators are defined in the next sub-section; the technique for resolving exits plays a part in the macro-expansion scheme. Firstly, however, the definitions are provided.

The basic construct to be used is written:

$$\underline{tixe}\ [a \mapsto t_1(a) \mid p(a)]\ \underline{in}\ t_2$$

Intuitively this requires that the basic tranformation (in E) to be used is t_2; if for a particular $\sigma \in \Sigma$ this results in a NIL *Abn* component then that element of t_2 is also an element of the overall transformation; if, however, an abnormal component is returned which satisfies the predicate p then transformation t_1 is used to attempt to resolve the abnormality; if an abnormal result is delivered by t_2, but this value does not satisfy p, then the same abnormal result is delivered by the overall construct. To define this formally the types are given first, suppose:

$$\begin{aligned} t_1 &: Abn \rightarrow E \\ t_2 &: E \\ p &: [Abn] \rightarrow Bool \\ E &= \Sigma \overset{\sim}{\rightarrow} \Sigma\ [Abn] \end{aligned}$$

then the type of the construct is:

$$(\underline{tixe}\ [a \mapsto t_1(a) \mid p(a)]\ \underline{in}\ t_2) : E$$

and the definition is:

$$\begin{array}{l}
(\underline{tixe}\ [a\mapsto t_1(a)\,|\,p(a)]\ \underline{in}\ t_2) \\
\quad \triangleq\ (\underline{let}\ e = [a\mapsto t_1(a)\,|\,p(a)] \\
\qquad \underline{let}\ r(\sigma,a) = (a\in\underline{dom}e\to r^{\bullet}e(a)(\sigma), T\to<\sigma,a>) \\
\qquad r^{\bullet}t_2)
\end{array}$$

Notice that r in the above expansion is used recursively. Thus if a-nother abnormal situation results which still satisfies p, the effect defined in e will again be used. This recursive form fits the problem of goto handling (see the definition in appendix III) very well. In earlier definitions a non-recursive "*trap*" was used with which it was then necessary to construct the required recursion in the semantic functions. For:

$$\begin{array}{l}
t_1 : Abn \to E \\
t_2 : E
\end{array}$$

the type of the *trap* construct is:

$$(\underline{trap}(a)\ \underline{with}\ t_1(a);t_2): E$$

and the definition is:

$$\begin{array}{l}
\underline{trap}(a)\ \underline{with}\ t_1(a);t_2 \\
\quad \triangleq\ (\underline{let}\ h(\sigma,a) = (a\neq\underline{NIL}\to t_1(a)(\sigma), T\to<\sigma,\underline{NIL}>) \\
\qquad h^{\bullet}t_2)
\end{array}$$

Returning to the "*tixe*" construct, it is necessary to explain its role in the macro-expansion. Firstly, it could be argued that, for a concept which is used very few times in a definition (cf. Henhapl 78 in this volume), it is not worth providing a special meta-language construct. The justification for so doing is precisely to provide a framework for the resolution of exits which does fit with a macro-expansion view of the text. The key point is the use of a mapping in the *tixe* construct. The finiteness criteria of mappings is met because, for finite texts, the predicate p must yield a finite number of abnormal conditions to be resolved. In order for the overall *tixe* construct to yield a denotation for a text, both the t_1 and t_2 semantic functions should only be used on sub-parts of the overall text being defined. Referring to the semantic function "*i-named-stmt-list*" in the definition in appendix III, it will be seen that the rôle of the *tixe* construct is essentially to generate a static environment the elements of which are mutually dependant on one another. (The discussion of this sort of recursion is given under the general topic of environments).

The remaining topic to be explained with regard to the macro-expansion is the rôle of the environment object (*Env*). In a language without block structure it would be possible to write semantic functions for each class (Θ) of abstract syntax objects.

$$m\text{-}\theta\colon\ \Theta \rightarrow (\Sigma \overset{\sim}{\rightarrow} \Sigma)$$

In a language which permits redefinition of identifiers at different levels of the block structure, an environment is introduced, so that most semantic functions are of type:

$$m\text{-}\theta\colon\ \Theta\ Env \rightarrow (\Sigma \overset{\sim}{\rightarrow} \Sigma)$$

The highest (program) construct of the language has a corresponding semantic rule which creates an empty environment and this semantic rule is thus still of type:

$$m\text{-}program\colon\ Program \rightarrow (\Sigma \overset{\sim}{\rightarrow} \Sigma)$$

If the language to be defined permits recursion (e.g. recursive procedures in Algol 60) the equations for creating environments will themselves be recursive. This recursion must, of course, be understood for the whole definition to be sound.

If the definition were in fact to be treated as though it were an abstract interpreter, it would be fairly easy to clarify the recursive equations on environments by adopting an appropriate mode for parameter passing. One way of ensuring that a denotational view was sound would be to set up an ordering over environments. Such an ordering could be defined pointwise over the transformations to be held as values in the environments. Here an alternative view is taken. Essentially the environment is viewed as a way of avoiding a name creation process for the denotations to be stored therein. The motivation for introducing this "linguistic level" can best be given by, initially, restricting the discussion to a language which only has local variables (i.e. no procedures). For any particular program in the language, it is easy to write an associated extended lambda expression. Thus:

```
begin integer a,b; ... end
```

might have a denotation like:

$(\underline{let}\ l_a{:}get\text{-}loc(a);\ \underline{let}\ l_b{:}get\text{-}loc(b);\ \ldots)$

As has been stated repeatedly, the problem that must be solved is to provide a way of generating denotations for any program. The possibility of using an elipses notation was excluded, but the environment can be seen precisely as a way of taking away the need for such a notation for the creation of names for the location. Thus for a program:

```
begin integer c,d; ... c:=d+c; ... end
```

the environment might be such that the denotation of the body is created by:

$i\text{-}body((\ldots c{:=}d{+}c;\ldots),\ [c\mapsto l_c, d\mapsto l_d])$

The definition of *i-body* will be such that this expands to:

```
...
let vd : contents(l_d);
let vc : contents(l_c);
assign (l_c,(vd+vc));
...
```

In other words, the environment itself disappears entirely during the macro-expansion process. The functions defining the meaning of blocks will create new locations for locally declared identifiers. For block structured languages the outer environment is updated with a pairing of the local names with the new locations. This modified environment has two uses: for local uses of identifiers the location is obtained from the environment; for nested blocks it is the basis for generating further environments. In order to exhibit the denoted expression, it is necessary to create (arbitrary) names for the locations.

It is now necessary to show how such an explanation can be extended to cover the more interesting case of languages which permit recursive procedures. For a program:

```
begin p1:proc ... p2 ...;
      p2:proc ... p1 ... p2 ...;
      ...
      p1
      ...
end
```

the definition will generate a denotation for the block via:

```
let nenv = env+[p1↦e-proc-den((...p1...),nenv),
                p2↦e-proc-den((...p1...p2...),nenv)]
i-body((...p1...),nenv)
```

Introducing names for the procedure denotations, this expands to:

```
let pden1 = ... pden2 ...
let pden2 = ... pden1 ... pden2 ...
... pden1 ...
```

This time the disappearance of *nenv* is doubly welcome: with it has gone the recursion whose least fixed point could only be discussed in terms of an ordering. A different recursion (that in terms of procedure denotations) has become visible but for one thing it was there anyway and for another it is precisely the recursion over transformations which has to be explained below.

This section, then, has shown how a definition can be viewed as a series of macros which expand objects of the language to be defined into expressions in an extended lambda notation. At the expense of introducing the concept of names for locations and procedure denotations, the explanation of these extended expressions below will not be burdened with the concept of environments.

6.3 Simplifying Extended Expressions

The previous sub-section has shown how a definition of the type given in appendix III can be used to create (by macro-expansion) an expression which is the denotation of the given object language text. This expression itself denotes a transformation. However, the expression is not yet in pure lambda notation and the purpose of this sub-section will be to explain those combinators which can be regarded simply as "syntactic sugar" for making the generated result more readable.

It will be easier to comprehend the full combinator definitions if the idea is first explained with a simplified problem. Suppose the denotation of some compound object:

$a \in A$ where
$A = B\ C$

is sought. Given two functions:

$$m\text{-}b : B \rightarrow (\Sigma \tilde{\rightarrow} \Sigma)$$
$$m\text{-}c : C \rightarrow (\Sigma \tilde{\rightarrow} \Sigma)$$

there are many possible ways of combining them to create a transformation. Given the objective that the denotations of compound objects should depend only on the denotations of their components, one of the most pleasing combinations is a composition of the two created functions. Thus:

$$\begin{aligned} m\text{-}a(mk\text{-}a(b,c)) &= m\text{-}c(c) \cdot m\text{-}b(b) \\ &= \lambda\sigma.(m\text{-}c(c)(m\text{-}b(b)(\sigma))) \end{aligned}$$

One objective of introducing combinators (like composition) is to avoid having to write out many "λ"'s and "σ"'s. In addition to a "σ-*free*" style, a much more natural definition can be achieved by the use of well chosen combinators. In this case we can define a "semicolon" combinator:

for: $t_1 : \Sigma \tilde{\rightarrow} \Sigma$
$t_2 : \Sigma \tilde{\rightarrow} \Sigma$

the type is: $(t_1;t_2) : \Sigma \tilde{\rightarrow} \Sigma$

and the definition: $t_1;t_2 \triangleq t_2 \cdot t_1$

and then rewrite the above example:

$$\begin{aligned} m\text{-}a(mk\text{-}a(b,c)) = {} & m\text{-}b(b); \\ & m\text{-}c(c) \end{aligned}$$

In a simple language definition, the meaning of a sequence of statements is likely to be the composition of the meanings of the single statements. Since, in the concrete syntax of such an object language, the elements of the sequence are likely to be separated by "*;*", a pleasing symmetry has been created. Notice, however, that the meta-language semicolon operator has been formally defined and there is no element of circularity in such a link between object and meta-language. It would be useful to define a few other combinators (e.g. *if*, *while*) on these simple transformations. This is not done here since the languages which are of general interest require slightly more complex denotations. Since the combinators given below can be specialized very easily to those for the simple transformations, attention is now turned to the more general versions.

The object language features which demand richer denotations are those which manifest an "abnormal *exit*" behaviour. The archetypal statement is "*goto*", but also much error handling can best be dealt with in the same way. One approach to this problem is to use transformations which create states and optional abnormal indications.

The type of a transformation now becomes:

$$E = \Sigma \overset{\sim}{\to} \Sigma\ [Abn]$$

where Abn is chosen for a particular definition. The intuitive meaning of such a (partial) function is that it maps state values into pairs whose first element is always a (possibly different) state value; in the case that the computation was "normal", the second component will be the elementary object $\underline{NIL}$; if termination was abnormal, the second component will be other than $\underline{NIL}$ and its actual value will provide information about the encountered exception.

The notation introduced for type clauses must now be extended. Rather than write:

$$m\text{-}\theta\colon\ \Theta\ Env \to (\Sigma \overset{\sim}{\to} \Sigma\ [Abn])$$

such function types will be written:

$$m\text{-}\theta\colon\ \Theta\ Env \Rightarrow$$

and for value returning transformations:

$$m\text{-}\theta: \; \Theta \; Env \rightarrow (\Sigma \xrightarrow{\sim} \Sigma \; [Abn] \; Val)$$

is written:

$$m\text{-}\theta: \; \Theta \; Env \Longrightarrow Val$$

It is now necessary to define the combinators which are used to combine transformations of type E. Firstly if a simple transformation

$$t \; : \; \Sigma \xrightarrow{\sim} \Sigma$$

is written in a context where a transformation of type E is required, it is interpreted as:

$$t \triangleq \lambda\sigma.<t(\sigma),\underline{NIL}>$$

that is, viewing

$$t \; : \; E$$

it is ensured that it always has a normal termination. Notice, in particular "I" the identity over Σ gives:

$$I \triangleq \lambda\sigma.<\sigma,\underline{NIL}>$$

The full semicolon combinator can now be introduced. Intuitively its purpose is to avoid applying the second transformation if the first results in abnormal termination. Formally, given transformations with types:

$$t_1 \; : \; E$$
$$t_2 \; : \; E$$

then the type of the result is:

$$(t_1;t_2) \; : \; E$$

and its definition:

$$(t_1;t_2) \triangleq (\lambda\sigma,a.a=\underline{NIL}\rightarrow t_2(\sigma),T\rightarrow<\sigma,a>)^{\bullet} t_1$$

In order to signal, with some given value v (in Abn), that an abnormal *exit* is to occur, the *exit* combinator is used, its type is:

$$(\underline{exit}(v)) : E$$

and its definition:

$$(\underline{exit}(v)) \triangleq \lambda\sigma.<\sigma,v>$$

The *error* construct of the meta-language is used to indicate that no defined result is given. Thus the value of *error* anywhere in the denotation is to show an abnormal result for the whole text. To achieve this, the *exit* can be used with the rule that no *tixe* covers the *ERROR* case; thus:

$$\underline{error} \triangleq \underline{exit}\ (\underline{ERROR})$$

The *tixe* combinator, which was explained as part of the macro-scheme, and the semicolon combinator have provided two ways of conditionally applying subsequent transformations. A combinator which causes application of a second transformation in both normal and abnormal cases is "*always*". Given:

$$t : \Sigma \tilde{\rightarrow} \Sigma$$
$$e : E$$

then the type is:

$$(\underline{always}\ t\ \underline{in}\ e) : E$$

and the definition:

$$(\underline{always}\ t\ \underline{in}\ e)$$
$$\triangleq (\lambda\sigma,a.<t(\sigma),a>)^{\bullet}e$$

If t is, in fact, of type E then it is an *error* if a non-*NIL* second component is ever returned.

(There are occasions where an *exit* or *error* construct is required also with pure functions. It is straightforward to redefine composition in a way analogous to the semicolon combinator. Having done so, of course, semicolon is defined in terms of (the revised) composition).

This concludes the combinators which are especially designed for building up expressions from (expressions denoting) transformations of type E. Attention is now turned to value returning transformations of type:

$$R = \Sigma \tilde{\to} \Sigma\ [Abn]\ V$$

The most basic way of generating such a transformation is by the *return* combinator. Thus:

$$\textit{for } v \in V,\ (\underline{return}(v)) : R$$

is defined:

$$(\underline{return}(v)) \triangleq \lambda\sigma.\langle\sigma,\underline{NIL},v\rangle$$

A transformation of type R may be placed after one of type E (separated by "$;$") and make the whole of type R. Thus given:

$$t : E$$
$$r : R$$

then:

$$(t;r) : R$$

is defined:

$$(t;r) \triangleq (\lambda\sigma,a.(a=\underline{NIL}\to r(\sigma),T\to\langle\sigma,a,\underline{?}\rangle))^{\bullet}t$$

Notice that as a consequence of this definition abnormal *exit* results in returning an undefined (here $\underline{?}$) result.

The value generated by a value returning transformation can be used in a following transformation in a way which is analogous to the simple *let* shown below. Given:

$$r : R$$
$$t : V \to E$$

then:

$$(\underline{let}\ v{:}r;t(v)) : E$$

is defined:

$$(\underline{let}\ v{:}r;t(v)) \triangleq (\lambda\sigma,a,v.(a{=}\underline{NIL}\rightarrow t(v)(\sigma),T\rightarrow<\sigma,a>))^{\bullet}r$$

To reduce the number of separate cases to be distinguished, the contents operator ($\underline{c}$) is considered to be a value returning transformation. Thus:

$$\underline{c} : REF \rightarrow R$$

and:

$$\underline{c}r \triangleq \lambda\sigma.<\sigma,\sigma(r)>$$

The contents operator, or any other value returning transformation,can occur in contexts where the construct is defined only for values. The meaning is:

$$\ldots\ r\ \ldots \triangleq (\underline{let}\ v{:}r;\ \ldots\ v\ \ldots)$$

In the case of the *while* combinator this rule must be interpreted as:

$$(\underline{while}\ r\ \underline{do}\ t) \triangleq (\underline{let}\ w = (\underline{let}\ v{:}r;\ \underline{if}\ v\ \underline{then}\ (t;w)\ \underline{else}\ \mathrm{I})\ \underline{in}\ w)$$

The recursive *let* on the right hand side defines that w should be the minimal fixed-point of the equation.

The remaining items of syntactic sugar to be defined are more basic (i.e. are not specialized to particular sorts of transformations.) In its simplest form *let* provides an abbreviation:

$$(\underline{let}\ v = e1\ \underline{in}\ e2(v)) \triangleq (\lambda v.e2(v))(e1)$$

The recursive form of *let*:

$$(\underline{let}\ v = e1\ \underline{in}\ e2(v)) \triangleq (\lambda v'.e2(v'))(Y\lambda v.e1(v))$$

defines v to be be the minimal fixed point of expression $e1$.

Functional composition is straightforward:

$(f^{\bullet}g) = (\lambda x.f(g(x)))$

The dynamic conditional is defined in terms of a combinator discussed in the next section:

$(\underline{if}\ v\ \underline{then}\ t_1\ \underline{else}\ t_2) \triangleq cond(t_1,t_2)(v)$

6.4 Basic Combinators

A definition as viewed in section 6.2 is a way of generating, for any object text whose meaning is sought, an expression in an extended lambda notation. The length of such expressions is reduced and the readability much increased by the use of various combinators. Section 6.3 has shown how these can be eliminated in a way which yields a much longer expression whose structure corresponds much less closely to that of the original text. The advantage of this expression is however, that it is almost in pure lambda notation and it is now only necessary to discuss the meaning of two remaining combinators.

In fact all that is done at this point is to rely on the work of others. Thus along with the models of the lambda calculus in Stoy 74, the minimal fixed point operator (Y) and the "doubly strict" conditional are adopted:

$$\begin{aligned}
cond(t_1,t_2)(\top) &= \top\\
cond(t_1,t_2)(\underline{TRUE}) &= t_1\\
cond(t_1,t_2)(\underline{FALSE}) &= t_2.\\
cond(t_1,t_2)(\bot) &= \bot
\end{aligned}$$

The adequacy of the doubly strict functions comes from the explicit treatment of errors in the meta-language.

6.5 Other Issues

The question of how to define arbitrary order of evaluation in a language has been omitted for reasons explained elsewhere. There are, however, some points where it appears to occur in the definitions given in this volume. Consider the use of:

$\underline{let}\ l \in Sc\text{-}loc\ \underline{be}\ \underline{s.t.}\ l \notin \underline{dom}\ \underline{c}\ \underline{R\text{-}STG}$
$\underline{R\text{-}STG} := \underline{c}\ \underline{R\text{-}STG} \cup [l \mapsto \underline{?}]$

Clearly, this intends to show a freedom of choice which would be lost by pre-defining an order over elements of *Sc-loc* and always choosing the "next" free location. Equally clearly, it would be unnecessarily complex to use some general treatment for non-determinism (e.g. all functions are $\Sigma \to \Sigma\text{-}set$) in order to provide a formal definition of this construct. Essentially, it is clear that the particular choice makes no difference and it is no more worthwhile to prove this claim than it is to over-define (see Jones 77c) and then prove what properties of the definition are irrelevant.

It must, however, be clear that the "$\underline{let}\ \underline{be}\ \underline{s.t.}$" construct could be used unwisely and each use should really be accompanied by its own argument of well-foundedness.

Similarly, the simple rule that value returning transformations can be written in positions where values are required presents a danger. The expansion given in section 6.3 that they should be extracted and formed into a preceding $\underline{let}$ is only clear if either there is only one such value returning transformation or if their order will have no effect on the result. This latter is the case in:

$\underline{let}\ nenv : env + ([id \mapsto e\text{-}type(dclm(id), env) \mid id \in \underline{dom} dclm] \cup \ldots)$

Finally the topic of errors in the definition should be mentioned. At a number of points it has been indicated that something is simply considered to be a wrong use of the meta-language (e.g. an incomplete case construct). In such a case, just as with:

$(\iota i)(7 < i < 3)$ or $\sqrt{'ABC'}$

nothing is defined. This is in distinction to the use of $\underline{error}$ which shows that the object text is defined by the definition to be in *error*. In this case it is permitted for a (valid) implementation to produce any result (although a diagnostic is, of course, the most useful result!)

ACKNOWLEDGEMENTS

The basis of the work on formal definition was laid in the Vienna Laboratory during the 1960's; Bekic 70a already showed a dissatisfaction with the "abstract interpreter" approach; the actual meta-language described in this paper was developed at that Laboratory during the period 1973/75. This author would like to acknowledge the stimulus and enjoyment he has derived from his years in Vienna. In particular the co-operation with Hans Bekić both in writing the notation section of Bekić 74 and in discussing a draft of this paper, have been invaluable.

APPENDIX I: Concrete Syntax of the Meta-language

The rules given below are an outline of a concrete syntax in the notation of Walk 69. Basically, the rules are context-free, but sub-classes of expressions are used in order to indicate the types of permitted arguments to operators.

Written definitions use a number of relaxations on this syntax which are not formally defined.

a) Brackets around blocks and *cond-stmts* as well as commas and "<u>*in*</u>" are omitted where indentation or line breaks make the result unambiguous.

b) Comments, enclosed in "/**/" may be used freely.

c) Where an expression occurs at the end of an *expr-ld*, "<u>*result*</u> <u>*is*</u>" can be used.

d) The order of precedence of operators (standard) is modified by use of blanks and line-breaks in order to avoid excessive bracketing.

defn	*::=*	*{·{fn-defn|tr-defn}...}*
tr-defn	*::*	*tr-id{(defs)}... = block*
tr-id		usually begins with "*i-*" or "*e-*"
defs	*::=*	*[,·def...]*
def	*::=*	*v-id|constructor*
construcor		"*mk-*" followed by name of abstract syntax class (possibly followed by *defs* in parenthesis)
block	*::=*	*([exit-spec][let-cl]{;·stmt...})*
exit-spec	*::=*	<u>*tixe*</u> *map-expr* <u>*in*</u>
let-cl	*::=*	<u>*let*</u> *def:expr;*
expr-block		as *block* but is value returning
stmt	*::=*	*stmt-block|cond-stmt|iter-stmt|stmt-ld|assign-stmt|int-stmt|return-stmt|*I*|exit-stmt|*<u>*error*</u>

```
cond-stmt   ::= if expr then stmt [else stmt]|
                ({,·{expr→stmt}...}[,T→stmt]|
                (cases expr: {,·{expr→stmt}...}[,T→stmt])

iter-stmt   ::= for v-id=expr to expr do stmt |
                while expr do stmt

stmt-ld     ::= (ldl;stmt)
assign-stmt ::= v-id:=expr
int-stmt    ::= tr-id{(args)}...
return-stmt ::= return(expr)
exit-stmt   ::= exit(args)

fn-defn     ::= fn-id(defs) = pure-expr

expr        ::= prefix-expr|infix-expr|quant-expr|const|fn-ref|eval-stmt
                var-ref|(expr)|cond-expr|expr-block|expr-ld

fn-ref      ::= fn-id(args)
eval-stmt   ::= eval-id{(args)} ...
args        ::= [,·expr···]

var-ref     ::= [c] v-id[(args)]
cond-expr   ::= if expr then expr else expr |
                ({,·{expr→expr}...}[,T→expr])|
                (cases expr:{,·{expr→expr}...}[,T→expr])
expr-ld     ::= (ldl;expr)
ldl         ::= {;·ld...}
ld          ::= let {def|fn-id(defs)} = expr in

pure-expr       as expr but does not (directly) contain:
                expr-block, c, eval-stmt
```

<u>general expressions</u>

```
prefix-expr ::= hd tuple-expr
descr-expr  ::  (ιdef[∈set-expr])(log-expr)
```

see also selectors and constructors of abstract syntax.

arith-expr

```
    prefix-expr ::= len tuple-expr|card set-expr
```

see also operators on any standard sets.

log-expr

```
    prefix-expr ::= ¬log-expr
    infix-expr  ::= log-expr {&|∨|⇒|⇔}log-expr|
                    expr∈set-expr|
                    set-expr {⊂|⊆}set-expr
                    expr {=|≠}expr

    quant-expr  ::= ({∀|∃|∃!}def[∈set-expr])(log-expr)

    const       ::= TRUE|FALSE
```

see also operators (e.g. relational) on standard sets and "*is-*" prefixed to abstract syntax class names.

set-expr

```
    prefix-expr ::= {union|power}set-expr|
                    {dom|rng}map-expr|
                    {inds|elems}tuple-expr

    infix-expr  ::= set-expr{∪|∩|-}set-expr

    const       ::= {[,·expr···]}|
                    {expr|log-expr}|
                    {arith-expr:arith-expr}
```

see also standard set names: *Bool*, *Nat*, *Nat*$^{\bullet}$, *Int*

<u>tuple-expr</u>

```
prefix-expr ::= {tl|conc}tuple-expr

infix-expr  ::= tuple-expr^tuple-expr

const       ::= <[,·expr···]>|
                <tuple-expr⊥log-expr>
```

<u>map-expr</u>

```
infix-expr  ::= map-expr{+|∪|°}map-expr
                map-expr{\|⊥}set-expr

const       ::= [{,·{expr→expr}···}]|
                [expr→expr⊥log-expr]
```

APPENDIX II: Conventions

Apart from notation which is strictly part of the described meta-language, a number of conventions have developed in its use. While changes to the notation should be reflected in a revised definition of the meta-language, the conventions can be ignored with impunity.

This appendix lists some possible naming conventions:

usage	example	convention
keyword	<u>*for*</u>	underlined sequence of lower case (*l.c.*)
operators	<u>*dom*</u>	underlined sequence of lower case (*l.c.*)
variables	*val*	sequence of *l.c.*
of list type	*intl*	last letter "*l*"
of set type	*cas*	last letter "*s*"
of map type	*dclm*	last letter "*m*"
class names	*Program*	first *u.c.*, rest *l.c.*
elementary objects	<u>*INT*</u>	underlined sequence of *u.c.*
references	<u>*R-STG*</u>	first letter "*R*"
context conditions	*is-wf-expr*	predicate names begins "*is-wf-*"
selector names	*s-bdl*	begins "*s-*"

APPENDIX III: Example Definition

This appendix presents an example whose study will considerably increase the reader's understanding of the meta-language. The definition itself is divided into abstract syntax, context conditions, semantic objects and semantic function parts. Before these are presented a few points of explanation will be offered. (In addition Bjørner 78b has taken examples from this definition and thus provided a useful introduction).

The abstract syntax should present no difficulties.

The context conditions are also straightforward, but note that in addition to the convention for dropping formulae corresponding to "splitting rules", an obvious extension permits equations for rules of the form:

A :: B C D

to be omitted if they are of exactly the form:

```
is-wf-a(mk-a(b,c,d),env) =
   is-wf-b(b,env) & is-wf-c(c,env) & is-wf-d(d,env)
```

Notice that the context condition *is-wf-block* shows that recursive calls of procedures are allowed while references to locally declared identifiers within the definitions of variables (i.e. bound lists) are prohibited. It can also be observed that *is-wf-for* shows that the control variable of a *for* construct is strictly local to the body of the *for*.

The description of the semantic objects yields most insight into the language. The state is divided into three components of which the most important is storage; this is defined as a disjoint union of two types of mapping. The environment auxiliary object provides denotations for the identifiers. In the simple case for variables, these denotations are locations; for arrays the locations are themselves (one-one) mappings. For control variable identifiers the denotation is simply an integer because no assignment to such variable is allowed (cf. *i-for*, *e-con-var-ref*). Procedure denotations are functions from argument lists (and *Aid-set*) to transformations (cf. *e-proc-den*, which is a function rather than a transformation, and *i-call*). The basic model for goto definition is discussed in Jones 78b. Here, because procedures can be passed as arguments, there is an additional problem of locating the proper generation of a label. This problem is handled by activation identifiers. For a more detailed description of this approach see Jones 75.

Given an understanding of the semantic objects an examination of the "type" clauses of the semantic functions should give an overview of the whole definition. Notice that *i-program* is the only function which does not create a (value returning) transformation. The function *i-block* defines *nenv* recursively to create the appropriate procedure denotations (see section 6.2). The handling of exits is simplified in this language (cf. *i-named-stmt-list*) since no compound statement is available.

ABSTRACT SYNTAX

```
Program     :: Stmt
Stmt        =  Block|If|For|Call|Goto|Assign|In|Out|NULL
Block       :: s-dcls:(Id→Type) Proc-set Named-stmt*
Type        :: s-sc-type: Scalar-type s-bds:[(Expr|*)+]
Proc        :: s-nm: Id Parm* Stmt
Parm        :: s-id: Id s-attr:(Type|PROC)
Named-stmt  :: s-nm:[Id] s-stmt: Stmt
If          :: s-b: Expr s-th: Stmt s-el: Stmt
For         :: s-con-var: Id s-init: Expr s-step: Expr s-limit: Expr Stmt
Call        :: s-pn: Id s-args:(Var-ref|Id)*
Goto        :: Id
Assign      :: s-lhs: Var-ref s-rhs: Expr
In          :: Var-ref
Out         :: Expr
Expr        =  Infix-expr|Rhs-ref|Con-var-ref|Const
Infix-expr  :: Expr Op Expr
Rhs-ref     :: Var-ref
Var-ref     :: Id s-sscs:[Expr+]
Con-var-ref :: Id
Const       =  Int-const|Bool-const                      disjoint
Op          =  Int-op|Bool-op|Comp-op                    disjoint
Scalar-type =  INT|BOOL
```

CONTEXT CONDITIONS

```
Env-static = Id → (Type | PROC | LABEL | CON-VAR)


is-wf-program(mk-program(t)) = is-wf-stmt(t,[])


is-wf-block(mk-block(dclm,procs,nsl),env) =
     let ll = <s-nm(nsl(i)) | 1≤i≤len nsl & is-id(s-nm(nsl(i)))>
     let pnms = {s-nm(p) | p∈procs}
     is-unique-ids(ll) &
     (p1∈procs & p2∈procs & p1≠p2 ⇒ s-nm(p1)≠s-nm(p2)) &
     is-disjoint(<dom dclm, pnms, elems ll>) &
    (let renv = env \ (dom dclm ∪ pnms ∪ elems ll)
     let nenv = renv ∪
                    [id ↦ star(dclm(id)) | id ∈ dom dclm] ∪
                    [id ↦ PROC | id ∈ pnms] ∪
                    [id ↦ LABEL | id ∈ elems ll]
     (dcl∈rng dclm ⇒ is-wf-type-dcl(dcl,renv)) &
     (pr∈procs ⇒ is-wf-proc(pr,nenv)) &
     (1≤i≤len nsl ⇒ is-wf-stmt(s-stmt(nsl(i)),nenv))
    )


is-wf-type-dcl(mk-type(sctp,bdl),env) =
     is-wf-type(mk-type(sctp,bdl),env) &
     (bdl=NIL v
      is-expr-list(bdl) & (1≤i≤lenbdl ⇒ ex-tp(bdl(i),env)=INT))


is-wf-proc(mk-proc(nm,parml,st),env) =
     is-unique-ids(<s-id(parml(i)) | 1≤i≤len parml>) &
     (let nenv = env +
                   [s-id(parml(i)) ↦ s-attr(parml(i)) | i∈{1:len parml}]
      is-wf-stmt(st,nenv) )
```

```
is-wf-parm(mk-parm(id,attr),env) =
    attr=PROC ∨ s-bds(attr)=NIL ∨ is-*-list(s-bds(attr))


is-wf-if(mk-if(b, , ),env) = ex-tp(b,env)=BOOL


is-wf-for(mk-for(id,init,step,limit,st),env) =
    let nenv = env + [id ↦ CON-VAR]
    ex-tp(init,env) = ex-tp(step,env) = ex-tp(limit,env) = INT &
    is-wf-stmt(st,nenv)


is-wf-call(mk-call(pn,al),env) =
    pn ∈ dom env & env(pn)=PROC &
    (1≤i≤len al ⇒ is-wf-var-ref(al(i),env) ∨ env(al(i))=PROC)


is-wf-goto(mk-goto(id),env) = id∈dom env & env(id)=LABEL


is-wf-assign(mk-assign(lhs,rhs),env) =
    is-scalar(lhs,env) &
    ex-tp(rhs,env) = var-ref-tp(lhs,env)


is-wf-in(mk-in(vr),env) = is-scalar(vr,env)


is-wf-infix-expr(mk-infix-expr(e1,op,e2),env) =
    is-int-op(op) & ex-tp(e1,env)=ex-tp(e2,env)=INT ∨
    is-bool-op(op) & ex-tp(e1,env)=ex-tp(e2,env)=BOOL ∨
    is-comp-op(op) & ex-tp(e1,env)=ex-tp(e2,env)=INT


is-wf-rhs-ref(mk-rhs-ref(r),env) = is-scalar(r,env)
```

```
is-wf-var-ref(mk-var-ref(id,el),env) =
    id∈dom env & is-type(env(id)) &
    (el=NIL v (s-bds(env(id))≠NIL &
               len el = len s-bds(env(id)) &
               (1≤i≤len el ⇒ ex-tp(el(i),env)=INT)))


is-scalar(mk-var-ref(id,sscl),env) =
    sscl=NIL ⇒ s-bds(env(id))=NIL


is-wf-con-var-ref(mk-con-var-ref(id),env) =
    id∈dom env & env(id)=CON-VAR


ex-tp(e,env) =
    cases e:
    mk-infix-expr(e1,op,e2) →
        (is-int-op(op) → INT
         is-bool-op(op) → BOOL
         is-comp-op(op) → BOOL)
    mk-rhs-ref(vr) → var-ref-tp(vr,env)
    mk-con-var-ref(id) → INT.
    T                 → (is-int-const(e) → INT
                         is-bool-const(e) → BOOL)


var-ref-tp(mk-var-ref(id, ),env) = s-sc-type(env(id))


is-unique-ids(idl) = /* true iff no duplicates */
type: Id* → Bool


star(t) = /* all bounds changed to * */
type: Type → Type


is-disjoint(sl) = /* tests sets pairwise disjoint */
type: Set* → Bool
```

is-scalar-type(t) = is-type(t) & s-bds(t)=<u>NIL</u>

is-array-type(t) = id-type(t) & s-bds(t)≠<u>NIL</u>

SEMANTIC OBJECTS

```
State       = (R-STG → Storage) U (R-IN → Value*) U (R-OUT → Value*)
Storage     = (Bool-loc → (Bool | ?)) U (Int-loc → (Int | ?))
Value       = Int | Bool
Env         = Id → (Loc | Con-var-den | Label-den | Proc-den)
Loc         = Array-loc | Scalar-loc
Array-loc   = (Int+ ↔ Bool-loc) | (Int+ ↔ Int-loc)
              constraint: l∈Array-loc ⇒ (∃il ∈ Nat+)(dom l =rect(il))
Scalar-loc  = Bool-loc | Int-loc
Con-var-den = Int
Label-den   :: Aid Id
Aid            is an infinite set
Proc-den    :: ((Loc | Proc-den)* Aid-set → Tr)
Tr          = State ⥲ State Abn
Abn         = [Label-den]
Int-loc   }
Bool-loc  }  infinite sets s.t. Bool-loc ∩ Int-loc ={}
```

FUNCTIONS

```
i-program(mk-program(t))(inl) =
   (let in-state =
              [R-STG↦[], R-IN↦inl, R-OUT↦<>]
    let fin-state = i-stmt(t,[],{})(in-state)
    fin-state(R-OUT)
   )
type: Program → (Value* → Value*)


i-stmt:  Stmt Env Aid-set ⇒
```

```
i-block(mk-block(dclm,procs,nsl),env,cas) =
   (let aid∈Aid be s.t. aid∉cas
    let nenv: env +
         ([id ↦ mk-label-den(aid,id) | is-cont(id,nsl)] ∪
          [id ↦ e-type(dclm(id),env) | id∈domdclm] ∪
          [s-nm(p) ↦ e-proc(p,nenv) | p∈procs]);
    always
         (let locs = {nenv(id) | id∈domdclm}
          let sclocs = {l∈locs | is-scalar-loc(l)} ∪
                          union{rngl | l∈locs & ¬is-scalar-loc(l)}
          R-STG := c R-STG\sclocs
         )
    in i-named-stmt-list(nsl,nenv,cas∪{aid},aid)
   )


e-type(mk-type(sctp,bdl),env) =
    if bdl=NIL then
         (let l∈Scalar-loc be s.t.
                                 is-tp-scalar-loc(sctp,l) & l∉dom c R-STG
          R-STG := c R-STG ∪ [l ↦ ?];
          return(l)
         )
    else
        (let ebdl: <e-expr(bdl(i),env) | 1≤i≤lenbdl>;
         if (∃i∈{1:lenbdl})(ebdl(i)<1) then error;
         let l∈Array-loc be s.t.
              scl∈rng l ⇒ is-tp-scalar-loc(sctp,scl) &
              dom l = rect(ebdl) &
              rng l ∩ dom c R-STG = {}
         R-STG := c R-STG∪[scl↦? | scl∈rng l];
         return(l)
        )

type: Type Env  ⇒ Loc
```

```
e-proc(mk-proc( ,parml,st),env) =
    let f(denl,cas) =
         (if lendenl≠lenparml ∨
             (∃i∈{1:lenparml})(¬is-pmatch(denl(i),s-attr(parml(i)))
               then error;
          let lenv = env +
               [s-id(parml(i)) ↦ denl(i) | i ∈ {1:len parml}]
          i-stmt(st,lenv,cas)
         )
    mk-proc-den(f)

type: Proc Env → Proc-den


i-named-stmt-list(nsl,env,cas,aid) =
    tixe [mk-label-den(aid,tid)↦it-named-stmt-list(tno,nsl,env,cas.
                               | 1≤tno≤lennsl & s-nm(nsl(tno))=tid≠NIL]
    in it-named-stmt-list(1,nsl,env,cas)

type: Named-stmt* Env Aid-set Aid ⇒


it-named-stmt-list(i,nsl,env,cas) =
    for j = i to len nsl do i-stmt(s-stmt(nsl(j)),env,cas)

type: Nat Named-stmt* Env Aid-set ⇒


i-if(mk-if(be,th,el),env,cas) =
   (let bv: e-expr(be,env);
    if bv then i-stmt(th,env,cas)
    else       i-stmt(el,env,cas)
```

```
i-for(mk-for(cv,ie,se,le,st),env,cas) =
   (let i: e-expr(ie,env);
    let s: e-expr(se,env);
    let l: e-expr(le,env);
    let f(x) = if (s>0 → x≤l, s<0 → x≥l, s=0 → TRUE)
                    then (i-stmt(st,env+[cv→x],cas);f(x+s))
    f(i)
   )


i-call(mk-call(pid,al),env,cas) =
   (let mk-proc-den(f) = env(pid)
    let dl: <is-var-ref(al(i)) → e-var-ref(al(i),env),
                 T                   → env(al(i))     | 1≤i≤len al>;
    f(dl,cas)
   )


i-goto(mk-goto(id),env) =
    exit(env(id))


i-assign(mk-assign(vr,e),env) =
   (let l: e-var-ref(vr,env);
    let v: e-expr(e,env);
    assign(l,v)
   )


i-in(mk-in(vr),env) =
   (if c R-IN  = <> then error;
    let l: e-var-ref(vr,env);
    let v: hd c R-IN;
    if ¬is-vmatch(v,var-ref-tp(vr)) then error
    else
         (R-IN := tl c R-IN;
          assign(l,v)
```

```
i-out(mk-out(e),env) =
   (let v: e-expr(e,env);
    R-OUT := c R-OUT ^ <v>
   )


e-expr: Expr Env ⇒ Value


e-infix-expr(mk-infix-expr(e1,op,e2),env) =
   (let v1: e-expr(e1,env);
    let v2: e-expr(e2,env);
    let v: apply-op(v1,op,v2);
    return(v)
   )


e-rhs-ref(mk-rhs-ref(vr),env) =
   (let l: e-var-ref(vr,env);
    contents(l)
   )


e-var-ref(mk-var-ref(id,sscl),env) =
    if sscl=NIL then return(env(id))
    else
        (let aloc = env(id)
         let esscl: <e-expr(sscl(i),env) | 1≤i≤lensscl>;
         if esscl∉dom aloc then error;
         return(aloc(esscl)))

type: Var-ref Env ⇒ Loc


e-con-var-ref(mk-con-var-ref(id),env) = return(env(id))


e-const(e,) = /* return a Value corresponding to the Constant */
```

```
apply-op(v1,op,v2) = /* Returns an appropriate Value */
type: Value Op Value ⇒ Value


assign(l,v) =
    R-STG := c R-STG + [l ↦ v]

type: Scalar-loc Value ⇒
pre:  l∈dom c R-STG, is-lmatch(l,v)


contents(l) =
    let v: c R-STG(l);
    if v=? then error;
    else return(v)

type: Scalar-loc ⇒ Value
pre:  l∈dom c R-STG


rect(il) = /* generates a dense rectangle of integers */
type: Int⁺ → (Int⁺)-set
pre:  1≤i≤ len il ⇒ il(i)≥1


is-lmatch(l,v) = /* checks location and value are of same scalar type */
type: Scalar-loc Value → Bool


is-vmatch(v,t) = /* checks value is of scalar type */
type: Value Scalar-type → Bool


is-pmatch(d,s) = /* checks the argument matchs parameter specification *,
type:  (Loc | Proc-den) (Type | PROC) → Bool
```

```
is-tp-scalar-loc(tp,l) =
    tp=BOOL → is-bool-loc(l)
    tp=INT → is-int-loc(l)

type: Scalar-type Scalar-loc → Bool


is-cont(id,nsl) = (∃i∈{1:lennsl})(id=s-nm(nsl(i)))
type: Id Named-stmt* → Bool
```