famous PERS

TECHNICAL REPORT

TR 25.145
16 February 1976

FORMAL DEFINITION IN COMPILER DEVELOPMENT

C. B. JONES

IBM LABORATORY VIENNA

FORMAL DEFINITION IN COMPILER DEVELOPMENT

C.B.Jones

Abstract:

This report indicates how the task of Compiler Development can be decomposed. Specifically, the proposal is to begin with a language definition which defines the class of (abstract) programs and their semantics by a mapping to functions over (abstract) states; then to document a mapping from programs to transformations of the target machine states and to "annotate" this in a way which shows it to be correct with respect to the language definition; a further mapping, from programs to sequences of instructions of the target machine, is then documented and "annotated" to show how the given instructions realize the transformations.

C O N T E N T S

1. INTRODUCTION

The existence of a formal definition for a programming language makes it possible to give a precise statement of the correctness criteria of a compiler for that language. This is true whether the definition is documented in the "mathematical", "operational" or "axiomatic" style. Compiler construction is however difficult and tends to result in a large program: the precision with which the final assertion of correctness can be given may be of a little use if the compiler is in fact in error. Thus it is more interesting to study how a language definition can be employed in a development method for compilers.

The essence of the method proposed below is to decompose the development into steps of a manageable size. Providing the intent of each is clearly understood, it should then be possible to make each step with a high probability of correctness. Furthermore, if the role of each step can be formalised, a structure exists to provide a formal correctness argument. It is not being suggested that formal proofs should be given for whole compilers, only that a designer now has a framework in which he can provide indications as he sees fit. (even one, well placed, assertion can immensly simplify the task of understanding the intention of a design step.)

It will be possible to describe the method below more concisely if the author can firstly ensure that no misunderstandings will result from an over-literal reading. The process is a stepwise one and it is intended that the final record of a development would exhibit this structure. However, this does not imply that the designer is being asked to think in a fixed order ("top-down"). It is in fact essential that the designer has a picture of the future development. But, as the decisions in his design become fixed it is equally important that they are documented in a structure: it is the purpose of the current paper to propose such a structure. Notice, in the same way, no automatic design process is being claimed. The invention of what should be documented must come from the designer.

Another possible reaction to the description of the method as applied to the example developed below is that some people will consider that the whole process is too formal. The level of formality used in different parts of the example, in fact, differs widely. Nearly all parts are more formal than would be proposed for a normal project. It is, however, simpler to appreciate what is to be done by studying this level of formality. It is only based on this understanding that it is reasonable to go on and consider useful and safe "rules of thumb" to be used in development.

One last area of misunderstanding that it is worth trying to avoid is the role of the example. The subject of the paper is neither the particular implementation of the compiler nor its proof as given. Rather the subject is a documentation style in which such a development and proof can be presented.

The method itself is closely related to those of refs. 4 and 6 and attempts to document compiler development in the following major stages:

Language Definition
Mapping (to target machine states)
Code Sequences
Translator Development

The formal definition of the source language is based on a class of abstract programs. Members of this class are intended to correspond to one or more concrete programs (as written, according to the concrete syntax of the language); they are in a sense a normal form for programs removing many of the textual details which do not influence the meaning of a program. The class is defined by a set of abstract syntax rules in such a way that its members can be considered to be trees in which access to information is greatly eased. The basic class can be restricted by "Context Conditions" which specify constraints to be satisfied by the members, where such constraints depend only on the static form of the program. It is also convenient to compute certain results from the static form of a program by means of "Context Functions".

Given the class of abstract programs, the semantics of the language is specified by providing a mapping onto transformations (i.e. functions from states to states). This mapping is written using a meta-language, that used below is taken from ref. 5. The "states" chosen for a given language should be abstract in the sense that they should eschew properties which are irrelevant to showing the final result of a program. The semantics should be "Denotational" in the sense that the meaning of a program (fragment) should depend only on the meaning ascribed to its subparts.

The objective of developing a Mapping is to provide a new function from abstract programs to transformations. In this case, however, the states in question should be representable on (i.e. expressed in terms of the data objects of) the target machine being considered. This new mapping follows as closely as possible the structure of the source definition. Furthermore the transformations are still defined in the meta-language. The new representable states are concrete realisations of an abstraction (i.e. the state of the source definition) and as such there will normally be a many to one relation between the classes. If this relation between source states (Σ) and machine states ($m\text{-}\Sigma$) is recorded by a function:

retr-Σ : $m\text{-}\Sigma \rightarrow \Sigma$

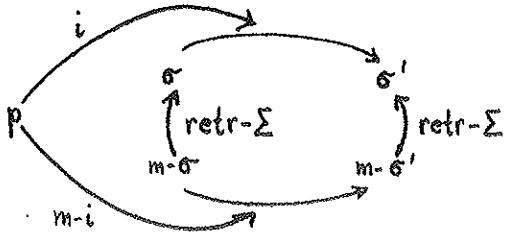
the source definition is given by:

$$i : p \rightarrow (\Sigma \rightarrow \Sigma)$$

and the "mapping" by:

$$m\text{-}i : p \rightarrow (m\text{-}\Sigma \rightarrow m\text{-}\Sigma)$$

then it is necessary to show:



Because both the source and mapping functions are based on the structure of abstract programs, it is possible to decompose this into an inductive proof on this structure. Details of the proof steps and the annotation style proposed are given in section 2.2.

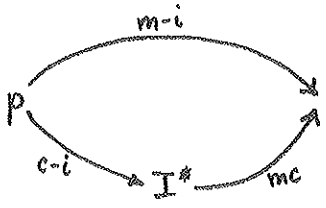
Notice that this major stage of developing a mapping can be decomposed into smaller steps employing the same method.

The next major stage is to document a function from abstract programs to Code Sequences (i.e. lists of machine instructions of the target machine).

Assuming:

$$mc : I^* \rightarrow (m\text{-}\Sigma \rightarrow m\text{-}\Sigma)$$

it is necessary to show:

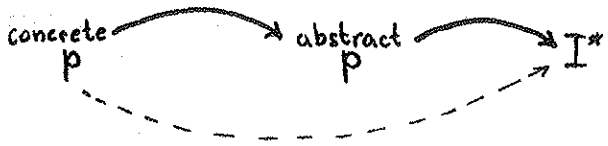


The correctness argument at this stage is simply that the machine instructions given should perform the same transformation as had earlier been described in meta-language concepts. Section 2.2.2 discusses how such a mapping can be annotated to record its justification. Depending on how well the target machine is known and on how formally the proof is to be presented, it may be necessary to have a formal definition of the machine instructions (i.e. the function mc above).

The subject of representing the overall mapping in terms of a composition of more than one mapping is discussed in section 2.2.3.

At this stage the designer is still very far from his running compiler, he has documented a mapping from abstract programs to instruction sequences, in this mapping a number of context functions are used. Furthermore the tree form of abstract programs has made it easy to gather auxiliary information (e.g. to jump ahead of a left to right scan). The mapping functions themselves will have used abstract objects which are not directly available on the translator machine. As regards abstract programs it is straightforward to document the relationship with concrete programs or, more usefully, with a linear form of parse trees which will be created by a first pass of the translator. An example of the relationship between PL/I abstract programs and the internal text created by a product compiler is given in ref. 9.

Thus by considering:



an implicit specification of the translation process (t) is available. It is then possible to begin a (stepwise) Translator Development. The process of developing concrete representations for the abstract (translate time) objects and developing algorithms for implicitly defined functions is discussed in part 3 of ref. 6. Of particular importance in this process is the part played by abstract syntax. The mere use of this abstraction has made the translator specification very abstract. In developing towards the concrete representation of programs, it must be borne in mind that the actual input to the compiler is an arbitrary character string, and for most compilers good design of diagnostics identifying errors in non-programs will be of considerable importance.

It is likely that the final translator structure will be multi-pass for most interesting programming languages. It is important to beware pre-empting an undesirable phase division in developing the specification of the mapping. This is not likely to result from a (well chosen) abstract syntax; the creation of a dictionary is a fairly safe assumption; a split like that considered in section 2.2.3 for post optimisation is, however, likely to be difficult to unravel in the eventual development. There is no objection to introducing assumptions about the translator structure during the Mapping and Code Sequence stages, provided only that the designer is satisfied that these assumptions will correspond to the final translator structure.

A method has been proposed for decomposing compiler development into a number of stages. One observation might serve to illustrate the advantages of the decomposition. Any requirement to "understand" the source language semantics has been eliminated by the end of the mapping stage; after the code sequence stage the whole compiler problem has been reduced to a simple string (strictly tree) processing specification. Those involved with performing the next stage of this process need not be concerned with how the earlier stages were constructed. Even if one is interested in comparing the code sequences (cf. Section 6) with the source definition (cf. Section 3), it is only necessary to study the development of the mapping (sections 4,5) where some understanding of the correctness argument is required.

ACKNOWLEDGEMENTS

The work leading to the current paper was done in conjunction with members of the Vienna Laboratory (see references): the author is grateful to these colleagues without whom the current work would not have been possible. For direct contributions to the current report (cf. ref. 8) thanks are due to H.Izbicki for cooperation on the for mapping and F.Weissenböck for many of the code sequences.

REFERENCES

1. E.W.Dijkstra, "Recursive Programming", Num.Math.2, No.5, 1960.

2. W.Henhapl and C.B.Jones, "The Block Concept and Some Possible Implementations, with Proofs of Equivalence", IBM Laboratory Vienna, Lab.Report TR 25.104, April 1970.

3. C.B.Jones and P.Lucas, "Proving Correctness of Implementation Techniques", in "Symposium on Semantics of Algorithmic Languages" (Ed.) E.Engeler, Springer-Verlag Lecture Notes in Mathematics No. 188, October 1970.

4. P.Lucas , "On Program Correctness and the Stepwise Development of Implementations", presented at A.I.C.A Conference, Pisa University, March 1973.

5. H.Bekic, D.Bjørner, W.Henhapl, C.B.Jones and P.Lucas, "A Formal Definition of a Subset of PL/I", IBM Laboratory Vienna, Report TR 25.139, December 1974.

6. C.B.Jones, "Formal Definition in Program Development", in "Programming Methodology", Springer-Verlag Lecture Notes in Computer Science No.23, 1974.

7. W.Henhapl and F.Weissenböck, "A Formal Mapping Description", IBM Laboratory Vienna, Lab.Note LN 25.3.105, February 1975.

8. H.Bekic, H.Izbicki, C.B.Jones and F.Weissenböck, "Some Experiments with using a Formal Definition in Compiler Development", IBM Laboratory Vienna, Note LN 25.3.107, December 1975.

9. F.Weissenböck, "A Formal Interface Specification", IBM Laboratory Vienna, Techn.Report TR 25.141, February 1975.

10. H.Izbicki, "A Type-matching Theorem for all Expressions in a High-level Language Defined by Meta-IV", IBM Laboratory Vienna, Note LN 25.3.106, April 1975.

11. C.B.Jones, "Mathematical Semantics of Goto: Exit Formulation and its Relation to Continuations", forthcoming.

12. R.Milne and C.Strachey, "Mathematical Semantics"(?), forthcoming book.

2. NOTATION AND METHOD

2.1 Notation

Apart from some minor extensions, the notation used in the current report is that of ref 5. This section provides only a quick "reading guide". For more detailed definition the earlier report should be consulted.

2.1.1 Objects

Bool = {true,false}
xyz arbitrary elementary object

Sets:

$\{x_1,x_2,\ldots,x_n\}$	set with given elements
{}	empty set
{x | p(x)}	implicit set definitions
{x∈X | p(x)}	
{f(x) | p(x)}	
{m:n} = {i | m≤i≤n}	
x∈S	test for membership

∪ union, ∩ intersection, ⊂ proper subset, ⊆ subset, \ difference
$\underline{B}S = \{s \mid s \subseteq S\}$
$\underline{\text{union}}\ (\{S_1,S_2,\ldots,S_n\}) = S_1 \cup S_2 \cup \ldots \cup S_n$

Lists:

$\langle x_1,x_2,\ldots,x_n\rangle$	list with given elements
<>	empty list
<f(i) | i∈{m:n}>	implicit list definitions
<f(i) | m≤i≤n>	
<c | i∈{m:n}>	

$\underline{l}$ length, $\underline{h}$ head, $\underline{t}$ tail, l[i] element selection, ^ concatenation
$l[i:j] = \langle l[i],l[i+1],\ldots,l[j]\rangle$
$\underline{D}l = \{1:\underline{l}\ l\}$
$\underline{R}l = \{l[i] \mid 1\le i\le \underline{l}\ l\}$

Maps:

$[d_1 \to r_1,d_2 \to r_2,\ldots,d_n \to r_n]$	map with given pairs
[ ]	empty map
[d → f(d) | p(d)]	implicit map definitions
[g(x) → h(x) | p(x)]	
$\underline{D}$ set of domain elements, $\underline{R}$ set of range elements	
m(d)	application
$m_2 + m_2$	overwrite
$m_1 \cup m_2$	union
$[d \to r] \in M = d \in \underline{D}M \wedge M(d) = r$	
$M \setminus S = [d \to m(d) \mid d \in (\underline{D}M \setminus S)]$	
$M \mid S = [d \to m(d) \mid d \in (\underline{D}M \cap S)]$	
$M1 \subseteq M2 = \underline{D}M1 \subseteq \underline{D}M2 \wedge (d \in \underline{D}M1 \supset M2(d) = M1(d))$	

2.1.2 Abstract Syntax

Rules:

N = A	Define set N
N = A | B	N = A ∪ B
N :: A	N = {mk-N(a) | a∈A}
N = (D1 -> R1) $\underline{u}$ (D2 -> R2)	N = (D1 ∪ D2) -> (R1 ∪ R2) s.t. d∈D1 ⊃ N(d)∈R1 etc.

Right hand sides:

$A_1\ A_2\ \ldots\ A_n$	$\{\langle a_1,a_2,\ldots,a_n\rangle \mid a_1 \in A_1 \wedge a_2 \in A_2 \wedge \ldots \wedge a_n \in A_n\}$
A*	list
A+	non-empty-list
A-set	$\underline{B}A$
[A]	optional element
A -> B	Map
A <-> B	One-one map

Related functions:

mk-θ( )	Constructor
s-a( )	selector
is-θ(o)	Predicate: o∈θ

2.1.3 Functions and Expressions

Functions:

Lambda notation used -

f(d) = e	≡ f = λd.e
g°f	= λd.g(f(d))

Expressions:

if then else, conditional expressions, cases	
(let r = e; B)	= (λr.B)(e)
¬ not, ∧ and, ∨ or, ⊃ implies, ≡ equivalence,	
(∃x∈X)(p(x))	Existensional quantification(bounded)
(∀x∈X)(p(x))	Universal quantification
(∃!x∈X)(p(x))	there exists exactly one
(ιx)(p(x))	the unique object such that

usual arithmetic operators.

2.1.4 Transformations

For states Σ:

⊆ r	= λσ.(σ(r))
r := v	= λσ.(σ ∔ [r → v])

Transformations are, in the simplest case, functions from states to states.

Tr = Σ -> Σ

Normal Sequencing (";"), if then else and for are defined in an obvious way, as are the extensions to value returning transformations (Σ -> Σ B).

Exit:

Transformations from which exits can arise are of the form:

TR = Σ -> Σ [Abn]	
s∈Σ -> Σ	= λσ.<s(σ),nil>
exit (abn)	= λσ.<σ,abn>
$s_1;s_2$	= λσ.(let <σ',abn'> = s_1(σ) if abn' = nil then s_2(σ') else <σ',abn'>)
(trap exit (abn) with f(abn); e)	= λσ.(let <σ',abn'> = e(σ); if abn' = nil then <σ',nil> else f(abn)(σ'))

Arbitrary order:

For the purposes of the current report it is possible to regard the "," combinator as leaving only the local freedom to invert the operands before changing to ";".

branch	local sequence changes only
call / ret	obvious

2.2 Method

In the introduction, a sequence of stages was outlined for the complete development of a compiler. This section describes the method to be used below for the two stages: Mapping to Machine States and Mapping to Code Sequences. It is important to bear in mind that the proposals being made for decomposing the development into stages provide a structure for the final documentation: it is not the intention to suggest that thinking should be constrained to a top-down structure. In fact the division of Map and Code Sequence stages introduces a very nasty trap for those who do not have a sketch of the next stage in mind. In the Map stage the designer is describing transformations on the target machine state in the meta-language; he is

only forced to think about the ease of realising these transformations by instructions in the next stage. Although most required transitions are likely to be attainable, the number of instructions required may be a serious argument for choosing a different (sequence of) transformation(s). (A case in point is the somewhat special restrictions on using the index/increment/test instructions of the 360 machine). Unless a clear idea of the instruction repertoire influences the design of the transformations either poor code performance or expensive backtracking will be the toll paid. It is however the opinion of the current author that there is virtue in dividing the documentation of the development in order to attain a higher chance of correctness.

2.2.1 Mapping to Target Machine

A mapping must contain at least a description of the states of the target machine and a (series of) function(s) which map abstract source programs to target transformations (i.e. functions from target states to target states). In addition an argument for the correctness of this mapping can be recorded by "annotating" with the source transformations and assertions of equivalence.

Suppose that the machine states were the same as those of the source definition, proving that m-i is a valid implementation of i would require:

$$(\forall \sigma \in \underline{D}\ i)(m\text{-}i(\sigma) = i(\sigma))$$

If however, we are forced to consider implementations which are not functions but rather relations the problem is to show:

a $(\forall \sigma \in \underline{D}\ i)((\exists \sigma')(r\text{-}i(\sigma,\sigma')) \wedge (r\text{-}i(\sigma,\sigma'') \supset \sigma' = i(\sigma)))$

One is brought to the notion of the implementation being a relation because the states of the target machine not only differ from those of the source definition but also do not stand in a one to one relation thereto. Thus:

$$q : m\text{-}\Sigma\ \Sigma \to Bool$$
$$m\text{-}i : m\text{-}\Sigma \to m\text{-}\Sigma$$
$$r\text{-}i(\sigma,\sigma') = (\exists m\text{-}\sigma)(q(m\text{-}\sigma,\sigma) \wedge q(m\text{-}i(m\text{-}\sigma),\sigma'))$$

Because of the inductive structure to be applied in the proof, it is convenient to prove:

b $(\forall \sigma \in \underline{D}\ i)(q(m\text{-}\sigma,\sigma) \supset q(m\text{-}i(m\text{-}\sigma),i(\sigma)))$

Unfortunately use of either the universally true or universally false relation would then permit the "verification" of any implementation however false it were. Intuitively there should be at least one target representation for each distinct source state:

c $(\forall \sigma)(\exists m\text{-}\sigma)(q(m\text{-}\sigma,\sigma))$

and one representation cannot represent two distinct source states:

d $q(m\text{-}\sigma,\sigma_1) \wedge q(m\text{-}\sigma,\sigma_2) \supset \sigma_1 = \sigma_2$

To see that this is formally adequate for proving a by establishing b:

1	$\sigma \in \underline{D}\ i$	assm
2	$(\exists m\text{-}\sigma)(q(m\text{-}\sigma,\sigma))$	1,c
	<u>let</u> $m\text{-}\sigma$ be such:	
3	$q(m\text{-}\sigma,\sigma)$	2
4	$q(m\text{-}i(m\text{-}\sigma),i(\sigma))$	1,3,b
	thus $(\exists \sigma')(q(m\text{-}\sigma,\sigma) \wedge q(m\text{-}i(m\text{-}\sigma),\sigma'))$	3,4

and:

1	$\sigma \in \underline{D}\ i$	assm
2	$(\exists m\text{-}\sigma)(q(m\text{-}\sigma,\sigma) \wedge q(m\text{-}i(m\text{-}\sigma,\sigma'))$	assm
	<u>let</u> $m\text{-}\sigma$ be such:	
3	$q(m\text{-}\sigma,\sigma) \wedge q(m\text{-}i(m\text{-}\sigma),\sigma'))$	2
4	$q(m\text{-}i(m\text{-}\sigma),i(\sigma))$	3,b
5	$\sigma' = i(\sigma)$	3,4,d
	thus $(\exists m\text{-}\sigma)(q(m\text{-}\sigma,\sigma) \wedge q(m\text{-}i(m\text{-}\sigma),\sigma')) \supset \sigma' = i(\sigma)$	2,5

The proof of property b will be shown for each mapping function: these, as in the source definition, follow the structure of abstract programs. Since these functions are mutually recursive there is a hidden appeal to the finiteness of programs and their successive decomposition to justify the use of recursion induction.

The fact that source states stand in a one-many relation with target states will be shown below by defining the relation in terms of a retrieve function:

$$retr\text{-}\Sigma : m\text{-}\Sigma \to \Sigma$$

Since this function is not total (there are states in m-Σ which do not correspond to a state in Σ), the notion of cons-m-Σ is introduced below.

The situation is, of course, more complex than that discussed above: both the functions of the source definition and their corresponding mapping functions produce transformations (functions from states to states) only after being applied to other arguments. It is necessary to consider the equivalence of these other arguments. The objects which are (parts of) abstract programs are used directly by both series of functions. Typical of the auxiliary arguments is the environment (Env) of the source document. In the implementation this will be modelled partly by a static dictionary (Dict) and partly by information in storage (Dsa). Essentially then, the proof takes the form:

```
cons-m-Σ(m-σ) ⊃
(σ-env(dict,env)(m-σ) ∧ σ = retr-Σ(m-σ) ⊃
retr-Σ°m-i-θ(t,dict)(m-σ) = i-θ(t,env)(σ))
```

Where an explicit construction exists in the source definition it is necessary to show how an equivalent object can be modelled in the object states. An example of this is Array-loc which is defined to be a mapping (cf. DS6): this mapping can be reconstructed from the object state (cf. P16) with the aid of auxiliary variables (cf. section 7 for a discussion of alternatives). Such "ghost variables" play a part only in the correctness argument and none of the target transitions are made to depend on them.

There are situations in the source document where no explicit construction is defined, but rather certain properties of the required result are stated. An example of this is the choice of new scalar locations which are only required (cf. DF4) to be distinct from current locations. In such a case the mapping may choose any object which satisfies the property.

The argument that the mapping does in fact correspond to the source definition (in a way made precise in section 4.4) is recorded by annotating the functions which map abstract programs to target transformations.

Simple assertions apply only to the states of the target machine, thus:

```
pre  : D -> (m-Σ -> Bool)
post : D -> (m-Σ m-Σ -> Bool)
```

then, using the same conventions on "sigma-free" notation used in the source document:

```
pre(d)(m-σ) ∧
m-σ' = m-op(d)(m-σ) ⊃
post(d)(m-σ,m-σ')
```

will be written:

```
        !  pre(d)
m-op(d)
        !  post(d)     reasons
```

where, of course, m-op could be a compound construct in parenthesis.

To aid the reader, assertions on equivalence are marked by "!!", thus:

```
r1,r2 : D -> (m-Σ Σ -> Bool)
i : D -> (Σ -> Σ)
m-i : D -> (m-Σ -> m-Σ)

r1(d)(m-σ,σ) ⊃
   r2(d)(m-i(d)(m-σ),i(d)(σ))
```

will be written:

```
        !! r1(d)
m-i(d)
        !! i(d)          ⁻
        !! r2(d)         reasons
```

In a number of places it is convienient to document statements which differ slightly from those of the source definition. One particular "trick" will be to expand a set or mapping operation of the source document into a declare followed by an element generation loop in order to facilitate the placing of assertions. Where the difference is large enough to admit doubt of equivalence a separate argument for correctness can be provided (cf. map of int-for).

It is necessary to indicate the valid forms of reasoning for combinators. These are given in the form of Annotation Lemmas.

AL1

This relates to the "if" combinator and there are obvious extensions to conditional statements and the case construct. (In fact this reasoning is usually performed informally without explicit reference to AL1: numbering of the formulae "n.m" is used).

```
                ! let m-σ1 be current
                !! r
   if α
      then
         (      !! r                          AL1hyp
          m-th
                !! th                         ~
                !! r'                         reasons
                ! post(m-σ1,)                 reasons
         )
      else
         (      !! r                          AL1hyp
          m-el
                !! el                         ~
                !! r'                         reasons
                ! post(m-σ1,)                 reasons
         )
                !! r'                         AL1
                ! post(m-σ1,)                 AL1
```

AL2

This relates to the "for" combinator:

```
                ! let m-σ1 be current
                !! r
      for e ∈ S
         (      ! post(m-σ2,)                 AL2hyp
                !! r                          AL2hyp
          m-b(e)
                !! b(e)                       ~
                !! r                          reasons
                !! r'(e)                      reasons
                ! post(m-σ2,)                 reasons
         )
                !! r                          AL2
                !! e ∈ S ⊃ r'(e)              AL2
                ! post (m-σ2,)                AL2
```

Strictly, should also show non-interface:

```
                !! e' ∈ Sn ⊃ r'(e')
                ...
                !! e' ∈ (Sn ∪ {e}) ⊃ r'(e')
                   where Sn+1 = Sn ∪ {e}
```

AL3

This relates to the trap exit combinator:

```
                !  let m-σ1 be current
                !! r
                !! b                          ~
((trap exit (abn) with
                !  post1(m-σ1,)               AL3hyp
                !! r1                         AL3hyp
   (m-e;
                !! r2                         reasons
                !  post2(m-σ1,)               reasons
    exit (abn)
   );
                !  cur m-σ = m-σ1             AL3hyp
                !! r                          AL3hyp
   m-b
                !! r1                         reasons
                !  post1(m-σ1,)               reasons
);
                !  post1(m-σ1,)               AL3hyp
                !! r1                         AL3hyp
m-n
                !! r2                         reasons
                !  post2(m-σ1,)               reasons
)
                !! r2                         AL3
                !! post2(m-σ1,)               AL3
```

It is also considered a valid mapping step to simply ignore any test whose sole purpose in the source definition was to yield error. This indicates that a user of the compiler being designed will not always receive a warning when he contravenes a language rule (e.g. subscript out of range). If this is acceptable to the users then the designer is using the freedom to interpret error in the source definition as "subsequent results not defined".

There are two points which deserve special mention. Firstly there is the problem of describing the relationship between procedure denotations in the source definition and their realisations in the machine state. In DS11 Proc-den's are described as functions, but these functions have been created (cf. DF 5) so as to reflect the environment (Env) current at the point of their declaration. They are "closures". The result of branching to the label denoting the first instruction of a target procedure (cf MS27) depends, however, on the current contents of those parts of the state which mirror Env (i.e., Env1, g-tp, g-ai). It is, then, necessary to constrain the class of states in which an equivalent result may be expected by "applying" an m-Proc-den. This is achieved by storing a Proc-inf (cf MS29) for each procedure, in a new ghost variable, which gives the value of the three components mentioned as they were at the point of declaration. It is then possible to describe that states are only valid if, for all stored Proc-inf's the current values of the three considered components "extend" the recorded values.

Stated thus the property is still not strong enough because an arbitrary state where the Proc-infs have also been changed would not be valid. The states considered acceptable must also extend the Proc-inf at the time of declaration. All of this rather complex requirement is made rather easy to prove because of the ability to characterise state changes which preserve the properties (cf. L5).

The other aspect of the proof which should be mentioned is the reasoning used for the goto statement of the source language. The definition in section 3 uses the "exit mechanism" of the metalanguage. As mentioned in ref 11, one important measure of the suitability of the exit mechanism as a definition tool is the ease with which it can support subsequent reasoning. Here the proof of the required implementation technique for goto statements which can abnormally terminate blocks or procedures has been divided into two stages. (The split of section 4.6, 5 was made for other reasons, cf. section 4.1.2). In the first stage the action to be performed in the trap exit unit is changed: in most cases to the identity operation. It is then possible to use the Branch instruction to exactly simulate the transformation caused by performing exit. At both stages the reasoning is relatively simple.

2.2.2 Code Sequences

The stage described in section 2.2.1 results in a mapping from abstract programs to transformation functions for the target machine state. Notice that the annotations, including those for updating the ghost variables, can now be ignored. The next stage is to consider the mapping from abstract programs to sequences of instructions for the object machine. Each instruction of the object machine can also be thought of as a transformation. Overall the requirement is that the two transformations are equivalent. The argument that this be so is a composition of very simple local arguments and is indicated by annotating the code sequence map with the transformations.

2.2.3 Discussion

The arguments as documented via annotations are easy to follow and by no means difficult to construct. This is, however, a little deceptive in that the real work lies in finding the appropriate relationships between states and consistency conditions thereon.

The problem of register allocation has not really been considered in the mappings given below (an arbitrary number assumed). If it were to be treated it should come in a separate stage after the current mapping to (an idealised) machine. This would then correspond to an extra translator pass:

```
f    : θ -> (Σ -> Σ)
m-f  : θ -> (m-Σ -> m-Σ)
c-f  : θ -> m-I*
m    : m-I* -> (m-Σ -> m-Σ)
p2   : m-I* -> mI2*
m2   : m-I2* -> (m-Σ2 -> m-Σ2)
```

The proof should be simple because of the closeness of m-Σ and m-Σ2.

Notice that this is not the treatment proposed for those optimisations which can be easily described by a function over the structure of abstract programs. In this case the "optimization" should be included in the first mapping.

This section ended on the previous page.

3 SOURCE DEFINITION

3.1 Abstract Syntax

This section defines the class of objects considered as abstract programs. References to formulae of this section are of the form "DAn".

```
1  Prog :: Stmt

2  Stmt = Block | If | For | Call | Goto | Assn | In | Out | null

3  Block :: s-dcls:(Id -> (Type | Proc)) Nmd-stmt*

4  Type :: Sc-type s-bds:[(Expr | *)*]

5  Proc :: Parm* Stmt

6  Parm :: Id s-attr:(Type | proc)

7  Nmd-stmt :: s-nm:[Id] Stmt

8  If :: s-b:Expr s-th:Stmt s-el:Stmt

9  For :: s-cv:Id s-init:Expr s-step:Expr s-limit:Expr Stmt

10 Call :: s-pn:Id s-args:(Var-ref | Id)*

11 Goto :: Id

12 Assn :: s-lhs:Var-ref s-rhs:Expr

13 In :: Var-ref

14 Out :: Expr

15 Expr = Inf-expr | Rhs-ref | Cv-ref | Const

16 Inf-expr :: Expr Op Expr

17 Rhs-ref :: Var-ref

18 Var-ref :: Id s-sscs:[Expr*]

19 Cv-ref :: Id

20 Const = Intg-const | Bool-const          )
                                            ) disjcint alternatives
21 Op = Intg-op | Bool-op | Rel-op          )

22 Sc-type = intg | bool

23 Id   not further defined
```

3.2 Context Conditions

This section defines a restriction on the class of abstract programs: members of the restricted class are known as well-formed programs. This restriction is defined, as in ref. 5, by predicates of the form is-wf-0 which depend on a text and a static environment:

```
Env-s = Id -> (Type | proc | lab | cv)
```

The numbering of the context conditions is, as far as possible, that of the corresponding abstract syntax rules. References to these context condition functions will be of the form "DCn".

```
1  is-wf-prog(<t>) = is-wf-stmt(t,[])
```

```
3  is-wf-block(<dcls,ns-l>,env) =
       let ll = <s-nm°ns-l[i] | 1≤i≤lns-l ∧ is-id°s-nm°ns-l[i]>;
       is-unique-ids(ll) ∧
       is-disjoint(Ddcls,Rll) ∧
       (let r-env = env \ (Ddcls ∪ Rll);
        let n-env = r-env ∪
                    [id → (if is-type(dcls(id))
                              then star(dcls(id))
                              else proc) | id∈Ddcls] ∪
                    [id → lab | id∈Rll];
        dcl∈Rdcls ⇒
          (is-type(dcl) ∧
           (s-bds(dcl)=nil ∨
            is-expr-list°s-bds(dcl) ∧
            (1≤i≤ls-bds(dcl) ⇒ ex-tp(s-bds(dcl)[i],r-env)=intg)) ∧
           is-wf-type(dcl,r-env)) ∨
          (is-proc(dcl) ∧ is-wf-proc(dcl,n-env)) ∧
        1≤i≤lns-l ⇒ is-wf-stmt(s-stmt°ns-l[i],n-env))


5  is-wf-proc(<parm-l,st>,env) =
       is-unique-ids(<s-id°parm-l[i] | 1≤i≤lparm-l>) ∧
       (let n-env = env ∔
                    [s-id°parm-l[i]→s-attr°parm-l[i] | i∈{1:lparm-l}];
        is-wf-stmt(st,n-env))


6  is-wf-parm(<id,attr>,env) =
       attr = proc ∨
       s-bds(attr) = nil ∨ is-*-list°s-bds(attr)


8  is-wf-if(st,env) =
       ex-tp(s-b(st),env) = bool


9  is-wf-for(<id,init,step,limit,st>,env) =
       let n-env = env ∔ [id → cv];
       ex-tp(init,env) = intg ∧
       ex-tp(step,env) = intg ∧
       ex-tp(limit,env) = intg ∧
       is-wf-stmt(st,n-env)


10 is-wf-call(<pn,a-l>,env) =
       pn∈Denv ∧ env(pn) = proc ∧
       (1≤i≤la-l ⇒ is-wf-var-ref(a-l[i],env) ∨
                   env(a-l[i]) = proc)


11 is-wf-goto(<id>,env) =
       id∈Denv ∧ env(id) = lab


12 is-wf-assn(<lhs,rhs>,env) =
       is-scalar(lhs,env) ∧
       ex-tp(rhs,env) = vr-tp(lhs,env)


13 is-wf-in(<v-r>,env) =
       is-scalar(v-r,env)
```

Section: SOURCE DEFINITION

```
151 ex-tp(e,env) =
      cases e:
        mk-inf-expr(e1,op,e2) ->
           (is-intg-op(op) -> intg
            is-bool-op(op) -> bool
            is-rel-op(op) -> bool)
        mk-rhs-ref(v-r) -> vr-tp(v-r,env)
        mk-cv-ref(id) -> intg
        T -> (is-intg-const(e) -> intg
              is-bool-const(e) -> bool)

   type: Expr Env-s -> Sc-type


16 is-wf-inf-expr(<e1,op,e2>,env) =
      ex-tp(e1,env) = intg ∧ is-intg-op(op) ∧ ex-tp(e2,env) = intg ∨
      ex-tp(e1,env) = bool ∧ is-bool-op(op) ∧ ex-tp(e2,env) = bool ∨
      ex-tp(e1,env) = intg ∧ is-rel-op(op) ∧ ex-tp(e2,env) = intg


17 is-wf-rhs-ref(<r>,env) =
      is-scalar(r,env)


18 is-wf-var-ref(<id,e-l>,env) =
      id∈Denv ∧ is-type°env(id) ∧
      (e-l ≠ nil ⊃ (s-bds°env(id) ≠ nil ∧
                    le-l = ls-bds°env(id) ∧
                   (1≤i≤le-l ⊃ ex-tp(e-l[i],env) = intg)))


181 vr-tp(<id, >,env) =
      s-sc-type°env(id)

   type: Var-ref Env-s -> Sc-type


182 is-scalar(<id,ssc-l>,env) =
      ssc-l = nil ⊃ s-bds°env(id) = nil

   type: Var-ref Env-s -> Bool


19 is-wf-cv-ref(<id>,env) =
      id∈Denv ∧ env(id) = cv


901 is-unique-ids(id-l) =

   type: Id* -> Bool


902 star(t) =
      /* all bds changed to * */

   type: Type -> Type


903 is-disjoint(s-l) =
      /* sets are pairwise disjoint */

   type: Set* -> Bool


904 is-scalar-type(t) = is-type(t) ∧ s-bds(t)=nil


905 is-array-type(t) = is-type(t) ∧ s-bds(t)≠nil
```

3.3 State

References to formulae of this section are of the form "DSn".

1 Σ :: stg: Stg
in : Val*
out: Val*

note: "stg" etc. are used throughout as references to state components.

2 Val = Intg | Bool

3 Stg = (Bool-loc -> (Bool | ?)) $\cup$ (Intg-loc -> (Intg | ?))

4 Env = Id -> (Loc | Cv-den | Lab-den | Proc-den)

5 Loc = Array-loc | Sc-loc

6 Array-loc = (Intg* <-> Bool-loc) | (Intg* <-> Intg-loc)
constraint: l∈Array-loc ⇒ (∃il∈Intg*)(Dl=rect(il))

7 Sc-loc = Bool-loc | Intg-loc

8 Cv-den = Intg

9 Lab-den :: Aid Id

10 Aid = infinite set

11 Proc-den :: (Loc | Proc-den)* Ca -> Tr

12 Ca = Aid-set

13 Tr = Σ -> Σ Abn

14 Abn = [Lab-den]

15 Intg-loc = infinite set

16 Bool-loc = infinite set
constraint: Bool-loc ∩ Intg-loc = {}

3.4 Functions

The numbering of the semantic functions is, as far as possible, that of the corresponding abstract syntax rules. References tc formulae of this section are of the form "DFn".

```
1  int-prog(<t>)(in-l) =
     let σ0 = <[ ],in-l,<>>;
     let σn = int-stmt(t,[ ],{})(σ0):
     out(σn)

   type: Prog -> (Val* -> Val*)


2  int-stmt(st,env,ca) =
     cases st:

     mk-block( , ) -> int-block(st,env,ca)

     mk-if(be,s1,s2) ->
        let v:eval-expr(be,env);
        if v then int-stmt(s1,env,ca) else int-stmt(s2,env,ca)

     mk-for(cv,e-i,e-s,e-l,b) ->
        let i:eval-expr(e-i,env),
            s:eval-expr(e-s,env),
            l:eval-expr(e-l,env);
        let f(x) = if (s>0 -> x≤l, s<0 -> x≥l, s=0 -> T)
                   then (int-stmt(b,env + [cv→x],ca);f(x+s));
        f(i)

     mk-call(pid,al) ->
        let mk-proc-den(pden) = env(pid),
            dl: <(is-var-ref(al[i]) -> eval-var-ref(al[i],env),
                  T                 -> env(al[i]))
                                         | i∈{1:len al}>;
        pden(dl,ca)

     mk-goto(id) -> exit(env(id))

     mk-assn(vr,e) ->
        let l:eval-var-ref(vr,env),
            v:eval-expr(e,env);
        assign(l,v)

     mk-in(vr) ->
        if c in = <> then error;
        let l:eval-var-ref(vr,env),
            v:hc in;
        if is-vmatch(v,vr-tp(vr))
           then (in := tc in, assign(l,v))
           else error

     mk-out(e) ->
        let v:eval-expr(e,env);
        out := c out^<v>

     null -> I

   type: Stmt Env Ca =>
```

```
3  int-block(<dcls,ns-l>,env,ca) =
      let aid∈Aid be s.t. aid∉ca;
      let n-env:env +
                ([id → eval-type(dcls(id),env) |
                     id∈Ddcls ∧ is-type°dcls(id) ] ∪
                 [id → eval-proc-dcl(dcls(id),n-env) |
                     id∈Ddcls ∧ is-proc°dcls(id) ] ∪
                 [id → mk-lab-den(aid,id) |
                     id∈col-st-nms(ns-l) ]);
      (trap exit (abn) with
         (epilogue({id∈Ddcls | is-type(dcls(id))},n-env);
          exit(abn));
       int-nmd-stmt-list(1,ns-l,n-env,ca ∪ {aid},aid));
      epilogue({id∈Ddcls | is-type(dcls(id))},n-env)

   type: Block Env Ca =>


31 epilogue(ids,env) =
      let loc-set = {env(id) | id∈ids};
      let sc-loc-set = {l∈loc-set | is-sc-loc(l)} ∪
                       union {Rl | l∈loc-set ∧ ¬is-sc-loc(l)};
      stg := c stg \ sc-loc-set

   type: Id-set Env =>


32 col-st-nms(ns-l) =
      {s-nm(ns-l[i]) | 1≤i≤lns-l ∧ is-id°s-nm(ns-l[i])}

   type: Nmd-st* -> Id-set


4  eval-type(tp,env) =
      trap exit(abn) with error;
      cases tp:
      mk-type(sc-tp,nil) ->
         let l∈Sc-loc be s.t. is-tp-sc-loc(sc-tp,l) ∧ l∉Dc stg;
         stg := c stg ∪ [l → ?];
         return (l)
      mk-type(sc-tp,bd-l) ->
         let ebd-l: <eval-expr(bd-l[i],env) | i∈{1:lbd-l}>;
         if (∃i ∈ {1:lebd-l})(ebd-l[i] < 1) then error;
         let l' be s.t. is-array-loc(l') ∧
                        Dl' = rect(ebd-l) ∧
                        l∈Rl' ⊃ is-tp-sc-loc(sc-tp,l) ∧
                        Rl' ∩ Dc stg = {};
         stg := c stg ∪ [l → ? | l∈R l'];
         return (l')

   type: Type Env => Loc


5  eval-proc-dcl(<parm-l,st>,env) =
      let f(den-l,ca) =
         (if ¬is-match(den-l,<s-attr°parm-l[i] | 1≤i≤lparm-l>)
            then error;
          let n-env = env +
                      [s-id°parm-l[i] → den-l[i] | i∈{1:lparm-l}];
          int-stmt(st,n-env,ca));
      result is (mk-proc-den(f))

   type: Proc Env -> Proc-den


51 is-match(dl,sl) =

   type:(Loc | Proc-den)* (Type | proc)* -> Bool
```

```
70 int-nmd-stmt-list(sno,ns-l,env,ca,aid) =
     trap exit(<taid,tid>) with
       if taid = aid
       then (let tno = (Un)(s-nm°ns-l[n] = tid);
             int-nmd-stmt-list(tno,ns-l,env,ca,aid))
       else exit(<taid,tid>);
     for j = sno to lns-l do
       int-stmt(s-stmt°ns-l[j],env,ca)

   type: Intg Nmd-st* Env Ca Aid =>


15 eval-expr(e,env) =
     cases e:
     mk-inf-expr(e1,op,e2) ->
       let v1:eval-expr(e1,env),
           v2:eval-expr(e2,env);
       apply-op(v1,op,v2)
     mk-rhs-ref(vr) ->
       let loc:eval-var-ref(vr,env);
       contents(loc)
     mk-cv-ref(id) ->
       return(env(id))
     T -> /* is-const(e) */
       return (val-of(e))

   type: Expr Env => Val


18 eval-var-ref(<id,ssc-l>,env) =
     if ssc-l = nil
       then return (env(id))
       else (let a-loc=env(id),
                 ev-ssc-l:<eval-expr(ssc-l[i],env) | 1≤i≤lssc-l>;
             if ¬(ev-ssc-l ∈ Da-lcc) then error;
             return (a-loc(ev-ssc-l)))

   type: Var-ref Env => Loc


20 val-of(c) =

   type: (Intg-const -> Intg) ∪ (Bocl-const -> Bool)


21 apply-op(v1,op,v2) =

   type: Val Op Val => Val


921 assign(l,v) =
      stg := c stg ↓ [l↦v]

   type: (Bool-loc Bool =>) ∪ (Intg-loc Intg =>)
   pre: l∈Dc stg, is-lmatch(l,v)


922 is-lmatch(l,v) =

   type: Sc-loc Val -> Bool


923 contents(l) =
      let v = c stg(l);
      if v = ? then error else return(v)

   type: (Bool-loc => Bool) ∪ (Intg-loc => Intg)
   pre: l∈Dc stg
```

```
924 rect(il) =

   type: Intg* -> (Intg*)-set
   pre: 1≤i≤⌊il ⇒ il[i]≥1


925 is-vmatch(v,t) =

   type: Val Sc-type -> Bool


926 is-tp-sc-loc(tp,l) =
      tp=bool -> is-bool-loc(l)
      tp=intg -> is-intg-loc(l)

   type: Sc-type Sc-loc -> Bool
```

4. MAP TO MACHINE STATES

4.1 Introduction

4.1.1 Implementation Technique

This section attempts to provide an introduction to the formal mapping by making a number of comments on the implementation.

Target Machine is 360 like i.e. Register; two address with some store to store operations; Address computation from displacement constant, contents of base and index registers; not all operations can use the most general form of address.

Optimisation in the normal sense is not attempted, some special cases (e.g. those prompted by the non-uniform address formats) are identified in order to avoid "poor code".

The problem of loading base registers for program addressing has been ignored: furthermore all procedures have been "compiled in line".

For the mapping given, an unbounded number of registers has been assumed (see section 2.2.3 for discussion on this point).

No mapping is given for the input/output statements of the language.

The basic design is modelled on that of ref 1, i.e. all storage is used as a stack; local areas (called here Dynamic save areas) are allocated, in this case, only one per procedure; a display is used to locate the dsa.

All storage for nested blocks is associated with the dsa of the embracing procedure.

Non-nested (brother) blocks will use the same storage for their variables.

The top of the stack pointer points to the next available byte and is held in a register whose number is given by the function prno().

The stack pointer value on entering a block n is stored in the dsa at stack-p(dsap,n).

The display is stored in registers; and an up-to-date copy is kept in the dsa.

Array base addresses (even for parameter arrays) are also kept in registers and comprise part of the display in the sense that they are subject to the same storage and retrieval.

Constants are treated as variables of a conceptual outer block.

Array pointers address the "0" element of an array.

An "implementation restriction" to one dimensional arrays, and the fact that subscript range checking is not performed, make it unnecessary to have array dope-vectors.

Integer entities are allocated 4 bytes, Booleans 1 byte.

Goto statements, even where blocks or procedures are terminated abnormally, are implemented by a direct branch (note label parameters are not in the source language).

The values required by for are also stored in registers, they have not however been treated as part of the display because of the attempt to postpone and if possible avoid storing them. This gives rise to the problem that the members of the work register stack being used for this purpose would differ depending on the place from which a procedure is called. In order to obviate this problem another implementation restriction is made which states that control variables are not known within procedures.

Work registers are used in a stack, the next element being obtained by the function "nwrno(r-set)".

Other "notes" can be found with the individual mapping functions.

4.1.2 Comments on Proof

In looking at the subsequent development and its justification it is useful to bear in mind the distinction between developing representations for objects and finding ways of modelling the required control structure. Arguments in the latter category are: modelling exit (cf. Section 2.2.2); simulating function application with call; simulating the (dynamic) conditional statements with Branch.

With regard to developing concrete representations of objects the simplest case is the development of state components of the source definition (e.g. using a linear byte addressed storage to simulate the mapping in DS3). A second area is the mapping of "run-time" arguments (e.g. Env, which depends on Stg) into the target machine store (in this example, partly over the registers containing the display: storage space is provided separately for parameter

addresses). There is also another way open to the definition to "hide" values in the sense that no commitment is made as to how they will be stored or addressed. The definition can simply use local variables (e.g. let d1 in call case of DF2). The development must provide storage for these in the target state unless special purpose machine operations exist.

It is a general thesis of the approach described in section 1, that dividing the overall development into smaller stages makes it easier to ascertain whether the whole is correct. The (temporary) separation of the dsa's from the actual storage was made precisely to make the reasoning easier. It is, in sections 4.2 and 4.4, relatively easy to show what is remaining constant in storage or what is being extended as distinct from the overwrite type updates to which the storage for variables is subjected. In section 5 it is shown how the dsa's can be placed in storage.

4.2 Dictionary and Machine States

References to formulae of this section are of the form "MSn".

Dictionary:

```
1  Dict = Id -> De

2  De = Prop-sc-de | Parm-sc-de |
       Prop-arr-de | Parm-arr-de |
       Cv-de |
       Lab-de |
       Prop-proc-de | Parm-proc-de

3  Prop-sc-de :: s-off:Intg s-base:Reg-no

4  Parm-sc-de :: s-parm-off:Intg s-parm-base:Reg-no

5  Prop-arr-de :: s-base:Reg-no

6  Parm-arr-de :: s-parm-off:Intg s-parm-base:Reg-no s-base:Reg-no

7  Cv-de :: Reg-no

8  Lab-de :: s-tgt:c-Label s-drno:Reg-no s-dp:Depth

9  Prop-proc-de :: s-lab:c-Label s-em-disp:Reg-no-set s-drno:Reg-no

10 Parm-proc-de :: s-parm-off:Intg s-parm-base:Reg-no

11 Depth = Intg

12 Reg-no   ) not further defined
13 c-Label  )

14 regs-of-dict(d) =
     /* yields those reg-no's used in de's */

   type: Dict -> Reg-no-set
```

Machine States:

```
21 m-Σ1 :: stg1 : Byte*
           regs : (Reg-no -> Byte*)
           env1 : (Dsa-p -> Dsa)
           cc   : Byte
           g-tp : (m-Sc-loc -> Sc-type)
           g-ai : (m-Array-loc -> Array-inf)
           g-pi : (c-Label -> Proc-inf)
```

note: non-overlap of storage for current D g-tp is not formally shown below.

```
22 Dsa-p = Intg

23 Dsa :: ra : c-Addr
          dc : Dsa-p
          sc : (Dsa-p | 0)
          pms : (m-Loc | m-Proc-den)*
          disp-copy : (Reg-no -> Byte*)
          stack-p : m-Sc-loc*

24 m-Loc = m-Sc-loc | m-Array-loc
```

Section: MAPPING

```
25 m-Sc-loc = Intg

26 m-Array-loc = Intg

27 m-Proc-den :: Dsa-p c-Label

28 Array-inf :: Sc-type Intg

29 Proc-inf :: s-em-rs:Reg-no-set s-em-inf:Em-inf

30 Em-inf :: s-em-gtp : (m-Sc-Loc -> Sc-type)
             s-em-env : (Dsa-p -> Dsa)
             s-em-ai : (m-Array-loc -> Array-inf)

31 Tr1 = m-Σ11 -> m-Σ11 Abn

32 m-Val = m-Bool | m-Intg

33 m-Bool = m-true | m-false
      constraint m-Bool ⊆ Byte

34 m-Intg = Byte⁴

35 c-Addr    not further defined except for:

36 addr(L) =

   type: c-Label -> c-Addr


Auxiliary Functions:

51 x-val(l,tp) =

      cases tp :
      bool -> c stg1[l]
      intg -> c stg1[l:l+3]

   type: m-Sc-Loc Sc-type -> (m-Σ11 -> m-Val)


52 set(l,v,tp) =

      cases tp:
      bool ->
         stg1 := <if i = l
                     then v
                     else c stg1[i] | i≤i≤l c stg1>
      intg ->
         stg1 := <if (i-l) ∈ {0:3}
                     then v[i-l+1]
                     else c stg1[i] | i≤i≤l c stg1>

   type: m-Sc-Loc m-Val Sc-type -> Tr1


Notation:

remembering:
   c sel                ~ λm-σ.(sel(m-σ))

then:
   r(rn)                ~ rn°regs
   r(p)                 ~ prno()°regs
   c(rn)                ~ c regs(rn)
   c(p)                 ~
   cs(rns)              ~ [rn → c(rn) | rn∈rns]

   r-ra(dsap)           ~ ra°dsap°env1
   c-ra(dsap)           ~ ra(c env1(dsap))

   r-pm(dsap,i)    ~ pms[i]°dsap°env1
   c-pm(      )    ~ (pms(c env1(dsap)))[i]

      etc for:
   - dc
```

- sc
- disp-copy
- stack-p(,i)

Notice that although register contents are of type m-Intg they are used as Intg without explicit conversion. Furthermore, the assembler language like format "d(x,b)" will be used to denote addresses:

where:
d is a constant expression
{x,b} ⊂ Reg-no
d(x,b) ~ c(x) + c(b) + d

```
61 save-disp(rns,dsap) =

     r-disp-copy(dsap) := c-disp-copy(dsap) † [rn → c(rn) | rn∈rns]

   type: Reg-no-set Dsa-p =>
```

```
62 rest-disp(rns,dsap) =

     for rn∈rns do
        r(rn) := c-disp-copy(dsap)(rn)

   type: Reg-no-set Dsa-p =>
```

4.3 Context Functions

References to formulae of this section will be of the form "MCn".

As in ref. 5, context functions are used as a way of deriving certain values which depend only on the static form of the program (see section 7 for comments on the scope of their use). Although it would have been possible to define the derivation of these values in the mapping functions, which in common with the defining functions mirror the structure of the text, it was considered worthwhile making the separation for the sake of readability. In particular it will, for most purposes, be sufficient to read the text outlining the role of the functions and rely on the, hopefully, suggestive names in studying the mapping itself.

The reference conventions of ref. 5 are also used here. Thus the values defined are really for instances of their arguments in context. The omission of the program as an additional argument is jutified by the possibility of adding all context values as nodes of the tree.

Firstly, some preliminary definitions:

```
1  is-contained(t1,t2) =

     /* is text t1 a part of t2:
        see ref. 5 for formal definition */
```

```
2  is-prop-cont(t1,t2) =

     is-contained(t1,t2) ∧ t1 ≠ t2
```

```
3  is-b-cont(t1,t2) =

     is-prop-cont(t1,t2) ∧
     ¬(∃t)(is-prop-cont(t1,t) ∧
           is-prop-cont(t,t2) ∧
           (is-block(t) ∨ is-proc(t)))
```

The number of the single unique register used as a current stack pointer is given by the function:

Section: MAPPING

8 prno() =

The display (cf. ref. 1) contains not only pointers to the areas of storage used for storage of link information and local variables (dsa = dynamic save area) but also base registers for addressing arrays. Thus the register numbers are allocated so that:

```
proc 1                  drno(proc1)    )                    )
   (arr1,               cx-arr-b(arr1)) cx-disp-regs(proc1))
    arr2;               cx-arr-b(arr2))                     ) cx-disp-regs(proc2)
      proc2             drno(proc2)                         )
         (arr3;         cx-arr-b(arr3)                      )
           .
           .
        )
      .
      .
   )
```

In describing this ordering, the following function is used:

5 above(rs1,rs2) =

```
/* each member of rs2 is a higher register number
   than any member of rs1 */
```

type: Reg-no-set Reg-no-set -> Bool

The functions are now defined by stating properties which must be satisfied by their results.

6 cx-drno

$$\begin{aligned}
&\text{is-prog}(t) \Rightarrow \text{cx-drno}(t) = \underline{\text{drno-0}}\\
&(\text{is-prog}(t) \vee \text{is-proc}(t) \vee \text{is-block}(t)) \wedge \text{is-b-cont}(t_1,t) \wedge\\
&\quad \neg\text{is-proc}(t_1) \Rightarrow\\
&\qquad \text{cx-drno}(t_1) = \text{cx-drno}(t)\\
&\text{is-block}(t) \wedge \text{is-b-cont}(t_1,t) \wedge\\
&\quad \text{is-proc}(t_1) \Rightarrow\\
&\qquad \text{above}(\text{cx-disp-regs}(t), \{\text{cx-drno}(t_1)\})
\end{aligned}$$

type: Π -> Reg-no
note: applicable to any part of text

7 cx-arr-b

$$\begin{aligned}
&\text{is-proc}(t_1) \Rightarrow\\
&\quad (\underline{\text{let}} \langle \text{parm-1}, \rangle = t_1;\\
&\quad \underline{\text{let}}\ \text{ids} = \{\text{s-id}(\text{parm-1}[i]) \mid 1 \le i \le \underline{1}\ \text{parm-1} \wedge \text{is-array-type}^{\circ}\text{s-attr}(\text{parm-1}[i])\};\\
&\quad (\forall id \in \text{ids})(\text{above}(\{\text{cx-drno}(t_1)\}, \{\text{cx-arr-b}(id)\})) \wedge\\
&\quad (\forall id_1, id_2 \in \text{ids})(\text{cx-arr-b}(id_1) = \text{cx-arr-b}(id_2) \Rightarrow id_1 = id_2))\\
&(\text{is-prog}(t) \vee \text{is-proc}(t) \vee \text{is-block}(t)) \wedge \text{is-b-cont}(t_1,t) \wedge\\
&\quad \text{is-block}(t_1) \Rightarrow\\
&\qquad (\underline{\text{let}}\ \text{ids} = \{id \in \underline{D}\ \text{s-dcls}(t_1) \mid \text{is-array-type}(\text{s-dcls}(t_1)(id))\};\\
&\qquad (\forall id \in \text{ids})(\text{above}(\text{cx-disp-regs}(t), \{\text{cx-arr-b}(id)\})) \wedge\\
&\qquad (\forall id_1, id_2 \in \text{ids})(\text{cx-arr-b}(id_1) = \text{cx-arr-b}(id_2) \Rightarrow id_1 = id_2))
\end{aligned}$$

type: Id -> Reg-no

8 cx-disp-regs

```
is-prog(t) ⊃ cx-disp-regs(t) = {cx-drno(t)}
(is-prog(t) ∨ is-proc(t) ∨ is-block(t)) ∧ is-b-cont(t1,t) ∧
   is-block(t1) ⊃
      cx-disp-regs(t1) = cx-disp-regs(t) ∪
                         {cx-arr-b(t1,id) | id∈D s-dcls(t1) ∧
                                            is-array-type(s-dcls(t1)(id))}
is-block(t) ∧ is-b-cont(t1,t) ∧
   is-proc(t1) ⊃
      cx-disp-regs(t1) = cx-disp-regs(t) ∪ {cx-drno(t1)} ∪
                                           {cx-arr-b(t1,id) | id∈...}
(is-prog(t) ∨ is-proc(t) ∨ is-block(t)) ∧ is-b-cont(t1,t) ∧
   ¬(is-proc(t1) ∨ is-block(t1)) ⊃
      cx-disp-regs(t1) = cx-disp-regs(t)
```

type: Π -> Reg-no-set

9 cx-em-disp-regs(w) =

/* similar to MC8, but is applied to a block and yields the set relevent to its embracing text unit */

10 cx-env-regs(w) =

/* similar to MC8, but also includes the numbers of those work registers allocated to the local quantities of for loops of MF9 */

11 cx-em-env-regs(w) =

/* similar to MC9, but ... */

Within a procedure (remember that dsa's are not allocated for separate blocks within a procedure cf. Section 4.1.1) the offsets of scalar quantities can be allocated statically. The obvious constraints of non-overlapping etc. are described below:

12 cx-sc-off

```
is-block(t1) ⊃
   (let <dcls,>=t1;
    let em-offs = cx-em-sc-offs(t1),
        ids = {id∈D dcls | is-scalar-type°dcls(id)};
   (∀o∈em-offs,id∈ids)
      {o<cx-sc-off(id) ∧ ¬overlap(o,cx-sc-off(id))}
   (∀id1,id2∈ids)
      (s-sc-type°dcls(id1) = intg ⊃ mod(cx-sc-off(id2),4) = 0 ∧
       overlap(cx-sc-off(id1),cx-sc-off(id2)) ⊃ id1 = id2))
```

type: Id -> Intg
note: allignment handled.

An analogous function exists for constants:

13 cx-const-off(c) =

type: Const -> Intg

Section: MAPPING

14 cx-em-sc-offs

```
is-prog(t) ∨ is-proc(t) ⊃ cx-em-sc-offs(t) = {}
(is-prog(t) ∨ is-proc(t)) ∧ is-b-cont(t,t1) ⊃
   cx-em-sc-offs(t1) = {}
is-block(t) ∧ is-b-cont(t,t1) ∧
   ¬is-proc(t1) ⊃
      cx-em-sc-offs(t1) = cx-em-sc-offs(t) ∪
                          {cx-sc-off(id) | id∈D s-dcls(t) ∧
                                           is-scalar-type(s-dcls(t)(id))}
```

type: N -> Intg-set

Within a procedure the static depth of blocks is given by:

15 cx-dp

```
(is-proc(t) ∨ is-prog(t)) ⊃ cx-dp(t) = 0
(is-prog(t) ∨ is-proc(t) ∨ is-block(t)) ∧ is-b-cont(t1,t) ∧
   is-block(t1) ⊃
      cx-dp(t1) = cx-dp(t) + 1
(is-prog(t) ∨ is-proc(t) ∨ is-block(t)) ∧ is-b-cont(t1,t) ∧
   ¬(is-proc(t1) ∨ is-block(t1)) ⊃
      cx-dp(t1) = cx-dp(t)
```

type: N -> Depth

Because the names of procedures and/or labels can be repeated in the source program, context functions are used to generate the labels used in the code.

16 cx-proc-lab(id) =

type: Id -> c-Label

17 cx-nm-lab(id) =

type: Id -> c-Label
note: the ranges of cx-proc-lab and cx-nm-lab must be disjoint.

18 retr-nm(lab) =

/* inverse of cx-nm-lab */

type: c-Label -> Id

The ordering of the work registers is given by:

19 nwrno(rns) =

/* delivers r∉rns */

type: Reg-no-set -> Reg-no

4.4 Proof Notes

References to formulae of this section are of the form "Pn".

The relationship between objects of the definition and objects in the machine state is defined below. The classes Sc-loc and Aid of the source definition were only constrained by certain properties: no explicit construction was given. In these cases it is shown how aspects of the machine state model the objects in the sense that they satisfy all of the stated properties.

Thus (cf DS15, DS16):

1 Bool-loc :: Intg

2 Intg-loc :: Intg

Note they are disjoint because of the constructors.

```
3  make-tp-sc-loc(tp,m-loc) =

       cases tp:
       bool -> mk-bool-loc(m-loc)
       intg -> mk-intg-loc(m-loc)

   type: Sc-type m-Sc-loc -> Sc-loc
```

And (cf. DS10):

4 Aid :: Dsa-p Depth

The functions which define equivalence can now be given:

```
5  in-step(dict,env,ca,drno)(m-σ,σ) =

       in-step1(dict,env)(m-σ,σ) ^
       q-ca(ca,drno)(m-σ)


6  in-step1(dict,env)(m-σ,σ) =

       in-step-Σ(m-σ,σ) ^
       q-env(dict,env)(m-σ)


7  q-ca(ca,drno) =

       <dsap,dp>∈ca ⇒ (dsap ∈ D̲ c̲ env1 ^
                       dsap=c(drno) ⇒ dp≤lc-stack-p(c(drno)))

   type: Aid-set Reg-no -> (m-Σ1 -> Bool)


8  in-step-Σ(m-σ,σ) = σ = retr-Σ(m-σ)


9  retr-Σ =

       <[ make-tp-sc-loc(tp,l) → retr-val(x-val(l,tp)) | [l → tp]∈c̲ g-tp̲], <>, <>>

   type: m-Σ1 -> Σ


10 in-step-a()(<m-σ,m-a>,<σ,a>) =

       m-a = a ^
       in-step-Σ(m-σ,σ)


11 q-ca2(ca) =

       <dsap,dp>∈ca ⇒ (dsap∈(D̲ c̲ env1 \ {c(p̲)}))

   type: Aid-set -> (m-Σ1 -> Bool)


12 q-env(dict,env) =

       D̲ dict = D̲ env ^
       id∈D̲ dict ⇒ q-den(dict(id),env(id))

   type: Dict Env -> (m-Σ1 -> Bool)
```

Section: MAPPING

```
13 q-den(de,den) =

     cases de:
     mk-cv-de(r) -> retr-val(c(r)) = den
     mk-lab-de() -> retr-lab-den(de) = den
     mk-prop-proc-de(L,,em-drno) ->
        q-proc-den(<c(em-drno),L>,den,c q-pi>)
     mk-parm-proc-de(d,b) ->
        q-proc-den(c-pm(c(b),d),den,c q-pi)
     T -> retr-loc(de) = den

   type: De Den -> (m-Σ1 -> Bool)


14 retr-lab-den(<lab,drno,dp>) =

     mk-lab-den(mk-aid(c(drno),dp),retr-nm(lab))

   type: Lab-de -> (m-Σ1 -> Lab-den)


15 retr-loc(de) =

     cases de:
     mk-prop-sc-de(d,b) ->
        (let l = d(0,b);
         make-tp-sc-loc(c q-tp(l),l))
     mk-parm-sc-de(d,b) ->
        (let l = c-pm(c(b),d);
         make-tp-sc-loc(c q-tp(l),l))
     mk-prop-arr-de(b) ^
     mk-parm-arr-de(,,b) ->
        make-array-loc(c q-ai(c(b)),c(b))

   type: Var-de -> (m-Σ1 -> Loc)


16 make-array-loc(<tp,bd>,base) =

     [<i> -> make-tp-sc-loc(tp,m-sub-loc(base,i,tp)) | 1≤i≤bd]

   type: Array-inf m-Array-loc -> Array-Loc


17 m-sub-loc(b,i,tp) =

     b + (if tp = bool then i else 4*i)

   type: m-Array-loc Intg Sc-type -> m-Sc-Loc


18 q-proc-den(<dsap,L>,<f>,cgpi) =

     let em-rs = s-em-rs(cgpi(L));
     /* the following "annotation" should be valid (arbitrary m-σ) */
              ! pre-q-proc-den(cgpi,dsap,em-rs)
              !! q-ca2(ca)
              !! in-step-Σ()
              !! q-arg-1(c-pms(c(p)),d1)
     call L(r-ra(c(p)))
              !! f(d1,ca)
              !! in-step-a()
              ! post-q-proc-den(dsap,em-rs)

   type: m-Proc-den Proc-den (c-Label -> Proc-inf) -> Bool
   note: post-q-proc-den also used to ensure preservation of c(p)
```

19 pre-q-proc-den(cgpi,dsap,em-rs) =

```
cons-p-reg(c(p),D ⊆ q-tp) ^
cgpi ⊆ c q-pi ^
cons-gpi(R ⊆ q-pi, <c q-tp,c env1,c q-ai>) ^
c-sc(c(p)) = 0 ⇒ cons-disp-copy(c-disp-copy(dsap),cs(em-rs)) ^
c-sc(c(p)) ≠ 0 ⇒ c-sc(c(p)) = dsap
```

type: (c-Label -> Proc-inf) Dsa-p Reg-no-set -> (m-Σ1 -> Bool)

20 post-q-proc-den(dsap,em-rs) =

```
cons-disp-copy(c-disp-copy(dsap),em-rs) ^
pres-regs({prno()}) ^
pres-gtp ^ pres-gai ^ pres-gpi ^
almost-pres-env(c(p))
```

type: Dsa-p Reg-no-set -> (m-Σ1 m-Σ1 -> Bool)

21 almost-pres-env(p)(m-σ,m-σ') =

```
env1(m-σ') \ p = env1(m-σ) \ p ^
env1(m-σ)(p) ⊆° env1(m-σ')(p)
```

note: ⊆° tests equality except permits extension of disp-copy or stack-p

22 q-arg-l(m-den-l,den-l) =

```
l m-den-l = l den-l ^
1≤i≤lm-den-l ⇒ q-arg(m-den-l[i],den-l[i])
```

23 q-arg(m-den,den) =

```
is-sc-loc(den) ->
   make-tp-sc-loc(c q-tp(m-den),m-den) = den
is-array-loc(den) ->
   make-array-loc(c q-ai(m-den),m-den) = den
is-proc-den(den) ->
   q-proc-den(m-den,den,c q-pi)
```

type: (m-Loc | m-Proc-den)(Loc | Proc-den) -> (m-Σ1 -> Bool)

The appropriate relation for values is:

31 retr-val(m-v) =

type: m-Val -> Val

The notion of consistency arises for states of m-Σ11:

41 cons-m-Σ1(dict,drno,rns,sc-offs) =

```
cons-vi(c(drno),c-stack-p(c(drno),1),c(p),sc-offs,D ⊆ q-tp) ^
cons-pi(dict,c q-pi,c(drno),cs(rns),c q-tp, c env1, c q-ai)
```

type: Dict Reg-no Reg-no-set Intg-set -> (m-Σ1 -> Bool)
note: if p* = <>,c(p) used for c-stack-p (informal)

42 cons-vi(dsap,p-ptr$_1$,p-ptr,sc-offs,ls) =

```
cons-p-reg(p-ptr,ls) ^
cons-l-scs(dsap,l-scs(ls,dsap,p-ptr1),sc-offs) ^
cons-sc-offs(dsap,p-ptr1,sc-offs)
```

Section: MAPPING

43 cons-pi(dict,cgpi,dsap,rm,cgtp,cenv,cgai) =

cons-disp-copy(disp-copy(cenv(dsap)),rm) ∧
cons-gpi(R cgpi,<cgtp,cenv,cgai>) ∧
cons-kn-prs(dict,cenv,cgpi,rm)

44 l-scs(ls,dsap,p-ptr$_1$) = {l∈ls | dsap≤l<p-ptr$_1$}

45 cons-p-reg(p-ptr,ls) = l∈ls ⊃ l < p-ptr

46 cons-l-scs(dsap,lls,sc-offs) = l∈lls ⊃ (l - dsap)∈sc-offs

47 cons-sc-offs(dsap,p-ptr$_1$,sc-offs) = o∈sc-offs ⊃ (dsap + o)<p-ptr$_1$

48 cons-disp-copy(dc,rm) = dc ⊆ rm

49 cons-gpi(pi-set,cur-inf) = pi∈pi-set ⊃ s-em-inf(pi) ⊆* cur-inf

50 cons-kn-prs(dict,cenv,gpi,rm) =

mk-prop-proc-de(L,em-rs,em-drno) ∈ Rdict ⊃
(s-em-rs(gpi(L)) = em-rs ∧
cons-disp-copy(disp-copy(cenv(rm(em-drno))),rm))

61 pres(rns) =

pres-regs(rns) ∧
pres-env ∧ pres-gtp ∧ pres-gai ∧ pres-gpi

type: Reg-no-set -> (m-Σ1 m-Σ1 -> Bool)

62 pres-regs(rns)(m-σ,m-σ') =

regs(m-σ')|rns = regs(m-σ)|rns

type: Reg-no-set -> (m-Σ1 m-Σ1 -> Bcol)

63 pres-α(m-σ,m-σ') = α(m-σ) = α(m-σ')

64 ext-α(m-σ,m-σ') = α(m-σ) ⊆ α(m-σ')

65 ext(m-σ,m-σ') =

ext-α(m-σ,m-σ') for all components α

66 pt-ext-env(m-σ,m-σ') =

p∈Denv1(m-σ) ⊃ env1(m-σ)(p) ⊆* env1(m-σ')(p)

Notice that the preservation of constants in the state is not proven formally.

4.5 Some Initial Lemmas

References to these Lemmas are of the form "Ln".

These results show that under constrained changes consistency of a machine state or compatibility of a source/machine state pair are preserved.

Lemma 1

for all fns m-i-θ(w,dict,...)
regs-of-dict(dict) ≤ cx-em-env-regs(w)

from context functions and formation of Dict

Lemma 2

mk-prop-proc-de(,em-rs,) ∈ R dict(w) ⊃
em-rs ≤ cx-disp-regs(w)

from MP3 and by inspection

Lemma 4

cons-pi(dict,cgpi,dsap,rm,cgtp,cenv,cgai) ∧
cgtp ≤ cgtp' ∧
cgai ≤ cgai' ⊃
cons-pi(dict,cgpi,dsap,rm,cgtp',cenv,cgai')

from P43, P48, P49, P50

Lemma 5

cons-m-Σ1(dict,drno,rns,sc-offs)(m-σ) ∧
pres({prno(),drno} ∪ rns)(m-σ,m-σ') ⊃
cons-m-Σ1(dict,drno,rns,sc-offs)(m-σ')

from L4, P41, P42, P44-47

Lemma 6 (similarly L7 for q-arg-1)

q-arg(mden,den)(m-σ) ∧
ext-gai(m-σ,m-σ') ∧
ext-gtp(m-σ,m-σ') ∧
pres-gpi(m-σ,m-σ') ⊃
q-arg(mden,den)(m-σ')

immediate from P23

Lemma 8

q-proc-den(mpden,pden,cgpi) ∧
cgpi ≤ cpgi' ⊃
q-proc-den(mpden,pden,cgpi')

immediate from P18
(note class of states over which comparison performed is reduced)

Lemma 9

¬is-proc-de(de) ∧
q-den(de,den)(m-σ) ∧
pres-regs(regs-of-de(de))(m-σ,m-σ') ∧
ext-gtp(m-σ,m-σ') ∧
ext-gai(m-σ,m-σ') ∧
pt-ext-env(m-σ,m-σ') ⊃
q-den(de,den)(m-σ')

from P13, P14-17

Section: MAPPING

Lemma 10

```
q-env(dict,env)(m-σ) ^
pres-regs(regs-of-dict(dict))(m-σ,m-σ') ^
ext(m-σ,m-σ') ⊃
   q-env(dict,env)(m-σ')
```

from L8, L9, P12, P13

Lemma 11

```
q-ca(ca,drno)(m-σ) ^
ext-env-pt(m-σ,m-σ') ^
pres-regs({drno})(m-σ,m-σ') ⊃
   q-ca(ca,drno)(m-σ')
```

immediate from P7

Lemma 12

```
drno,disp-regs = cx-α(w) ^
in-step(dict,env,ca,drno)(m-σ,σ) ^
pres(rs ∪ disp-regs ∪ {prno()})(m-σ,m-σ') ⊃
in-step-a()(<m-σ',m-a'>,<σ',a'>) ⊃
   in-step(dict,env,ca,drno)(m-σ',σ')
```

proof from
L1 plus assumption (not formally proven) that wrs are in rs
i.e.
cx-env-regs(w) - cx-disp-regs(w) ⊆ rs
P5, P6, P10, L10, L11

4.6 Maps

References to the specifications within this section are of the form "MFn".

Equivalence of static types with those in the dict is not formally proven (cf. Ref [8]).

m-int-prog

spec:

```
m-int-prog(prog)
                !! int-prog(prog)(<>)
                !! in-step-Σ()
```

notes:
Without the input and output statements one could ask: Why bother to go further?

map:

```
1  m-int-prog(<prog>) =

                !! 1 stg := [ ]
   env1 := [ ];
                !  2 q-tp := [ ]
                !  3 q-ai := [ ]
                !  4 q-pi := [ ]
   r(drno-0) := GETMAIN;
   set-constants();
   r(p) := c(drno-0);
   r-disp-copy(c(drno-0)) := [ ];
                !  5 cons-m-Σ1([ ],drno-0,{drno-0},{})          P41-P50
                !! 6 in-step([ ],[ ],{},drno-0)               P5-9,P12
   m-int-stmt(prog,[ ],{});
                !! 7 int-stmt(prog,[ ],{})                    ~
                !! 8 in-step-Σ()                             5,6,7,MF2,P10
                !  post                                      DF1,1,7

type: Prog =>
```

m-int-stmt

spec:

```
let drno,disp-regs,sc-offs = cx-α(t)
                ! cons-m-Σ1(dict,drno,disp-regs,sc-offs)
                !! in-step(dict,env,ca,drno)
m-int-stmt(t,dict,rs)
                !! int-stmt(t,env,ca)
                !! in-step-a()
                ! pres(rs ∪ disp-regs ∪ {prno()})
```

map:

```
2  m-int-stmt(t,dict,rs) ≡

                ! 1 cons-m-Σ1                                    pre
                !! 2 in-step                                     pre
                !! 3 int-stmt(t,env,ca)                          ≈
     (is-block(t) -> m-int-block(t,dict,rs)
      is-if(t)    -> m-int-if(t,dict,rs)
      is-for(t)   -> m-int-for(t,dict,rs)
      is-call(t)  -> m-int-call(t,dict,rs)
      is-goto(t)  -> m-int-goto1(t,dict,rs)
      is-assn(t)  -> m-int-assn(t,dict,rs)
      is-null(t)  -> I
     )
                !! 4 in-step-a()                                  1,2,MF2,MF8-12
                ! 5 pres(rs ∪ disp-regs ∪ {prno()})              1,2,MF2,MF8-12
                ! post                                           DF2,3

  type: Stmt Dict Reg-no-set =>
```

m-int-block

spec:

```
let drno,disp-regs,sc-offs = cx-em-α(bl)
                ! cons-m-Σ1(dict,drno,disp-regs,sc-offs)
                !! in-step(dict,env,ca,drno)
m-int-block(bl,dict,rs)
                !! int-block(bl,env,ca)
                !! in-step-a()
                ! pres(rs ∪ disp-regs ∪ {prno()})
```

map:

```
3 m-int-block(m,dict,rs) =

    let <dcls,ns-1> = m;
    let drno   = cx-drno(m),
        dp     = cx-dp(m),
        em-rns  = cx-em-disp-regs(m),
        em-offs = cx-em-sc-offs(m),
        arr-rns = {cx-arr-b(id) | id∈D dcls ∧ is-array-type°dcls(id)},
        l-offs  = {cx-sc-off(id) | id∈D dcls ∧ is-scalar-type°dcls(id)};
    let drns = em-rns ∪ arr-rns,
        offs = em-offs ∪ l-offs;

                ! 1 cons-m-Σ1(dict,drno,em-rns,em-offs)                pre
                !! 2 in-step1(dict,env)                                pre,P5
                !! 3 q-ca(ca,drno)                                     pre,P5
                !! 4 let m-σ1 be current
    r-stack-p(c(drno),dp) := c(p);
                ! 5 let m-aid = <c(drno),dp>                           ~
                !! 6 m-aid∉ca                                          3,5,L=dp
                !! 7 q-ca(ca ∪ {m-aid},drno)                          3,5,P7
    dcl n-dict := dict † [id→<cx-nm-lab(id),drno,dp> | id∈col-st-nms(ns-1)];
                !! 8 dcl n-env:=env † [id→<m-aid,id>|id∈...]          ~
                !!10 in-step1(c n-dict,c n-env)                        2,L10,5,8,P14,MC18
                ! 11 cons-p-reg(c(p),D c q-tp)                        1,P41
                ! 12 cons-l-scs(c(drno),l-scs(...),em-offs)           1,P41
                ! 13 disj-l-offs(em-offs,l-offs)                      MC12
                ! 14 disj-arr-bs(em-rns,arr-rns)                      MC7
                ! 15 regs-of-dict(dict)⊆cx-em-env-regs(m)             L1
                ! 16 cons-sc-offs(c(drno),c-stack-p(c(drno),1),em-offs)
                                                                       1,P41
    let type-dcls = {id∈D dcls | is-type°dcls(id)},
        proc-dcls = {id∈D dcls | is-proc°dcls(id)};

    for id∈type-dcls do
       (let t = dcls(id);
                ! 17 cons-p-reg(c(p),D c stg)                         AL2 hyp
                ! 18 cons-sc-offs(c(drno),c-stack-p(c(drno),1),l-offs)
                                                                       AL2 hyp
                ! 19 s-bds(t)=nil ⇒ cx-sc-off(id)(0,drno)∉D c q-tp    P46,13,12
                ! 20 s-bds(t)≠nil ⇒ cx-arr-b(id)∉regs-of-dict(dict)   14,15
                !!21 in-step1(dict,env)                               AL2 hyp
        let de:m-eval-type(t,id,dict,rs);
                !!22 n-env:=c n-env † [id→eval-type(t,env)]           ~
                !!23 in-step-a()                                       17-22,MF4
                !!24 q-den(de,c n-env(id))                             17-22,MF4
                ! 25 post1-m-eval-type(regs-of-dict(dict) ∪ rs)       17-21,MF4
                ! 26 s-bds(t)=nil ⇒ cons-l-sc(c(drno),cx-sc-off(id))  17-21,MF4
        n-dict := c n-dict † [id→de]
                !!27 in-step1(dict,env)                                21,23,25,L10
                ! 28 cons-p-reg(c(p),D c q-tp)                        17,25,MF41,P45
       );
                !!29 in-step1(c n-dict,c n-env)                        AL2,10,27,22,24
                ! 30 cons-gpi(R c q-pi,<c q-tp,c env1,c q-ai>)         1,25,P49
                ! 31 cons-disp-copy(c-disp-copy(c(drno)),rs(em-rns))   1,25,P48
                ! 32 cons-p-reg(c(p),D c q-tp)                        AL2,11,28
                ! 33 cons-l-scs(c(drno),l-scs(...),offs)              12,26
                ! 34 cons-sc-offs(c(drno),c-stack-p(c(drno),1),offs)  16,25
```

```
      save-disp(arr-rns,c(drno));
                ! 35 cons-disp-copy(c-disp-copy(c(drno)),cs(drns))    31,MS61
                ! 36 cons-kn-prs(c n-dict, c env1, c q-pi,cs(drns))  1,25,35
                ! 39 let m-σ2 be current

      for id∈proc-dcls do
         (n-dict := c n-dict + [id→<cx-proc-lab(id),drns,drno>]
                ! 40 let cur-inf=<c q-tp,c env1,c q-ai>
                ! 41 q-pi:=c q-pi ∪ [cx-proc-lab(id)→<drns,cur-inf>]
                ! 42 cons-gpi(R c q-pi,cur-inf)                       30,41,P49
                ! 43 cons-kn-prs(c n-dict,c env1,c q-pi,cs(drns))     35,36,P50
         );
                !!44 let nn-env=c n-env +
                                  [id→eval-proc-dcl(dcls(id),nn-env)|id∈proc-dcls]  ~
                !!45 (∀id∈proc-dcls)(q-proc-den(<c(drno),cx-proc-lab(id)>,
                                        nn-env(id),c q-pi))          47 (ref time)
                !!46 q-env(c n-dict,nn-env)                          29,45,P13

      branch INTB;
      for id∈proc-dcls do
         (cx-proc-lab(id):
          m-eval-proc-dcl(dcls(id),c n-dict,drns)
                !!47 q-proc-den(by ref time)                         43,41,46,MP5
         );

      INTB:
                !!50 in-step(c n-dict,nn-env,ca ∪ {m-aid},drno)       46,29,7,P5
                ! 51 cons-m-Σ1(c n-dict,drno,drns,offs)              32-38,42,43,P41
      ((trap exit (abn) with
         (      ! 52 q-pi:=c q-pi \ proc-dcls
                ! 53 pres(rs ∪ drns ∪ {prno()})(m-σ2,)               AL3hyp
                !!54 in-step1(c n-dict,nn-env)                       AL3hyp
          m-epilogue(type-dcls,c n-dict,drno,dp)
                !!55 epilogue(type-dcls,nn-env)                      ~
                ! 56 pres(rs ∪ em-rns ∪ {prno()})(m-σ1,)             53,MF31,4,39
                !!57 in-step-a()                                     50,53,54,L12,MF31
          exit (abn)
         );
        m-int-nmd-stmt-list1(1,ns-1,c n-dict,rs);
                !!58 int-nmd-stmt-list(1,ns-1,nn-env,ca ∪ {m-aid},m-aid)   ~
                !!59 in-step-a()                                     50,51,58,MF70
                ! 60 pres(rs ∪ drns ∪ {prno()})(m-σ2,)              50,51,MF70
       );
                ! 61 q-pi:=c q-pi \ proc-dcls
                ! 62 pres(rs ∪ drns ∪ {prno()})(m-σ2,)               AL3hyp
                !!63 in-step-a()                                     AL3hyp
       m-epilogue(type-dcls,c n-dict,drno,dp)
                !!64 epilogue(type-dcls,nn-env)                      ~
                ! 65 pres(rs ∪ em-rns ∪ {prno()})(m-σ1,)             62,MF31,4,39
                !!66 in-step-a()                                     50,62,63,L12,MF31
      )
                !!67 in-step-a()                                     AL3,57,66
                ! 68 pres(rs ∪ em-rns ∪ {prno()})(m-σ2,)            AL3,56,65
                !!post                                               DF3,5,6,8,22,44,55,58,64
```

```
   type: Block Dict Reg-no-set =>


31 disj-l-offs(em-offs,l-offs) = /* no overlaps */

   type: Intg-set Intg-set -> Bool


32 disj-arr-bs(em-rs,arr-rns) =

      em-rs ∩ arr-rns = {}

   type: Reg-no-set Reg-no-set -> Bool
```

Section: MAPPING

m-epilogue

spec:

```
                !! in-step1(dict,env)
  m-epilogue(ids,dict,drno,dp)
                !! epilogue(ids,env)
                !! in-step-a()
```

furthermore, m-epilogue should act as an inverse of the actions taken for processing the declarations:

```
  m-σ2 = (r-stack-p(c(drno),dp) := c(r);
          for id∈type-dcls do
             (let t = dcls(id);
              let de:m-eval-type(t,id,dict,rs);
              n-dict:=c n-dict + [id→de]
             )
         )(m-σ1) ∧
  pres(rs ∪ cx-disp-regs(π) ∪ {prno()})(m-σ2,m-σ3) ∧
  m-σ4 = m-epilogue(type-dcls,cn-dict,drno,dp)(m-σ3) ⊃
     pres(rs ∪ cx-em-disp-regs(π) ∪ {prno()})(m-σ1,m-σ4)
```

map:

```
31 m-epilogue(ids,dict,drno,dp) =

                !! 1 in-step-Σ()                                  pre
    for id∈ids do
                !! 2 in-step-Σ()                                  AL2hyp
       (cases dict(id):
                !! 3 q-den(dict(id),env(id))                      pre,P12
        mk-prop-sc-de(d,b) ->
          (q-tp := c q-tp \ {d(0,b)};
                !! 4 stg := c stg \ env(id)                       ~
                !! 5 in-step-Σ()                                  2,3,4,P9,P15
          )
        mk-prop-arr-de(b) ->
          (let <tp,bd> = q-ai(c(b));
           q-tp := c q-tp \ {(tp=intg->4*i, T->i)(0,b) | 1≤i≤bd};
                !! 6 stg := c stg \ R(env(id))                    ~
                !! 7 in-step-Σ()                                  2,3,6,P9,P15
           q-ai := c q-ai \ c(b)
          )
       );
                !! 8 in-step-Σ()                                  AL2,1,5,7
    r(p) := c-stack-p(c(drno),dp)
                !! 9 inverse effect                               MF4, by inspection
                !!10 post                                         DF31,4,6

  type: Id-set Dict Reg-no Depth =>
```

m-eval-type

spec:

```
  let drno = cx-drno(t);
                !  cons-p-reg(c(p),D c q-tp)
                !  s-bds(t)=nil ⊃ (cx-sc-off(id)(0,drno)∉D c q-tp ∧
                                   cons-sc-offs(c(drno),c-stack-p(c(drno),1),cx-sc-off(id))
                !  s-bds(t)≠nil ⊃ cx-arr-b(id)∉regs-of-dict(dict)
                !! in-step1(dict,env)
  let de:m-eval-type(t,id,dict,rs)
                !! let l:eval-type(t,env)
                !! in-step-a()
                !! q-den(de,l)
                !  post1-m-eval-type(regs-of-dict(dict) ∪ rs)
                !  s-bds(t)=nil ⊃ cons-l-sc(c(drno),cx-sc-off(id))
```

```
41 post1-m-eval-type(rs) =

    pres-env ∧ pres-pi ∧ pres-regs(rs) ∧
    ext-gai ∧ ext-gtp ∧
    cons-gtp-ext

  type: Reg-no-set -> (m-Σ1 m-Σ1 -> Bool)
```

```
42 cons-gtp-ext(m-σ,m-σ') =
      cons-p-reg(regs(m-σ')(p),D g-tp(m-σ') - D g-tp(m-σ))

43 cons-l-sc(dsap,sc-off)(m-σ,m-σ') =
      cons-l-scs(dsap,D g-tp(m-σ') - D g-tp(m-σ),{sc-off})
   type: Dsa-p Intg -> (m-Σ1 m-Σ1 -> Bool)
```

```
map:
4  m-eval-type(t,id,dict,rs) =

                ! 1 cons-p-reg(c(p),D c g-tp)                          pre
                !! 2 in-step1(dict,env)                                pre
     trap exit (abn) with error;
     cases t:
     mk-type(sc-tp,nil) ->
        (let d = cx-sc-off(id),b = cx-drno(t);
         let de = mk-prop-sc-de(<d,b>);
         let l' = d(0,b);
                ! 3.1 cons-sc-offs(c(b),c-stack-p(c(b),1),{d})         pre,3
                !! 3.2 l'∉D c g-tp                                     pre,3
                !! 3.3 let l = make-tp-sc-loc(sc-tp,l')                ~
                !! 3.4 l∉D c stg                                       3.2,3.3,2,P9
                !! 3.5 is-tp-sc-loc(sc-tp,l)                           3.3, P3
                !! 3.6 l satisfies properties of Source                DP4,3.4,3.5
                ! 3.7 g-tp := c g-tp ∪ [l'→sc-tp]
                !! 3.8 q-den(de,l)                                     3.3,P15
                !! 3.9 stg := c stg ∪ [l→?]                            ~
         set(l',m-?,sc-tp);
                !! 3.10 in-step-a()                                    2,3.7,3.9,P9
                ! 3.11 cons-p-reg(c(p),{l'})                           3.1,p(1)≤p(i)
                ! 3.12 post1-m-eval-type(regs-of-dict(dict) ∪ rs)      MP41,3.11
                ! 3.13 cons-l-sc(c(b),d)                               MP43
         return (de)
                ! 3.14 post (this case)                                DP4,3.3,3.6,3.9
        )

     mk-type(sc-tp,<bd>) ->
        (
                ! 4.1 is-expr(bd)                                      impl.restr.
                ! 4.2 ex-tp(bd) = intg                                 EC3
         let b = cx-arr-b(t);
         let de = mk-prop-arr-de(b);
                ! 4.3 b∉regs-of-dict(dict)                            pre,4
         let n = nwrno(rs);
         m-eval-a-expr-to(bd,dict,n,rs);
                !! 4.4 let ebd-l: <eval-expr(bd,env)>                 ~
                !! 4.5 in-step-a()                                    4.2,2,4.4,MP151
                !! 4.6 ebd-l[1]=retr-val(c(n))                        4.2,2,4.4,MP151
                ! 4.7 pres(rs ∪ cx-disp-regs(t) ∪ {prno()})           4.2,2,MP151
         allign(prno(),sc-tp);
         r(b) := c(p) - (if sc-tp = intg then 4 else 1);
                ! 4.8 q-ai:=c q-ai ∪ [c(b)→<sc-tp,c(n)>]
                ! 4.9 let l' = make-array-loc(<sc-tp,c(n)>,c(b))       ~
                ! 4.10 is-array-loc(l')                                4.9,DS16,P16
                ! 4.11 l∈R l' ⊃ is-tp-sc-loc(sc-tp,l)                  4.9,P3
                ! 4.12 D l' = rect(ebd-l)                              4.6,4.9
                ! 4.13 R l' ∩ D c g-tp = {}                            1,4.9
                !! 4.14 R l' ∩ D c stg = {}                            4.13,4.5,P9
                !! 4.15 l' satisfies properties of source              DP4,4.10,4.11,4.12,4.14
                !! 4.16 q-den(de,l')                                   4.8,4.15,4.9,P15
         for i∈{1:c(n)} do
           (
                ! 4.17 g-tp := c g-tp ∪ [(i)(0,b)→sc-tp]
                !! 4.18 stg := c stg + [l'(<i>)→?]                     ~
            set((i)(0,b),m-?,sc-tp)
                !! 4.19 in-step-a()                                   4.5,4.18,P9
           );
                !! 4.20 in-step-a()                                   4.5,4.19
         r(p) := c(p) + (if sc-tp = intg then c(n)*4 else c(n));
                ! 4.21 cons-p-reg(c(p),{(i)(0,b)|1≤i≤c(n)})           P45
                ! 4.22 post1-m-eval-type(regs-of-dict(dict) ∪ rs)     4.21,MP41,4.3,4.17
         return(de)
                !! post (this case)                                   DP4,4.4,4.15,4.18
        )

type: Type Id Dict Reg-no-set => De
```

m-eval-proc-dcl

spec:

```
let em-drno,em-disp-regs = cx-em-α(pr)
                ! cons-kn-prs(dict,c env1,c g-pi,cs(em-disp-regs))
                ! c g-pi(L) = <em-disp-regs,<c g-tp,c env1,c g-ai>>
                !! q-env(dict,env)
L:
m-eval-proc-dcl(pr,dict,em-disp-regs)
                !! q-proc-den(<c(em-drno),L>,eval-proc-dcl(pr,env),c g-pi)
```

note:
The language is changed (an implementation restriction) in that values of control variables of do loops are not "known" within nested procedures. (Strictly, should revise DF5).

A case distinction will be made on call depending on whether the display must be reloaded or not - a flag is given in the setting of sc.

map:

```
5  m-eval-proc-dcl(m,dict,em-disp-regs) =

      let <parm-1,st> = m;
      let em-drno = cx-em-drno(m),
          drno = cx-drno(m),
          disp-regs = cx-disp-regs(m);
      let em-rm = cs(em-disp-regs);
                ! 1 cons-kn-prs(dict,c env1,c g-pi,em-rm)          pre
                !! 2 q-env(dict,env)                               pre
                ! 3 let d-gtp = c g-tp,
                       d-env = c env1,
                       d-gai = c g-ai,
                       d-gpi = c g-pi,
                       em-dsap = c(em-drno);
                ! 4 <em-disp-regs,<d-gtp,d-env,d-gai>>∈R d-gpi     pre,3
      let m-f =
          (     ! 4.1 note, new m-σ being considered - m-σ0
                ! 5 pre-q-proc-den(d-gpi,em-dsap,em-disp-regs)     P18pre
                !! 5.1 q-ca2(ca)                                   P18pre
                ! 6 cons-p-reg(c(p),D c g-tp)                      5,P19
                ! 7 d-gpi ≤ c g-pi                                 5,P19
                ! 8 cons-gpi(R c g-pi,<c g-tp,c env1,c g-ai>)      5,P19
                ! 9 d-gtp ≤ c g-tp ∧ d-env ≤ c env1 ∧ d-gai ≤ c g-ai        P49,4,7,8
                ! 10 c-sc(c(p))=0 ⇒ cons-disp-copy(c-disp-copy(em-dsap),
                                        cs(em-disp-regs))          5,P19
                ! 11 c-sc(c(p))≠0 ⇒ c-sc(c(p)) = em-dsap           5,P19
          if c-sc(c(p)) ≠ 0
            then rest-disp(em-disp-regs,c-sc(c(p)))
                ! 12 cons-disp-copy(c-disp-copy(em-dsap),
                                        cs(em-disp-regs))          10,11,MS62,P48
                ! 13 cs(em-disp-regs) = em-rm                      12,9
                ! 14 cons-kn-prs(dict,c env1,c g-pi,
                                        cs(em-disp-regs))          1,7,1,L2,13,P50

          r(drno) := c(p);
          let δ be s.t. δ≥ largest cx-sc-off(plus length) for any nested block;
          r(p) := c(p) + δ;
                ! 15 cons-p-reg(c(p),D c g-tp)                     6,P45
                ! 16 l-scs(D c g-tp,c(drno),c(p)) = {}             6,P44
                ! 17 cons-l-scs(c(drno),l-scs(,,),{})              16,P46
                ! 18 cons-sc-offs(c(drno),c(p),{})                 P47
                ! 19 cons-vi(c(drno),c(p),c(p),{},D c g-tp)        P42,15,17,18
                !! 21 in-step-Σ()                                  P18pre
                !! 22 q-arg-l(c-pms(c(drno)),dl)                   P18pre
          r-stack-p(c(drno)) := <>;
                !! 23 q-ca(ca,drno)                                P7,P11,5.1
                !! 24 q-env(dict,env)                              2,13,7,9,L1,L10
          dcl n-dict := dict \ cvs
                !! 25 dcl n-env := env \ cvs                       impl.restr. ~
```

```
for i ∈ {1:l parm-l} do                                                    AL2hyp
(       !! 26 q-env(dict,env)                                              AL2hyp
        !! 27 q-env(c n-dict,c n-env)                                      ~
let <id,t> = parm-l[i];                                                    22,P22,28
        !! 28 n-env:=c n-env + [id→dl[i]]
        !! 28.1 q-arg(c-pm(c(drno),i),c n-env(id))
is-scalar-type(t) ->                                                       26,L10
  (n-dict := c n-dict + [id→mk-parm-sc-de(i,drno)]                         27,28.1,P23,P15
        !! 29.1 q-env(dict,env)
        !! 29.2 q-env(c n-dict,c n-env)

  )
is-array-type(t) ->
  (n-dict := c n-dict + [id→mk-parm-arr-de(i,drno,cx-arr-b(id))];          26,L10
   r(cx-arr-b(id)) := c-pm(c(drno),i)                                      27,28.1,P23,P15
        !! 30.1 q-env(dict,env)
        !! 30.2 q-env(c n-dict,c n-env)

  )
is-proc(t) ->                                                              26,L10
  (n-dict := c n-dict + [id→mk-parm-proc-de(i,drno)]                       27,28.1,P23,P13
        !! 31.1 q-env(dict,env)
        !! 31.2 q-env(c n-dict,c n-env)

  )                                                                        AL2,24,(29.2,30.2,31.2)
);                                                                         32,21,23
        !! 32 q-env(c n-dict,c n-env)
        !! 33 in-step(c n-dict,c n-env,ca,drno)                           MS61,P48
save-disp(disp-regs,c(drno));                                              14,less
        !  34 cons-disp-ccpy(c(drno),cs(disp-regs))                        19,33,8,34,35,P41
        !  35 cons-kn-prs(c n-dict,...)                                    ~
        !  36 cons-m-Σ1(c n-dict,drno,disp-regs,{})
        !! 37 int-stmt(st,c n-env,ca)
((trap exit (abn) with                                                     AL3 hyp,42
        !! 38 in-step-a()                                                 AL3 hyp,43
        !  39 pres(disp-regs ∪ {prno()})
                                                                           38
(r(p) := c(p) - δ;                                                         39,4.1,P20,12
        !! 40 in-step-Σ()
        !  41 post-q-proc-den(c(drno),em-disp-regs)(m-σ0,)
  exit (abn)
);                                                                         36,33,37,MP2
m-int-stmt(st,c n-dict,{});                                                36,33,MP2
        !! 42 in-step-a()
        !  43 pres(disp-regs ∪ {prno()})
                                                                           AL3hyp,42
);      !! 44 in-step-a()                                                  AL3hyp,43
        !  45 pres(disp-resp ∪ {prno()})
                                                                           44
r(p) := c(p) - δ;                                                          45,4.1,P20,12
        !! 46 in-step-Σ()
        !  47 post-q-proc-den(c(drno),em-disp-regs)(m-σ0,)
                                                                           AL3,40,46
);      !! 48 in-step-a()                                                  AL3,41,47
        !  49 post-q-proc-den(c(drno),em-disp-regs)(m-σ0,)
ret (c-ra(c(p)))
    )
        !  consider declaring m-σ
        !! 50 q-proc-den(<c(em-drno),L>,                                   (DF5,25,28,37),
                          eval-proc-dcl(*,env),c q-pi)                     (5,20-22,48,49)


return m-f

type: Proc Dict Reg-no-set ->
```

m-int-nmd-stmt-list

spec:

```
   let drno,disp-regs,sc-offs = cx-α(ns-1)
                  ! cons-m-Σ1(dict,drno,disp-regs,sc-offs)
                  !! in-step(dict,env,ca,drno)
   m-int-nmd-stmt-list1(i,ns-1,dict,rs)
                  !! int-nmd-stmt-list(i,ns-1,env,ca,<c(cx-drno(ns-1)),cx-dp(ns-1)>)
                  !! in-step-a()
                  ! pres(rs ∪ disp-regs ∪ {prno()})
```

map (first stage):

```
70 m-int-nmd-stmt-list1(sno,ns-1,dict,rs) =

      trap exit (abn) with
         (let <<t-dsap,t-dp>,t-id> = abn;
          if t-dsap = c(cx-drno(ns-1)) ∧ t-dp = cx-dp(ns-1)
          then
            (let tno = (∪n)(s-nm°ns-1[n] = t-id);
             m-int-nmd-stmt-list1(tno,ns-1,dict,rs))
          else
            exit (abn)
         );
      for j = sno to 1 ns-1 do
         m-int-stmt(s-stmt°ns-1[j],dict,rs)
                  !! correctness obvious (use L5,L12)                    IF70,MF2
```

type: Intg Nmd-stmt* Dict Reg-no-set =>

note: that the induction required at this point is well founded follows from the fact that there are a finite number of labelled statements in ns-1 (cf. Ref 11).

m-int-if

spec:

```
let drno,disp-regs,sc-offs = cx-α(if)
                ! cons-m-Σ1(dict,drno,disp-regs,sc-offs)
                !! in-step(dict,env,ca,drno)
m-int-if (if,dict,rs)
                !! int-stmt(if,env,ca)
                !! in-step-a()
                ! pres(rs ∪ disp-regs ∪ {prno()})
```

notes: The special case of a relaticnal expression is recognised to avoid the operations and temporary which would result from a direct call to m-eval-b-expr.

The test for a false value in stcrage can only be made (by a machine instruction) if there is no index value: thus in the case of a reference to an element of an array of bool a work register is used.

The following proof omits many (obvious) steps in the area of expressions.

map:

```
8  m-int-if(<be,st1,st2>,dict,rs) =

   let drno,disp-regs,sc-offs = cx-α(<be,st1,st2>);
                ! 1 cons-m-Σ1(dict,drno,disp-regs,sc-offs)          pre
                !! 2 in-step(dict,env,ca,drno)                     pre
                ! 3 let m-σ1 be current
                ! 4 ex-tp(be) = bool                               DC8
                !! 5 let v:eval-expr(be,env)                       ~
   is-inf-expr(be) ∧ is-rel-op°s-op(be) ->
      (let <e1,op,e2> = be;
                ! 6.1 ex-tp(e1) = ex-tp(e2) = intg                 DC16
       let n1 = nwrno(rs);
       let n2 = nwrno(rs ∪ {n1});
       m-eval-a-expr-to(e1,dict,n1,rs);
       m-eval-a-expr-to(e2,dict,n2,rs ∪ {n1});
                !! 6.2 to get 10.2 etc., use                       6.1,2,DF15,MF151
                !! 6.3 v=retr-val(m-apply-rel-op(c(n1),op,c(n2)))
       IF ¬m-apply-rel-op(c(n1),op,c(n2)) THEN branch FALSE
      )
   is-rhs-ref(be) ∧ s-sscs°s-var-ref(be) ≠ nil ->
      (         ! 7.1 is-scalar°s-var-ref(be)
       let <d,x,b>:m-eval-b-expr(be,dict,rs);                      2,MF18
                !! 7.2 to get 10.2 etc. use
                ! 7.3 x=nwrno(rs)
                !! 7.4 v=retr-val(x-val(d(x,b),bool))
       IF x-val(d(x,b),bool) = m-false THEN branch FALSE
      )
   T ->
      (let <d,x,b>:m-eval-b-expr(be,dict,rs);
                ! 8.1 x = 0                                        2,MF18
                !! 8.2 to get 10.2 etc. use
                !! 8.3 v=retr-val(x-val(d(0,b),bool))
       IF x-val(d(0,b),bool) = m-false THEN branch FALSE
      )

   TRUE:                                                           6.3,7.4,8.3
                !!10.1 v=true
                !!10.2 in-step-a()
                ! 10.3 pres(rs ∪ disp-regs ∪ {prno()})(m-σ1,)     2,10.2,10.3,L12
                !!10.4 in-step(dict,env,ca,drno)                  1,10.3,L5
                ! 10.5 cons-m-Σ1(dict,drno,disp-regs,sc-offs)
   m-int-stmt(st1,dict,rs);                                        ~
                !!10.6 int-stmt(st1,env,ca)                       10.4,10.5,10.6,MF2
                !!10.7 in-step-a()                                10.3,10.4,10.5,MF2
                ! 10.8 pres(rs ∪ disp-regs ∪ {prno()})(m-σ1,)     10.7,10.8
                ! 10.9 using (to get 12)
   branch END;

   FALSE:
                !!11.1 v=false
   m-int-stmt(st2,dict,rs);                                        ~
                !!11.6 int-stmt(st2,env,ca)
   END:;                                                           DF2,5,10.1,10.6,11.1,11.6
                ! 12 post
```

type: If Dict Reg-no-set =>

m-int-for

spec:

```
let drno,disp-regs,sc-offs = cx-α(for)
                ! cons-m-Σ1(dict,drno,disp-regs,sc-offs)
                !! in-step(dict,env,ca,drno)
m-int-for(for,dict,rs)
                !! int-stmt(for,env,ca)
                !! in-step-a()
                ! pres(rs ∪ disp-regs ∪ {prno()})
```

In fact, the above proven using, int-for1 instead of int-stmt where:

```
is-wf-for(for) ⇒ int-for1(for,env,ca) = int-stmt(for,env,ca)
```

firstly, this result is established:

```
DF90 int-for1(<cv,e-i,e-s,e-l,b>,env,ca) =

     let i:eval-expr(e-i,env),
         s:eval-expr(e-s,env),
         l:eval-expr(e-l,env);
     let p(y) = (s>0 -> y≤l,
                 s<0 -> y≥l,
                 s=0 -> T);
     dcl dx := i;
     let f1() = (int-stmt(b,env + [cv→c dx],ca);
                 dx := c dx + s;
                 if p(c dx) then f1());
                 !! 1 let f(x) = if p(x)
                                 then (int-stmt(b,env + [cv→x],ca);
                                       f(x+s))                          ~
                 !! 2 p(c dx) ⇒ f1() = f(c dx)                          1,ind
     if p(c dx)
        then
           (     ! 3.1 p(c dx)
            f1()
                 !! 3.2 f(i)                                            ~
                 !! 3.3 =                                               3.1,2
           )
        else
           (     ! 4.1 ¬p(c dx)
            I
                 !! 4.2 f(i) = I                                        1,4.1
                 !! 4.3 =                                               4.2
           )
                 ! post                                                 DF2,1,3.2,4.2
```

type: For Env Ca =>

For convenience, m-f1 is handled separately:

```
spec:
                    !  {r-s,r-l,r-x} ∩ rs = {}
                    !! s = retr-val(c(r-s))
                    !! l = retr-val(c(r-l))
                    !  cons-m-Σ1(dict,drnc,disp-regs,sc-offs)
                    !! in-step(dict,env + [cv→c dx],ca,drno)
   m-f1(b,dict,rs)
                    !! f1()
                    !! in-step-a()
                    !  pres(rs ∪ disp-regs ∪ {prno()})
```

note: splits of tests to fit the loop/test instructions available on the machine.

map:

```
90 m-f1(b,dict,rs) =
                    !  1 {r-s,r-l,r-x} ∩ rs = {}                          pre
                    !! 2 s=retr-val(c(r-s))                               pre
                    !! 3 l=retr-val(c(r-l))                               pre
                    !  5 cons-m-Σ1(dict,drno,disp-regs,sc-offs)           pre
                    !! 6 in-step(dict,env + [cv→c dx],ca,drno) pre
                    !  7 let m-σ1 be current
     INTST1:
                    ! 11 pres(rs ∪ {r-s,r-l} ∪ disp-regs ∪ {prno()}) (m-σ1,)   7,16
                    !!12 in-step(dict,env + [cv→c dx],ca,drno) 6,15,16,L12
                    ! 13 cons-m-Σ1(dict,drnc,disp-regs,sc-offs)          5,16,L5
     m-int-stmt(b,dict,rs ∪ {r-s,r-l,r-x});
                    !!14 int-stmt(b,env + [cv→c dx],ca)                  ~
                    !!15 in-step-a()                                     12,13,14,MF2
                    ! 16 pres(rs ∪ {r-s,r-l,r-x} ∪ disp-regs ∪ {prno()}) 12,13,MF2
     IF c(r-s) < m-0 THEN branch NEGS1;
                    !!17 branch iff s<0                                  2,16
     IF c(r-s) = m-0 THEN
        (           !!18 s = 0                                           2,16
                    !!19 dx := c dx + 0                                  ~
         branch INTST1
                    !!20 branch if p(c dx)                               18
                    ! 21 using (to get 11)                              16
                    !!22 using (to get 12,13)                           15,16,L12,L5
        )
                    !!23 s > 0                                           17,20
     r(r-x) := c(r-x) + c(r-s);
                    !!24 dx := c dx + s                                  ~
                    !!25 in-step1(dict,env + [cv→c dx])                  12,15,16,L12,2,P13
     IF c(r-x) ≤ c(r-l) THEN branch INTST1;
                    !!26 branch if p(c dx)                               3
                              ELSE branch END1;
                    !!27 branch if ¬p(c dx)
     NEGS1:
                    !!28 s < 0                                           17
     r(r-x) := c(r-x) + c(r-s);
                    !!29 dx := c dx+s                                    ~
                    !!30 in-step1(dict,env + [cv→c dx])
     IF c(r-x) ≥ c(r-l) THEN branch INTST1;
                    !!31 branch if p(c dx)
     END1:
                    !!32 ¬ p(c dx)                                       31,26
                    !!33 in-step-a()                                     15
                    ! 34 pres(rs ∪ disp-regs ∪ {prno()})                 16
                    !  post                                              DF90,14,19,24,29,20,26,31,32

  type: Stmt Dict Reg-no-set =>
```

```
9  m-int-for(<cv,e-i,e-s,e-l,b>,dict,rs) =

                  ! 1 cons-m-Σ1(dict,drno,disp-regs,sc-offs)          pre
                  !! 2 in-step(dict,env,ca,drno)                      pre
                  ! 3 ex-tp(e-i) = ex-tp(e-s) = ex-tp(e-l) = intg     EC9
      let r-s = nwrno(rs),
          r-l = nwrno(rs ∪ {r-s}),
          r-x = nwrno(rs ∪ {r-s,r-l});
      m-eval-a-expr-to(e-s,dict,r-s,rs);
                  !! 4 let s:eval-expr(e-s,env)                       ~
      m-eval-a-expr-to(e-l,dict,r-l,rs ∪ {r-s});
                  !! 5 let l:eval-expr(e-l,env)                       ~
      m-eval-a-expr-to(e-i,dict,r-x,rs ∪ {r-s,r-l});
                  !! 6 let i:eval-expr(e-i,env)                       ~
                  !! 7 in-step-a()                                    2,3,MF151x3
                  ! 8 pres(rs ∪ disp-regs ∪ {prno()})                 2,3,MF151x3
                  !! 9 s=retr-val(c(r-s))
                  !!10 l=retr-val(c(r-l))
                  !!11 i=retr-val(c(r-x))
                  !!12 dcl dx:=i                                      ~
                  !!13 c dx=retr-val(c(r-x))                          11,12
      IF c(r-s) < 0 THEN branch NEGS;
                  !!14 branch iff s<0                                 9
      IF c(r-s) = 0 THEN branch INTF1;
                  !!15 branch iff s=0                                 9

                  !!16 s>0                                            14,15
      IF c(r-x) > c(r-l) THEN branch END;
                  !!17 branch if ¬ p(c dx)                            16,13,10
                         ELSE branch INTF1;
                  !!18 branch if p(c dx)                              16,13,10

      NEGS:
                  !!19 s<0                                            14
      IF c(r-x) < c(r-l) THEN branch END;
                  !!20 branch iff ¬p(c dx)                            19,13,10

      INTF1:
                  !!21 p(c dx)                                        15,18,20
      let n-dict = dict + [cv-mk-cv-de(r-x)];
                  !!22 in-step(dict,env + [cv-c dx],ca,drno)          2,7,8,L12,13,P13
                  ! 23 cons-m-Σ1(dict,drno,disp-regs,sc-offs)        2,7,8,L5
      m-f1(b,n-dict,rs);
                  !!24 f1()                                           ~
                  !!25 in-step-a()                                    9,10,23,22,24,MF90
                  ! 26 pres(rs ∪ disp-regs ∪ {prno()})               9,10,23,22,MF90
      END:I;
                  !!27 ¬p(c dx)                                       17,20
                  !!28 in-step-a()                                    25,7,8,L12
                  ! 29 pres(rs ∪ disp-regs ∪ {prno})                 26,8
                  ! post                                             DF90,4,5,6,12,21,24

   type: For Dict Reg-no-set =>
```

Section: MAPPING

m-int-call

spec:

```
   let drno,disp-regs,sc-offs = cx-α(t)
                ! cons-m-Σ1(dict,drno,disp-regs,sc-offs)
                !! in-step(dict,env,ca,drno)
   m-int-call(t,dict,rs)
                !! int-stmt(t,env,ca)
                !! in-step-a()
                ! pres(rs ∪ disp-regs ∪ {prno()})
```

notes: Case distinction (anticipated in m-eval-proc-dcl) on direct or parameter call aims to avoid reloading display where that required is a sub part of that of the caller. Furthermore, some economy on what must be restored after a call is possible.

Note that, in the final code when dsa's are merged into store, the parameters etc. are loaded "beyond" the current stack pointer.

```
10 m-int-call(ϖ,dict,rs) =

      let <pid,al> = ϖ;
      let drno = cx-drno(ϖ),
          drns = cx-disp-regs(ϖ);
                ! 1 cons-m-Σ1(dict,drno,drns,cx-em-sc-offs(ϖ))      pre
                !! 2 in-step(dict,env,ca,drno)                      pre
      env1 := c env1 ∪ [c(p) → null-dsa];
                ! 2.1 let m-σ1 be current
                ! 3 q-ca2(ca)                                       2,P7,P11
                !! 4 dcl dl:=<nil|1≤i≤l al>                         ~
      for i∈{1:l al} do
         (
                !! 5 in-step1(dict,env)                            AL2hyp
       is-var-ref(al[i]) ->
         (let <d,x,b>:m-eval-var-ref(al[i],dict,rs);
                !! 6 dl[i]:=eval-var-ref(al[i],env)                 ~
                !! 7 in-step-a()                                   5,6,MP10
                ! 8 pres(rs ∪ drns ∪ {prno()})                     5,MP10
          r-pm(c(p),i) := d(x,b)
                !! 9 q-arg(c-pm(c(p),i),c dl[i])                   5,6,MP10,P23
         )
       is-id(al[i]) ->
         (     !!10 dl[i]:=env(al[i])                             ~
          cases dict(al[i]):
          mk-prop-proc-de(L, ,em-drno) ->
            (  !!11 q-proc-den(<c(em-drno),L>,env(al[i]),c q-pi)   5,P13
             r-pm(c(p),i) := <c(em-drno),L>
                !!12 q-arg(c-pm(c(p),i),c dl[i])                   10,11,P23
            )
          mk-parm-proc-de(d,b) ->
            (  !!13 q-proc-den(c-pm(c(b),d),env(al[i]),c q-pi)     5,P13
             r-pm(c(p),i) := c-pm(c(b),d)
                !!14 q-arg(c-pm(c(p),i),c dl[i])                   10,13,P23
            )
                !!15 in-step1(dict,env)                            L12,(7,8),(I)
         )
      );
                !!16 in-step1(dict,env)                            AL2,2,15
                ! 17 pres(rs ∪ drns ∪ {prno()}) (m-σ1,)            8
                !!18 q-arg-1(c-pms(c(p)),c dl)                     9,12,14
      r-dc(c(p)) := c(drno);
      store-wrs(rs);
                ! 19 cons-m-Σ1(dict,drno,drns,cx-em-sc-offs(ϖ))    1,17,L5'
                !!20 let <f>=env(pid)                              ~
```

```
      cases dict(pid):
      mk-prop-proc-de(L,em-rns,em-drno) ->
         (        !!21 q-proc-den(<c(em-drno),L>,<f>,c q-pi)          16,20,P13
          r-sc(c(p)) := 0
                  ! 23 cons-kn-prs(dict,c env1,c q-pi,cs(drns))       19,P41
                  ! 24 cons-disp-copy(c-disp-copy(c(em-drno)),
                                      cs(em-rns))                     23,P50
                  ! 25 pre-q-proc-den(c q-pi,c(em-drno),em-rns)       19,24,P19
                  !!26 f(c dl,ca)                                     "
          ((trap exit (abn) with
                  ! 27 post-q-proc-den(c(em-drno),em-rns)             AL3hyp,32
                  !!28 in-step-a()                                    AL3hyp,31
            (r(drno) := c-dc(c(p));
             rest-disp(drns \ em-rns,c(drno));
             env1 := c env1 \ {c(p)};
             rest-wrs(rs);
                  ! 29 pres(rs u disp-regs {prno()})(m-σ2,)           17,27,P20
                  !!30 in-step-a()                                    28
             exit (abn)
            );
           call L(r-ra(c(p)))
                  !!31 in-step-a()                                    25,3,16,18,26,21,P18
                  ! 32 post-q-proc-den(c(em-drno),em-rns)             25,3,16,21,P18
           );
                  !!33 in-step-a()                                    AL3hyp,31
                  ! 34 post-q-proc-den(c(em-drno),em-rns)             AL3hyp,32
            r(drno) := c-dc(c(p));
            rest-disp(drns \ em-rns,c(drno));
            env1 := c env1 \ {c(p)};
            rest-wrs(rs)
                  ! 35 pres(rs u disp-regs u {prno()})(m-σ2,)         17,L2,34,P20
                  !!36 in-step-a()                                    33
         )
                  ! 37 pres(rs u disp-regs u {prno()})(m-σ2,)         AL3,29,35
                  !!38 in-step-a()                                    AL3,30,36
                  ! post (this case)                                  DF2,20,4,6,10,26
      )
      mk-parm-proc-de(d,b) ->
         (let <em-dsa,L> = c-pm(c(b),d);
                  !!41 q-proc-den(<em-dsa,L>,<f>,c q-pi)              16,20,P13
          r-sc(c(p)) := em-dsa;
                  ! 42 pre-q-proc-den(c q-pi,c(em-drno),em-rns)       19,P19
                  !!43 f(dl,ca)                                       "
          ((trap exit (abn) with
                  ! 44 post-q-proc-den(c(em-drno))                    AL3hyp,49
                  !!45 in-step-a()                                    AL3hyp,48
            (r(drno) := c-dc(c(p));
             rest-disp(drno,c(drno));
             env1 := c env1 \ {c(p)};
             rest-wrs(rs);
                  ! 46 pres(rs u disp-regs u {prno()})(m-σ2,)         17,44,P20
                  !!47 in-step-a()                                    45
             exit (abn)
            );
            call L(r-ra(c(p)))
                  !!48 in-step-a()                                    42,3,16,18,43,41,P18
                  ! 49 post-q-proc-den(c(em-drno),)                   42,3,16,18,41,P18
           );
                  !!50 in-step-a()                                    AL3hyp,48
                  ! 51 post-q-proc-den(c(em-drno),)                   AL3hyp,49
           r(drno) := c-dc(c(p));
           rest-disp(drns,c(drno));
           env1 := c env1 \ {c(p)};
           rest-wrs(rs)
                  ! 52 pres(rs u disp-regs u {prno()})(m-σ2,)         17,51,P20
                  !!53 in-step-a()                                    50
         )
                  ! 54 pres(rs u disp-regs u {prno()})(m-σ2,)         AL3,46,52
                  !!55 in-step-a()                                    AL3,47,53
                  ! post (this case)                                  DF2,20,4,6,10,43
      )

   type: Call Dict Reg-no-set =>


101 store-wrs(rns) =
      /* not further defined */

   type: Reg-no-set =>

Section: MAPPING
```

```
102 rest-wrs(rns) =
      /* not further defined */

  type: Reg-no-set =>
```

m-int-goto

```
spec:
                !! in-step1(dict,env)
  m-int-goto1(t,dict,rs)
                !! int-stmt(t,env,ca)
                !! in-step-a()
                !  pres(rs ∪ cx-disp-regs(t) ∪ {prno()})

map (first stage):

11 m-int-goto1(<id>,dict,rs) =
                !! 1 in-step1(dict,env)                                  pre
                !! 2 q-den(dict(id),env(id))                             1
     let <lab,t-drno,t-dp> = dict(id);
     let t-lab-den = mk-lab-den(mk-aid(c(t-drno),t-dp),retr-nm(lab));
                !! 3 env(id) = t-lab-den                                 2,P14
     exit(t-lab-den)
                !! 4 exit(env(id))                                       ~
                !! 5 in-step-a()
                !  6 pres(rs ∪ cx-disp-regs(<id>) ∪ {prno()})
                !  post                                                  PF2,4

  type: Goto Dict Reg-no-set =>
```

m-int-assn

spec:

```
                !! in-step1(dict,env)
   m-int-assn(t,dict,rs)
                !! in-stmt(t,env,ca)
                !! in-step-a()
                !  pres(rs ∪ cx-disp-regs(t) ∪ {prno()})
```

note: The case distinction within the bool case is designed to make use of the eventual target as a temporary whenever this is valid. Notice that no-share-poss is a 'fail-safe' static test whilst no-share applies to actual state.

Independance of m-Sc-locs required, cf. note after MS21.

map:

```
12 m-int-assn(w,dict,rs) =

      let <vr,e> = w;
                !  1 let r1 = rs ∪ cx-disp-regs(w) ∪ {prno()}
                !  2 let m-σ0 be current
                !! 3 in-step1(dict,env)                                  pre
                !! 3.1 is-scalar(vr)                                     DC12
      let <d,x,b>:m-eval-var-ref(vr,dict,rs);
                !! 4 let l:eval-var-ref(vr,env)                          ~
                !! 5 in-step1(dict,env)                                  3,4,MF18,L12(weak form)
                !! 6 l=make-tp-sc-loc(vr-tp(vr),d(x,b))                  3,4,MF18,3.1
                !  7 pres(r1)(m-σ0,)                                     3,MF18,1,2
                !  8 x=0 ∨ x=nwrno(rs)                                   3,MF18
      let rs' = rs ∪ (if x=0 then {} else {nwrno(rs)}) ∪
                     (if b∈regs-of-dict(dict) then {} else {nwrno(rs)});
                !  9 let r2 = r1 ∪ rs'
                ! 10 let m-σ1 be current
                !!11 let v:eval-expr(e,env)                              ~
      (cases vr-tp(vr):
         intg ->
            (     ! 12.1 ex-tp(e) = intg                                 12,DC12
            let n = nwrno(rs');
            m-eval-a-expr-to(e,dict,n,rs');
                  !!12.2 in-step-a()                                     5,11,12.1,MF151
                  ! 12.3 pres(r2)(m-σ1,)                                 5,12.1,9,10,MF151
                  !!12.4 v=retr-val(c(n))                                5,12.1,11,MF151
                  !!12.5 l=make-tp-sc-loc(intg,d(x,b))                   6,8,12.3
            set(d(x,b),c(n),intg)
                  !!12.6 using (to get 16)                               12.2,12.4,12.5,15,P9
                  ! 12.7 using (to get 17)                               12.3,7
            )
       bool ->
                  ! 13.1 ex-tp(e) = bool                                 13,DC12
         if no-share-poss(vr,e)
         then
            (     ! 13.2 no-share(<d,x,b>,e)
            m-eval-b-expr-to(e,dict,<d,x,b>,rs')
                  !!13.3 using (to get 16)                               5,6,13.1,13.2,MF153,11,15
                  ! 13.4 using (to get 17)                               5,6,13.1,13.2,MF153,7
            )
         else
            (
            let <d',x',b'>:m-eval-b-expr(e,dict,rs');
                  !!14.1 in-step-a()                                     5,13.1,MF152,11
                  ! 14.2 pres(r2)(m-σ1,)                                 5,13.1,MF152,10
                  !!14.3 v=retr-val(x-val(d'(x',b'),bool)                5,13.1,MF152,11
                  !!14.4 l=make-tp-sc-loc(bool,d(x,b))                   6,14.2
            set(d(x,b),x-val(d'(x',b'),bcol),bool)
                  !!14.5 using (to get 16)                               14.1,14.3,14.4,15,P9
                  ! 14.6 using (to get 17)                               14.2,7
            )
      )
                  !!15 assign(l,v)                                       ~
                  !!16 in-step-a()                                       12.6,13.3,14.5
                  ! 17 pres(rs ∪ cx-disp-regs(w) ∪ {prno()})(m-σ0,)      1,12.7,13.4,14.6
                  !    post                                             DP2,4,11,15
```

type: Assn Dict Reg-no-set =>

m-eval-a-expr-to

spec:

```
                !  n≠rs
                !  ex-tp(e) = intg
                !! in-step1(dict,env)
  m-eval-a-expr-to(e,dict,n,rs)
                !! let v:eval-expr(e,env)
                !! in-step-a()
                !! v=retr-val(c(n))
                !  pres(rs ∪ cx-disp-regs(e) ∪ {prno()})
```

notes: Evaluation always performed intc register targets. Case distinction cn seccnd operand of an infix expression avoids unnecessary loads (or the use of c applied to regs or stg!).

map:

```
151 m-eval-a-expr-to(e,dict,n,rs) =

    let disp-regsp = cx-disp-regs(e) ∪ {prno()};
                !  1 n≠rs                                          pre
                !  2 ex-tp(e) = intg                               pre
                !! 3 in-step1(dict,env)                            pre
                !  4 let m-σ1 be current                           pre
    cases e:
    mk-inf-expr(e1,op,e2) ->
      (         !  6.1 ex-tp(e1) = ex-tp(e2) = intg                2,DC16,DC151
                !  6.2 is-intg-op(op)                              2,DC151
       m-eval-a-expr-to(e1,dict,n,rs);
                !! 6.3 let v1:eval-expr(e1,env)                    ~
                !! 6.4 in-step1(dict,env)                          6.1,3,6.3,MF151,L12
                !  6.5 pres(rs ∪ disp-regsp)                       6.1,3,MF151
                !! 6.6 v1 = retr-val(c(n))                         6.1,3,6.3,MF151
                !! 6.7 let v2:eval-expr(e2,env)                    ~
       cases e2:
       mk-inf-expr() ->
        (let nn = nwrno(rs ∪ {n});
         m-eval-a-expr-to(e2,dict,nn,rs ∪ {n});
                !! 6.8.1 in-step-a()                              6.1,6.4,6.7,MF151
                !! 6.8.2 v2 = retr-val(c(nn))                     6.1,6.4,6.7,MF151
                !  6.8.3 pres(rs ∪ {n} ∪ disp-regsp)               6.1,6.4,MF151
         r(n) := m-apply-intg-op(c(n),op,c(nn))
                !! 6.8.4 using (to get 6.13)                      6.12,6.8.1
                !! 6.8.5 using (to get 6.14)                      6.12,6.6,6.8.2,6.8.1
                !  6.8.6 using (to get 6.15)                       6.5, 6.8.3
        )
       mk-rhs-ref(vr) ->
        (       !  6.9.1 is-scalar(vr)                             DC17
         let <d,x,b>:m-eval-var-ref(vr,dict,rs ∪ {n});
                !! 6.9.2 in-step-a()                              6.4,MF18,DF15,6.7
                !! 6.9.3 v2 = retr-val(x-val(d(x,b),intg))         6.4,MF18,DF15,P9
                !  6.9.4 pres(rs ∪ {n} ∪ disp-regsp)               6.4,MF18
         r(n) := m-apply-intg-op(c(n),op,x-val(d(x,b),intg))
                !! 6.9.5 using (to get 6.13)                      6.12,6.9.2
                !! 6.9.6 using (to get 6.14)                      6.12,6.6,6.9.3
                !  6.9.7 using (to get 6.15)                       6.5,6.9.4
        )
       mk-cv-ref(id) ->
        (let <r> = dict(id);
                !! 6.10.1 retr-val(c(r)) = env(id)                6.4,P13
                !! 6.10.2 v2 = retr-val(c(r))                     6.7,6.10.1,DF15
         r(n) := m-apply-intg-op(c(n),op,c(r))
                !! 6.10.3 using (to get 6.13)                     6.12,6.4
                !! 6.10.4 using (to get 6.14)                     6.12,6.6,6.10.2
                !! 6.10.5 using (to get 6.15)                     6.5
        )
       T -> /*is-const(e2) */
        (let d = cx-const-off(e2);
                !! 6.11.1 v2 = retr-val(x-val(d(0,drno-0),intg))  6.7,DF15
         r(n) := m-apply-intg-op(c(n),op,x-val(d(0,drno-0),intg))
        )
                !! 6.12 let v:apply-op(v1,op,v2)
                !! 6.13 in-step-a()
                !! 6.14 v = retr-val(c(n))
                !  6.15 pres(rs ∪ disp-regsp)(m-σ1,)
                !! post (this case)                               DF15,6.3,6.7,6.12
    )
```

```
mk-rhs-ref(vr) ->
      (     !  7.1 is-scalar(vr)                                   DC17
       let <d,x,b>:m-eval-var-ref(vr,dict,rs ∪ {n});
            !! 7.2 let l:eval-var-ref(vr,env)                     ~
            !! 7.3 in-step-a()                                    3,7.2,MF18
            !! 7.4 l = make-tp-sc-loc(intg,d(x,b))                3,7.2,MF18,7.1
            !  7.5 pres(rs ∪ {n} ∪ disp-regsp)                    3,MF18
       r(n) := x-val(d(x,b),intg)
            !! 7.6 let v:contents(l)                             ~
            !! 7.7 v = retr-val(c(n))                            7.4,7.3,7.6,P9
            !! post (this case)                                  DF15,7.2,7.6
      )
mk-cv-ref(id) ->
      (let <r> = dict(id);
            !! 8.1 let v = env(id)                               ~
            !! 8.2 v = retr-val(c(r))                           8.1,3,P13
       r(n) := c(r)
            !! 8.3 v = retr-val(c(n))                           8.2
            !! post (this case)                                 DF15,8.1
      )
T -> /* is-const(e) */
      (let d = cx-const-off(e);
            !! 9.1 let v = val-of(e)                            ~
            !! 9.2 v = retr-val(x-val(d(0,drno-0),intg))
       r(n) := x-val(d(0,drno-0),intg)
            !! post (this case)                                 DF15,9.1
      )

type: Expr Dict Reg-no Reg-no-set =>
```

m-eval-b-expr

spec:

```
                ! ex-tp(e) = bool
                !! in-step1(dict,env)
  let <d,x,b>:m-eval-b-expr(e,dict,rs)
                !! let v:eval-expr(e,env)
                !! in-step-a()
                !! v = retr-val(x-val(d(x,b),bool))
                ! pres(rs ∪ cx-disp-regs(e) ∪ {prno()})
                ! x≠0 ⇒ (is-rhs-ref(e) ∧ s-sscs°s-var-ref(e)≠nil)
```

note: (implied) use of SS instructions, cf. MF151

map:

```
152 m-eval-b-expr(e,dict,rs) =

    let b = cx-drno(e);
                !! 1 ex-tp(e) = bool                                  pre
                !! 2 in-step1(dict,env)                              pre
    cases e:
    mk-inf-expr() ->
       (let d = cx-temp-off(e);
        m-eval-b-expr-to(e,dict,<d,0,b>,rs);
                !! 6.1 let v:eval-expr(e,env)                        ~
                !! 6.2 in-step-a()                                   1,2,6.1,MF153
                !! 6.3 v = retr-val(x-val(d(0,b),bool))              1,2,6.1,MF153
                ! 6.4 pres(rs ∪ cx-disp-regs(e) ∪ {prno()})         1,2,MF153
        return (<d,0,b>)
                ! post (this case)
       )                                                             6.3,6.1
    mk-rhs-ref(vr) ->
                ! 7.1 is-scalar(vr)
       (let <d,x,b>:m-eval-var-ref(vr,dict,rs);
                !! 7.2 let l:eval-var-ref(vr,env)                    ~
                !! 7.3 in-step-a()                                   2,MF18,7.2
                !! 7.4 l = make-tp-sc-loc(bool,d(x,b))               2,MF18,7.2,7.1
                ! 7.5 pres(rs ∪ cx-disp-regs(e) ∪ {prno()})         2,MF18
        return (<d,x,b>)
                !! 7.6 let v:contents(l)                             ~
                !! 7.7 v = retr-val(x-val(d(x,b),bool))              7.4,7.3,7.6
                                                      unless error
                ! 7.8 x≠0 ⇒ s-sscs(vr)≠nil
                ! post (this case)                                   DF15,7.2,7.6
       )
    mk-cv-ref() ->
       (        ! 8.1 can not occur!
       )
    T -> /* const */
       (        ! 9.1 let v = val-of(e)                              ~
        let d = cx-const-off(e);
                !! 9.2 v = retr-val(x-val(d(0,drno-0),bool))
        return (<d,0,drno-0>)
                ! post (this case)                                   DF15,9.1
       )
```

type: Expr Dict Reg-no-set => c-Opd

m-eval-b-expr-to

spec:

```
                ! ex-tp(e) = bool ∧ no-share(tg,e)
                !! in-step1(dict,env)
                !! l = make-tp-sc-loc(bool,tg)
   m-eval-b-expr-to(e,dict,tg,rs)
                !! (let v:eval-expr(e,env);assign(l,v))
                !! in-step-a()
                ! pres(rs ∪ cx-disp-regs(e) ∪ {prno()})
```

note: (implied) use of SS instructions, cf. MF151

map:

```
153 m-eval-b-expr-to(e,dict,<d,x,b>,rs) =

      let disp-regsp = cx-disp-regs(e) ∪ {prno()};
                ! 1 ex-tp(e) = bool                                   pre
                ! 2 no-share(<d,x,b>,e)                              pre
                !! 3 in-step1(dict,env)                              pre
                !! 4 l=make-tp-sc-loc(bool,<d,x,b>)                  pre
                ! 5 let m-σ1 = current state
      cases e:
      mk-inf-expr(e1,op,e2) ->
         (is-bool-op(op) ->
            (     ! 6.1 ex-tp(e1) = ex-tp(e2) = bool                1,DC16
                  ! 6.2 no-share(<d,x,b>,e1) ∧ no-share(<d,x,b>,e2) 2
             m-eval-b-expr-to(e1,dict,<d,x,b>,rs);
                  !! 6.3 let v1:eval-expr(e1,env)                   ~
                  !! 6.4 in-step-a() except l, but not referenced   6.2,6.1,6.3,3,MF153
                  ! 6.5 pres(rs ∪ disp-regsp)                       3,6.1,MF153
                  !! 6.6 in-step1(dict,env)                         3,6.4,6.5,L12
             let <d',x',b'>:m-eval-b-expr(e2,dict,rs);
                  !! 6.7 let v2:eval-expr(e2,env)                   ~
                  !! 6.8 in-step-a()                                6.1,6.6,6.7,MF152
                  !! 6.9 v2 = retr-val(x-val(d'(x',b'),bool))       6.1,6.6,6.7,MF152
                  ! 6.10 pres(rs ∪ disp-regsp)                      6.1,6.6,MF152
             set(d(x,b),m-apply-bool-op(x-val(d(x,b),bool),op,
                                        x-val(d'(x',b'),bool)),bool)
                  !! 6.11 assign(l,apply-cp(v1,op,v2))              ~
                  ! 6.12 in-step-a()                                6.8,P9,6.11
                  ! 6.13 pres(rs ∪ disp-regsp)(m-σ1,)               6.5,6.10
                  ! post (this case)                                DF15,6.3,6.7,6.11
            )
          is-rel-op(op) ->
            (     ! 7.1 ex-tp(e1) = ex-tp(e2) = intg                1,DC16
             let n1 = nwrno(rs);
             m-eval-a-expr-to(e1,dict,n1,rs);
                  !! 7.2 let v1:eval-expr(e1,env)                   ~
                  !! 7.3 in-step-a()                                7.1,3,7.2,MF151
                  !! 7.4 v1 = retr-val(c(n1))                       7.1,3,7.2,MF151
                  ! 7.5 pres(rs ∪ disp-regsp)                       7.1,3,MF151
                  !! 7.6 in-step1(dict,env)                         3,7.3,7.5,L12
             let n2 = nwrno(rs ∪ {n1});
             m-eval-a-expr-to(e2,dict,n2,rs ∪ {n1});
                  !! 7.7 let v2:eval-expr(e2,env)                   ~
                  !! 7.8 in-step-a()                                7.1,7.6,7.7,MF151
                  !! 7.9 v2 = retr-val(c(n2))                       7.1,7.6,7.7,MF151
                  ! 7.10 pres(rs ∪ {n1} ∪ disp-regsp)               7.1,7.6,MF151
             (cc := m-apply-rel-op(c(n1),op,c(n2));
              set(d(x,b),retr-truth(c cc),bool))
                  !! 7.11 assign(l,apply-op(v1,op,v2))              ~
                  !! 7.12 in-step-a()
                  ! 7.13 pres(rs ∪ disp-regsp)(m-σ1,)               7.5,7.10
                  ! post (this case)                                DF15,7.2,7.7,7.11
            )
         )
```

```
    mk-rhs-ref(vr) ->
       (        !  8.1 is-scalar(vr)
        let <d',x',b'>:m-eval-var-ref(vr,dict,rs);
                !! 8.2 let l':eval-var-ref(vr,env)                    ~
                !! 8.3 in-step-a()                                   3,8.2,MP18
                !! 8.4 l' = make-tp-sc-loc(bool,d'(x',b'))           3,8.2,8.1,MP18
                !  8.5 pres(rs ∪ disp-regsp)                         3,MP18
                !! 8.6 let v':contents(l')                           ~
                !! 8.7 v' = retr-val(x-val(d'(x',b'),bool))          8.3,8.6,P9
        set(d(x,b),x-val(d'(x',b'),bool),bool)
                !! 8.8 assign(l,v')                                  ~
                !! 8.9 in-step-a()
                !  post (this case)                                  DF15,8.2,8.6,8.8
       )
    mk-cv-ref() -> /* cannot occur */
    T -> /* is-const(e) */
       (let d':cx-const-off(e);
                !! 9.1 let v = val-of(e)                             ~
                !! 9.2 v=retr-val(x-val(d'(0,drno-0),bool)
        set(d(x,b),x-val(d'(0,drno-0),bool),bool)
                !! 9.3 assign(l,v)                                   ~
                !! 9.4 in-step-a                                     3.4,9.2
                !  9.5 pres(rs ∪ disp-regsp)
                !  post (this case)                                  DF15,9.1,9.3
       )
```

type: Expr Dict c-Opd Reg-no-set =>

```
154 m-apply-intg-op(v1,op,v2)=
      /* (restricted) analog of apply-cp */
```

type: m-Intg Intg-op m-Intg -> m-Intg

```
155 m-apply-bool-op(v1,op,v2) =
      /* (restricted) analog of apply-op */
```

type: m-Bool Bool-op m-Bool -> m-Bool

```
156 m-apply-rel-op(v1,op,v2) =
      /* (restricted) analog of apply-cp */
```

type: m-Intg Rel-op m-Intg -> m-Bool

m-eval-var-ref

spec:

```
                !! in-step1(dict,env)
  let <d,x,b>:m-eval-var-ref(vr,dict,rs)
                !! let l:eval-var-ref(vr,env)
                !! in-step-a()
                !! if is-scalar(vr)
                      then l=make-tp-sc-loc(vr-tp(vr),d(x,b))
                      else (d = x = 0 ∧
                            l = make-array-loc(c g-ai(c(b)),c(b)))
                !  pres(rs ∪ cx-disp-regs(vr) ∪ {prno()})
                !  x=0 ∨ x=nwrno(rs)
                !  b∈regs-of-dict(dict) ∨ b=nwrno(rs)
                !  x=0 ∨ b∈regs-of-dict(dict)
```

map:

```
18 m-eval-var-ref(vr,dict,rs) =

     let <id,ssc-l> = vr;
                !! 1 in-step1(dict,env)                                  pre
                !! 2 env(id) = retr-loc(dict(id))                        1,P13
     cases ssc-l:
     nil ->
        (cases dict(id):
         mk-prop-sc-de(d,b) ->
           (     !  3.1 is-scalar(vr)                                   DC182
                 !! 3.2 env(id) = make-tp-sc-loc(c g-tp(d(0,b)),d(0,b))   2,P15
                 !  3.3 c g-tp(d(0,b)) = vr-tp(vr)
                 !  3.4 b∈regs-of-dict(dict)                            MS14
            return (<d,0,b>)
                 !  3.5 state unchanged
                 !! post (this case)                                    DF18,3.1-3.5
           )
         mk-parm-sc-de(d,b) ->
           (     !  4.1 is-scalar(vr)                                   DC182
            let n = nwrno(rs);
            r(n) := c-pm(c(b),d);
                 !! 4.2 env(id) = make-tp-sc-loc(c g-tp(c(n)),c(n))     2,P15
                 !  4.3 c g-tp(c(n)) = vr-tp(vr)
            return (<0,0,n>)
                 !  4.4 only state change to c(n)
                 !! post (this case)                                    DF18,4.1-4.4
           )
         mk-prop-arr-de(b) ∧ mk-parm-arr-de(_,b) ->
           (     !  5.1 ¬is-scalar(vr)                                  DC182
                 !  5.2 b∈regs-of-dict(dict)                            MS14
                 !! 5.3 env(id) = make-array-loc(c g-ai(c(b)),c(b))     2,P15
            return (<0,0,b>)
                 !  5.4 state unchanged
                 !! post (this case)                                    DF18,5.1-5.4
           )
        )
      <ssc> ->
         (       !  6.1 is-scalar(vr)                                   DC182
                 !  6.2 is-expr(ssc)                                    impl.restr.
                 !  6.3 ex-tp(ssc) = intg                               DC18
                 !! 6.4 let a-loc = env(id)                            ~
          let b = s-base(dict(id));
                 !  6.5 b∈regs-of-dict(dict)                            MS14
                 !! 6.6 a-loc = make-array-loc(c g-ai(c(b)),c(b))       6.4,2,P15
          let n = nwrno(rs);
          m-eval-a-expr-to(ssc,dict,n,rs);
                 !! 6.7 let ev-ssc-l:<eval-expr(ssc,env)>              ~
                 !! 6.8 in-step-a()                                    6.3,1,6.7,MF151
                 !! 6.9 ev-ssc-l[1] = retr-val(c(n))                   6.3,1,6.7,MF151
                 !  6.10 pres(rs ∪ cx-disp-regs(ssc) ∪ {prno()})       6.3,1,MF151
                 !! 6.11 1≤ev-ssc-l[1]≤bd          else error
          if vr-tp(vr) = intg
            then r(n) := c(n) * 4;
                 !! 6.12 a-loc(ev-ssc-l) = make-tp-sc-loc(vr-tp(vr),0(n,b))  P16,6.6,6.9
          return (<0,n,b>)
                 !! post (this case)                                   DF18,6.8,6.10,6.1,
                                                                       6.12,6.4,6.7
         )

   type: Var-ref Dict Reg-no-set => c-Cpd
```

Section: MAPPING

5. MAP STAGE II

References to formulae of this section are of the form "MGn".

In the actual machine the dsa's are stored in the same storage as the variables themselves, the split of the first stage was intended only to make the reasoning more perspicuous.

```
1  m-Σ2 :: stg2 : Byte*
           regs : (Reg-no -> Byte*)
           cc   : Byte
           g-di : (Intg -> Dsa-inf)
           g-li : (c-Label -> Lab-inf)
```

The ghost variable "g-di" will permit retrieval of a state of m-Σ1 as follows:

```
2  Dsa-inf :: pm-inf : (loc | proc)*
              dc-inf : (Intg Reg-no*)
              ps-inf : (Intg Intg)
```

```
3  retr-Σ1 : m-Σ2 -> m-Σ1
```

Such a function is not difficult to visualise (it does not, of course, have to generate the ghost variables of m-Σ1).

The changes to the map functions revolve mainly around reinterpretation of the "macros" which referenced the dsa's (c-stack-p etc.). To define these, without recourse to the new ghost variable, will require a number of new context functions (these are not given here: see Section 6).

There is, however, another step of development which can be conveniently handled at this point. In the eventual code a goto statement will be modelled by a reset of the stack pointer (prno()) and a branch. This direct treatment is possible, in the current language, even for goto statements which abnormally terminate blocks and/or procedures.

The current trap exit's are used (ignoring those which only result in error) as follows:

MF2 m-int-block : reset prno() (changes to ghost variables can be ignored)

MF5 m-eval-proc-dcl : reset prno()

MF70 m-int-nmd-stmt-list : ascertain if target of goto is local to the current list (dynamic occurrence thereof) and, if so, resume at the correct statement.

MF10 m-int-call : reset display
reset drno
destroys dsa

Firstly, notice that the "destruction" of dsa's is now automatic in resetting the stack pointer.

The main part of the work consists in showing that if the following two functions are used, the result is basically unchanged:

```
4  m-int-goto2(<id>,dict,rs) =

     let <lab,t-drno,t-dp,> = dict(id);
     let drno = cx-drno(id),
         dp   = cx-dp(id);
     if drno ≠ t-drno ∨ dp ≠ t-dp
        then (rest-wrs(cx-act-wrs(id));
              r(p) := c2-stack-p(c(t-drno),t-dp)
             )
     let t-lab-den = mk-lab-den(mk-aid(c(t-drno),t-dp),retr-nm(lab));
     exit(t-lab-den)

   type: Goto Dict Reg-no-set =>
```

```
5  m-int-nmd-stmt-list2(al) = m-int-nmd-stmt-list1(al)
```

The trap exit units of all other functions are made to simply propogate the exit.

A slight extension (cf. MS8) is made to dictionary entries for labels so that they contain the register numbers of the display of the block in which a label was declared:

```
6  Lab-de :: c-Label Reg-no Depth Reg-no-set
```

And the new ghost variable contains:

```
7  Lab-inf :: Dsa-p Reg-no-set m-Sc-loc
```

This introduces a consistency notion for m-Σ2 (notice, this property is only true because it is not possible to pass labels as parameters):

```
8  cons-m-Σ2(dict) =

       mk-lab-de(lab,drno,dp,drns) ∈R dict ⊃
          (let <dsap,rns,p> = c g-li(lab);
             c(drno) = dsap ∧
             c2-stack-p(dsap,dp) = p ∧
             drns = rns)

   type: Dict -> (m-Σ2 -> Bool)
```

The sequence of states m-σ2 now generated differ slightly from those of section 4. In particular, there are certain states arising during abnormal block exit where the registers of an m-σ2 state are no longer in step with those of the corresponding m-σ1!! The only way to retrieve states of m-σ1 is, then, to locate the places where this information is sure to be found. In this case, both the stack pointer and the display registers can be recovered from the dsa's:

```
9  retr-Σ1(dsap,dp,rns)=

       <retr-stg1,
        [prno() ~ c2-stack-p(dsap,dp)] ⊎ [c2-disp-copy(dsap) | rns],
        retr-env1,
        c cc>

   type: Dsa-p Depth Reg-no-set -> (m-Σ2 -> m-Σ1)
```

It is now possible to show:

```
       !! in-step2(c(cx-drno),cx-dp,cx-disp-regs)
   m-i2
       !! m-i1
       !! in-step-a2(c(cx-drno),cx-dp,cx-disp-regs)
```

where:

```
10 in-step2(dsap,dp,drns)(m-σ2,m-σ1) =

       m-σ1 = retr-Σ1(dsap,dp,drns)(m-σ2)
```

```
11 in-step-a2(dsap,dp,drns)(<m-σ2,a2>,<m-σ1,a1>) =

       a2 = a1 ∧
       a2 = nil ⊃ in-step2(dsap,dp,drns)(m-σ2,m-σ1) ∧
       a2 ≠ nil ⊃
            (let <<dsap',dp'>,id> = a2;
             let <dsap',rns,p> = g-li(m-σ2)(cx-nm-lab(id));
             in-step2(dsap',dp',rns)(m-σ2,m-σ1))
```

Section: STAGE II

The proof should be straightforward. The interesting case is MG5 where the trap exit can cancel the abnormal value exactly under conditions which establish the context given in-step2.

Since the trap exit units now cause no state changes the exit can be eliminated and c-int-goto can perform a direct branch.

This section ended on the previous page.

6. MAP TO CODE SEQUENCES

This section documents the mapping from abstract programs to sequences of machine instructions.

The argument for correctness is indicated by annotating with the meta-language expressions which were used to describe a transformation on the state of the object machine. The argument is a local one and is simply that the foregoing sequence of machine instructions achieve the given transformation. (The instruction format used is not exactly that of the assembler: brackets have been inserted and length operands moved to facilitate a mechanical check. Furthermore, some instructions are used in an indexed form (e.g. MVXC) where an obvious expansion is required into a load address and non-indexed instruction).

6.1 Context Functions

References to formulae of this section are of the form "CCn". The following context functions are extra to those given in section 4.3 and are used to give the format of the dsa.

```
1  cx-parm-off(pr,i) =

     /* offset in dsa for i'th parameter, where:
          first 12 bytes of dsa skipped
          4 bytes per loc
          8 bytes per pden                    */

   type: Proc Intg -> Intg


2  cx-arg-off(t,i) =

     /* similar to CC1 but assumes arguments match the parameters */

   type: Call Intg -> Intg


3  cx-dsa-regs(m,lim,dp) =

   type: M (max|min) Depth -> Reg-no


4  cx-dsa-store-off(t,dp) =

     /* offset in dsa for display register copy of this block
        (i.e. depth ≠ 0 gives array base registers).          */

   type: (Proc|Block) Depth -> Intg


5  cx-p-off(m,dp) =

     /* offset in dsa for storage of stack pointer copy */

   type: M Depth -> Intg


6  cx-const-disp-1(t) =

     /* length of constant part of the dsa */

   type: M -> Intg


7  c2-stack-p(dsap,dp) =
     /* uses context function (CC5) to determine offset in dsa */

   type: Dsa-p Depth => m-Sc-loc
```

6.2 Maps

References to formulae of this section are of the form "CFn".

```
1  c-int-prog(<prog>) =

       GETMAIN(drno-0);
                   !! r(drno-0) := GETMAIN
       LR(p,drno-0);
                   !! r(p) := c(drno-0)
                   !! env1 := [ ]
       set-constants()
       c-int-stmt(prog,[ ],{})
                   !! m-int-stmt(prog,[ ],{})

   type: Prog -> Instr*


2  c-int-stmt(t,dict,rs) =

       is-block(t) -> c-int-block(t,dict,rs)
       is-if(t)    -> c-int-if(t,dict,rs)
       is-for(t)   -> c-int-for(t,dict,rs)
       is-call(t)  -> c-int-call(t,dict,rs)
       is-goto(t)  -> c-int-goto(t,dict,rs)
       is-assn(t)  -> c-int-assn(t,dict,rs)
       is-null(t)  -> I

   type: Stmt Dict Reg-no-set -> Instr*


3  c-int-block(w,dict,rs) =

       let <dcls,ns-1> = w;
       let drno =
           drns =

       let p-off = cx-p-off(w,dp);
       ST(p,p-off(0,drno));
                   !! r-stack-p(c(drno),dp) := c(p)
       let n-dict = dict + ...;
       let type-dcls,proc-dcls = ...;
       for id∈type-dcls do
          (c-eval-type(dcls(id),id,dict,rs)
                   !! let de:m-eval-type(dcls(id),id,dict,rs)
          );
       let dsa-l = cx-dsa-regs(w,min,dp),dsa-h = cx-dsa-regs(w,max,dp),
                                          dsa-off = cx-dsa-store-off(w,dp);
       STM(dsa-l,dsa-h,dsa-off(drno));
                   !! save-disp(arr-rns,c(drno))
       B(INTB);
       for id∈proc-dcls do
          (cx-proc-lab(id):
           c-eval-proc-dcl(dcls(id),n-dict,drns)
                   !! m-eval-proc-dcl(dcls(id),n-dict,drns)
          );
       INTB:
       c-int-nmd-stmt-list(ns-1,n-dict,rs);
                   !! m-int-nmd-stmt-list1(1,ns-1,n-dict,rs)
       L(p,p-off(0,drno))
                   !! r(p) := c-stack-p(c(drno),dp)

   type: Block Dict Reg-no-set -> Instr*
   note: for last line cf. MP31
```

```
4  c-eval-type(t,id,dict,rs) =

     cases t:
     mk-type(sc-tp,nil) ->
        ()
     mk-type(sc-tp,<bd>) ->
        (let b = cx-arr-b(t);
         let n = nwrno(rs);
         c-eval-a-expr-to(bd,dict,n,rs);
                  !! m-eval-a-expr-to(bd,dict,n,rs)
         allign(p,sc-tp);
         LR(b,p);
         if sc-tp = intq
           then S(b,lit=4)
           else S(b,lit=1);
                  !! r(b) := c(p) - (if sc-tp = intq then 4 else 1)
         if sc-tp = intq
           then SLA(n,2(0));
         AR(p,n);
                  !! r(p) := c(p) + (if sc-tp = intq then c(n) * 4 else c(n))
       )

   type: Type Id Dict Reg-no-set -> Instr*


5  c-eval-proc-dcl(w,dict,em-disp-regs) =

     let <parm-l,st> = w;
     let drno =
     ST(13,(0)(0,p));
                  !! r-ra(c(p)) := return address (cf.call)
     CLC((8)(p),4,lit=0);
     BE(DISP-RESTD);
     let dsa-l' =
         dsa-h' =
         dsa-off' =
     L(n,(8)(p));
     LM(dsa-l',dsa-h',dsa-off'(n));
                  !! if c-sc(c(p)) ≠ 0 then rest-disp(em-disp-regs,c-sc(c(p)))
     DISP-RESTD:
     LR(drno,p);
                  !! r(drno) := c(p)
     let δ = cx-const-disp-l(w);
     A(p,lit=δ);
                  !! r(p) := c(p) + δ
     let n-dict = (dict \ cvs) + ...;
     for i = 1 to l parm-l do
        (let <id,t> = parm-l[i];
         is-array-type(t) ->
           (L(cx-arr-b(id),cx-parm-off(w,i)(0,drno))
                  !! r(cx-arr-b(id)) := c-pm(c(drno),i)
           )
         T -> ()
        )
     let dsa-l = cx-dsa-regs(w,min,0),
         dsa-h = cx-dsa-regs(w,max,0),
         dsa-off = cx-dsa-store-off(w,0);
     STM(dsa-l,dsa-h,dsa-off(drno));
                  !! save-disp(disp-regs,c(drno))
     c-int-stmt(st,n-dict,{});
                  !! m-int-stmt(st,n-dict,{})
     S(p,lit=δ);
                  !! r(p) := c(p) - δ
     L(13,(0)(0,p));
     BR(13)
                  !! return (c-ra(c(p)))

   type: Proc Dict Reg-no-set -> Instr*


70 c-int-nmd-stmt-list(ns-l,dict,rs) =

     for j = 1 to l ns-l do
        (cx-nm-lab(s-nm°ns-l[j]):
         c-int-stmt(s-stmt°ns-l[j],dict,rs))

   type: Nmd-stmt* Dict Reg-no-set -> Instr*
```

```
8  c-int-if(<be,st1,st2>,dict,rs) =

      is-inf-expr(be) ∧ is-rel-op°s-op(be) ->
         (let <e1,op,e2> = be;
          let n1,n2 =
          c-eval-a-expr-to(e1,dict,n1,rs);
                   !! m-eval-a-expr-to(e1,dict,n1,rs)
          c-eval-a-expr-to(e2,dict,n2,rs ∪ {n1});
                   !! m-eval-a-expr-to(e2,dict,n2,rs ∪ {n1})
          let m be the mask s.t. ¬op ≡ retr-test(m);
          CR(n1,n2);
          BC(m,FALSE)
                   !! IF ¬m-apply-rel-op(c(n1),op,c(n2))
                         THEN branch FALSE
         )
      is-rhs-ref(be) ∧ s-sscs°s-var-ref(be) ≠ nil ->
         (let <d,x,b>:c-eval-b-expr(be,dict,rs);
                   !! let <d,x,b>:m-eval-b-expr(be,dict,rs)
          let n2 = nwrno(rs ∪ {x});
          LA(n2,d(x,b));
                   !! n2 =
          TM((0)(n2),m-true);
          BZ(FALSE)
                   !! IF x-val(d(x,b),bool) = m-false
                         THEN branch FALSE
         )
      T ->
         (let <d,0,b>:c-eval-b-expr(be,dict,rs);
                   !! let <d,x,b>:m-eval-b-expr(be,dict,rs)
          TM(d(b),m-true);
          BZ(FALSE)
                   !! IF x-val(d(0,b)),bool) = m-false
                         THEN branch FALSE
         )
      TRUE:
      c-int-stmt(st1,dict,rs);
                   !! m-int-stmt(st1,dict,rs)
      B(END);
                   !! branch END
      FALSE:
      c-int-stmt(st2,dict,rs);
                   !! m-int-stmt(st2,dict,rs)
      END:;

   type: If Dict Reg-no-set -> Instr*
```

```
9  c-int-for(<cv,e-i,e-s,e-l>,dict,rs) =

     let r-s,r-l,r-x = ... s.t. is-even-wrnc(r-s);
     c-eval-a-expr-to(e-s,dict,r-s,rs);
                 !! m-eval-a-expr-to(e-s,dict,r-s,rs)
     c-eval-a-expr-to(e-l,dict,r-l,rs ∪ {r-s});
                 !! m-eval-a-expr-to(e-l,dict,r-l,rs ∪ {r-s})
     c-eval-a-expr-to(e-i,dict,r-x,rs ∪ {r-s,r-l});
                 !! m-eval-a-expr-to(e-i,dict,r-x,rs ∪ {r-s,r-l})
     LTR(r-s,r-s);
     BM(NEGS);
                 !! IF c(r-s) < m-0 THEN branch NEGS
     BZ(INTF1)
                 !! IF c(r-s) = m-0 THEN branch INTF1
     CR(r-x,r-l);
     BH(END);
     B(INTF1);
                 !! IF c(r-x) > c(r-l) THEN branch END
                                      ELSE branch INTF1
     NEGS:
     CR(r-x,r-l);
     BL(END);
                 !! IF c(r-x) < c(r-l) THEN branch END
     INTF1:
     INTST1:
     c-int-stmt(b,dict,rs ∪ {r-s,r-l,r-x});
                 !! m-int-stmt(b,dict,rs ∪ {r-s,r-l,r-x})
     LTR(r-s,r-s);
     BM(NEGS1);
                 !! IF c(r-s) < m-0 THEN branch NEGS1
     BZ(INTST1)
                 !! IF c(r-s) = m-0 THEN branch INTST1
     BXLE(r-x,r-s,INTST1)
     B(END1)
                 !! (r(r-x) := c(r-x) + c(r-s);
                     IF c(r-x) ≤ c(r-l) THEN branch INTST1
                                        ELSE branch END1)
     NEGS1:
     BXH(r-x,r-s,INTST1);
     CR(r-x,r-l);
     BE(INTST1);
                 !! IF c(r-x) ≥ c(r-l) THEN branch INTST1
     END1:
     END:

type: For Dict Req-no-set -> Instr*
note: cf. MP9 and MP90 which are here merged.
      requirement for even-wrnc not reflected in MP9.
```

```
10 c-int-call(w,dict,rs) =

      let <pid,al> = w
      let drno =
      for i = 1 to l al do
         (let off = cx-arg-off(w,i);
          is-var-ref(al[i]) ->
            (let <d,x,b> = c-eval-var-ref(al[i],dict,rs);
                 !! let <d,x,b>:m-eval-var-ref(al[i],dict,rs)
             let n = nwrno(rs ∪ {x});
             LA(n,d(x,b));
             ST(n,off(0,p))
                 !! r-pm(c(p),i) := d(x,b)
            )
          is-id(al[i]) ->
            (cases dict(al[i]):
             mk-prop-proc-de(L,,em-drnc) ->
               (ST(em-drno,off(0,p)) ->
                ST(addr(L),(off+4)(0,p));
                   !! r-pm(c(p,i)) := <c(em-drno),L>
               )
             mk-parm-proc-de'(d',b) ->
               (MVC(off(p),8,d'(b))
                   !! r-pm(c(p),i) := c-pm(c(b),d)
               )
            )
         );
      ST(drno,(4)(0,p));
                   !! r-dc(c(p)) := c(drno)
      store-wrs(rs);

      cases dict(pid):
      mk-prop-proc-de'(L,em-rns,em-drno,dsa-l) ->
         (MVC((8)(p),4,lit=0);
                   !! r-sc(c(p)) := 0
          BAL(13,L);
                   !! call L(r-ra(c(p)))
          L(drno,(4)(0,p));
                   !! r(drno) := c-dc(c(p))
          let dsa-h = cx-dsa-regs(w,max,dp);
              dsa-off = cx-dsa-store-off(w,dp);
          LM(dsa-l,dsa-h,dsa-off(drno));
                   !! rest-disp(drno \ em-rns,c(drno))
          rest-wrs(rs)
         )
       mk-parm-proc-de'(d',b) ->
         (          !! let <em-dsa,L> = ...
          MVC((8)(p),4,d'(b));
                   !! r-sc(c(p)) := em-dsa
          L(13,(d' + 4)(0,b));
          BALR(13,13);
                   !! call L(r-ra(c(p)))
          L(drno,(4)(0,p));
                   !! r(drno) := c-dc(c(p))
          let dsa-l = cx-dsa-regs(w,min,0),
              dsa-h = cx-dsa-regs(w,max,dp),
              dsa-off = cx-dsa-store-off(w,dp);
              LM(dsa-l,dsa-h,dsa-off(drno));
                   !! rest-disp(drno,c(drno))
          rest-wrs(rs)
         )

   type: Call Dict Reg-no-set -> Instr*
```

```
11 c-int-goto(<id>,dict,rs) =

    let <lab,t-drno,t-dp,p-off> = dict(id);
            /* p-off from declaring block */
    let drno,dp = ...;
    if drno ≠ t-drno ∨ dp ≠ t-dp
       then
          (rest-wrs(cx-act-wrs(id));
           LA(n,p-off(0,t-drno));
           L(p,0(0,n))
                !! r(p) := c2-stack-p(c(t-drno),t-dp)
          )
    B(cx-nm-lab(id))
                !! exit, trap of m-int-nmd-stmt-list

type: Goto Dict Reg-no-set -> Instr*
note: cf. MG4


12 c-int-assn(<vr,e>,dict,rs) =

    let <d,x,b>:c-eval-var-ref(vr,dict,rs);
                !! let <d,x,b>:m-eval-var-ref(vr,dict,rs)
    let rs' = ...
    (cases vr-tp(vr):
     intg ->
        (let n = nwrno(rs');
         c-eval-a-expr-to(e,dict,n,rs');
                !! m-eval-a-expr-to(e,dict,n,rs')
         ST(n,d(x,b))
                !! set(d(x,b),c(n),intg)
        )
     bool ->
        (if no-share-poss(vr,e)
           then
              (c-eval-b-expr-to(e,dict,<d,x,b>,rs')
                !! m-eval-b-expr-to(e,dict,<d,x,b>,rs')
              )
           else
              (let <d',x',b'>:c-eval-b-expr(e,dict,rs');
                 !! let <d',x',b'>:m-eval-b-expr(e,dict,rs')
               MVXC(<d,x,b>,1,<d',x',b'>)
                 !! set(d(x,b),x-val(d'(x',b'),bool),bool)
              )
        )
    )

type: Assn Dict Reg-no-set -> Instr*
```

```
151 c-eval-a-expr-to(e,dict,n,rs) =

      cases e:
      mk-inf-expr(e1,op,e2) ->
         (c-eval-a-expr-to(e1,dict,n,rs):
                  !! m-eval-a-expr-to(e1,dict,n,rs)
          (cases e2:
           mk-inf-expr() ->
            (let nn = nwrno(rs ∪ {n});
             c-eval-a-expr-to(e2,dict,nn,rs ∪ {n});
                  !! m-eval-a-expr-to(e2,dict,nn,rs ∪ {n})
               (cases op:
                add -> AR(n,nn)
                sub -> SR(n,nn)
               )
                  !! r(n) := m-apply-intg-op(c(n),op,c(nn))
            )
           mk-rhs-ref(vr) ->
            (let <d,x,b>:c-eval-var-ref(vr,dict,rs ∪ {n});
                  !! let <d,x,b>:m-eval-var-ref(vr,dict,rs ∪ {n})
               (cases op:
                add -> A(n,d(x,b))
                sub -> S(n,d(x,b))
               )
                  !! r(n) := m-apply-intg-op(c(n),op,x-val(d(x,b),intg))
            )
           mk-cv-ref(id)
            (let <r> = dict(id);
               (cases op:
                add -> AR(n,r)
                sub -> SR(n,r)
               )
                  !! r(n) := m-apply-intg-op(c(n),op,c(r))
            )
           T -> /* is-const(e2) */
            (let d = cx-const-off(e2);
               (cases op:
                add -> A(n,d(0,drno-0))
                sub -> S(n,d(0,drno-0))
               )
                  !! r(n) := m-apply-intg-op(c(n),op,x-val(d(0,drno-0),intg)))
            )
          )
         )

      mk-rhs-ref(vr) ->
         (let <d,x,b>:c-eval-var-ref(vr,dict,rs ∪ {n});
                  !! let <d,x,b>:m-eval-var-ref(vr,dict,rs ∪ {n})
          L(n,d(x,b))
                  !! r(n) := x-val(d(x,b),intg)
         )
      mk-cv-ref(id) ->
         (let <r> = dict(id);
          LR(n,r)
                  !! r(n) := c(r)
         )
      T -> /* is-const(e) */
         (let d = cx-const-off(e);
          L(n,d(0,drno-0))
                  !! r(n) := x-val(d(0,drno-0),intg))
         )

   type: Expr Dict Reg-no Reg-no-set -> Instr*
```

```
152 c-eval-b-expr(e,dict,rs) =

    let b = cx-drno(e);
    cases e:
    mk-inf-expr() ->
       (let d = cx-temp-off(c);
        c-eval-b-expr-to(e,dict,<d,0,b>,rs);
              !! m-eval-b-expr-to(e,dict,<d,0,b>,rs)
        return (<d,0,b>)
       )
    mk-rhs-ref(vr) ->
       (let l:c-eval-var-ref(vr,dict,rs);
              !! let l:m-eval-var-ref(vr,dict,rs)
        return (l)
       )
    T -> /* is-const(e) */
       (let d:cx-const-off(e);
        return (<d,0,drno-0>)
       )

type: Expr Dict Reg-no-set -> Instr* c-Opd


153 c-eval-b-expr-to(e,dict,<d,x,b>,rs) =

    cases e:
    mk-inf-expr(e1,op,e2) ->
       (is-bool-op(op) ->
          (c-eval-b-expr-to(e1,dict,<d,x,b>,rs);
                !! m-eval-b-expr-to(e1,dict,<d,x,b>,rs)
           let <d',x',b'>:c-eval-b-expr(e2,dict,rs);
                !! let <d',x',b'>:m-eval-b-expr(e2,dict,rs)
             (cases op:
              and -> NXC(<d,x,b>,1,<d',x',b>)
              or  -> OXC(<d,x,b>,1,<d',x',b>)
             )
                !! set(d(x,b),...)
          )
        is-rel-op(op) ->
          (let n1,n2 = ...;
           c-eval-a-expr-to(e1,dict,n1,rs);
                !! m-eval-a-expr-to(e1,dict,n1,rs)
           c-eval-a-expr-to(e2,dict,n2,rs ∪ {n1});
                !! m-eval-a-expr-to(e2,dict,n2,rs ∪ {n1})
           CR(n1,n2);
                !! cc := m-apply-rel-op(c(n1),op,c(n2))
           let m = ...;
           (BC(m,TRUE);
            MVXI(<d,x,b>,1,m-false);
            B(END);
            TRUE:
            MVXI(<d,x,b>,1,m-true);
            END:
           )
                !! set(d(x,b),retr-truth(c cc),bool)
          )
       )
    mk-rhs-ref(vr) ->
       (let <d',x',b'>:c-eval-var-ref(vr,dict,rs);
        MVXC(<d,x,b>,1,<d',x',b>)
       )
    T -> /* is-const(e) */
       (let d' = cx-const-off(e);
        MVXC(<d,x,b>,1,<d',0,drno-0>))

type: Expr Dict c-Opd Reg-no-set -> Instr*
```

```
18 c-eval-var-ref(vr,dict,rs) =

      let <id,ssc-l> = vr;
      cases ssc-l:
      nil ->
          (cases dict(id):
           mk-prop-sc-de(d,b) ->
             return (<d,0,b>)
           mk-parm-sc-de'(d',b) ->
             (let n = nwrno(rs);
              L(n,d'(0,b));
                    !! r(n) := c-pm(c(b),d)
              return (<0,0,n>)
             )
           mk-prop-arr-de(b) ^ mk-parm-arr-de(,,b) ->
             return (<0,0,b>)
          )
      <ssc> ->
          (let b = s-base(dict(id));
           let n = nwrno(rs);
           c-eval-a-expr-to(ssc,dict,n,rs);
                    !! m-eval-a-expr-to(ssc,dict,n,rs)
           if vr-tp(vr) = intg
             then (SLA(n,2(0))
                    !! r(n) := c(n)*4
                   );
           return (<0,n,b>)
          )

   type: Var-ref Dict Reg-no-set -> Instr* c-Opd
```

7. DISCUSSION

7.1 Comments on Current Report

This report represents the last (for the moment, at least) of a series of experiments in how a correctness argument for a compiler can be related to a language definition. From a comparison of the various experiments, many of which are not printed, a number of comments can be made.

In the current report the separation of language concepts is poor (cf. ref 3, for example). The cause of this is largely the excessive formality of the argument documented. The idea of separating language topics does not eliminate the need to check for undesirable interactions, but it should permit a somewhat less formal check than is, for example, required to ascertain that executing assignment statements preserves all the relationship between source and target environments.

Another difference from ref. 3 is the relatively low number of development steps employed here. In the same way that the development of the mapping of for statements is handled here, several (manuscript) versions decomposed the development of the block part (cf. also ref 2). One possible direction for such a split is to generate dsa's (perhaps via declare) which are themselves maps from offsets to values. At this stage, only address computation is considered and even parameter addresses can be passed as such. In the next stage the problems of representing this map in linear storage and of storing parameter addresses can be considered.

The choice of how to develop static information about abstract programs is still fairly open. There are three possibilities:

1) Generate in an environment which also contains dynamic information (the recording of types in Env is in this category).

2) Generate, still in the main definition of mapping functions, a separate component which contains only static information (Dict is of this sort).

3) Define via context functions (examples include ex-tp and cx-dp).

That the choice is somewhat arbitrary can be seen by observing that the whole dictionary could be made redundant by context functions which yielded the attributes of uses of identifiers.

In connection with the above choices, one word of warning is in order. If there is more than one way of deriving the same information a consistency condition must exist. Whilst it is often a "short cut" to insert a new component with exactly the required information, the subsequent work in showing that the new component is compatible with that with which it partially overlaps frequently makes it worthwhile redesigning the whole set.

7.2 Comparison to Ref.8

In considering the current report some special comments can be made in comparing it with ref.8.

One important change is that the earlier report always used relations between source objects and their representations. In the current report the one-many nature of some relations has been emphasised by the use of "retrieve functions". Even in a trivial example like that of retr-val, more properties are immediately apparent.

Associated with this change is the move from auxiliary arguments containing correspondences (in ref. 8, lc = LOC<->m-LOC, roughly, and had to be passed as an auxiliary argument to many of the equivalence relations) to the use of ghost variables in the target state (cf. MS21 which facilitates retrieval of Stg).

In ref. 8 this part of storage containing dsa type information was to have been extracted by the function x-env. In stage I of the current document, all such parts of store are separated into the Env1 component.

In ref.8 certain predicates (e.g. q-tr) served both as predicates on the target state as such and on its relation to the source state. Although such an approach would certainly have saved a number of lines in the current report, it was considered clearer to separate the issues.

The somewhat pedantic deductions of Ref 8 regarding is-wf-0 have been dropped here since the predicate must be true for any sub-part of a well formed program.

7.3 Further Work

There is a whole spectrum of topics which require further investigation. At the more general end there is the problem of the degree of difficulty to react to changes during such a formal development. The top-down approach would quickly lose many of its advantages if changes to the source language occurred all of the way through a project. On the other hand the precise documentation available should make it possible to quickly estimate the impact of any reasonable change. The topic for further research is to find guidelines for the development which retain a reasonable degree of flexibility.

Because of the recursive nature of the predicates used, their existence should be demonstrated. A prerequisite for this work is a more careful treatment of the meta-language used. A system whose foundations are clearly given can be expected in ref. 12.

A point in the spectrum where the task is more clearly defined but the problem is far from trivial is the formal description of all combinators used; the valid forms of reasoning over same; and derived lemmas which are useful as rules of thumb.

An example of a detailed problem, relating to an earlier version of this report, is that of retrieving functions.

Given:

$g : S \to S$
$retr\text{-}S : L \to S$

the model:

$m\text{-}g : L \to L$

is proven correct, roughly, by:

$$(\forall l)(retr\text{-}S \circ m\text{-}g(l) = g \circ retr\text{-}S(l))$$

Suppose however, a model is sought for:

$h : D \to (S \to R)$

then:

$m\text{-}h : D \to (L \to R)$

appears to require a retrieve function:

$retr\text{-}f : (L \to R) \to (S \to R)$

which could be defined:

$$retr\text{-}f(m\text{-}f) = m\text{-}f \circ retr\text{-}S^{-1}$$

But the inverse of this retrieve function is not likely to be a function (cf. Sets -> their list representations)! Thus it is easier to state:

$$retr\text{-}f(m\text{-}f) = (\cup f)(f \circ retr\text{-}S = m\text{-}f)$$

but this introduces the problem of establishing existence (i.e. f is a function) and uniqueness. Versions of retr-proc-den were written, and these in turn made it possible to define retr-env, but appeared more complicated than the approach adopted here.

One place where an immediate improvement could be made is by the employment of a new combinator for situations where some action should follow either normal or abnormal termination of the preceding action without losing the knowledge of which one. For example in int-block:

```
int-bl() =
   prologue;
   (m-int-nmd-stmt-list();
    epi())
```

where:

$op1;op2 \simeq \lambda\sigma.$ (let $\langle\sigma', abn'\rangle = op1(\sigma)$
let $\langle\sigma'', \rangle = op2(\sigma')$
$\langle\sigma'', abn'\rangle$)

The saving in the definition is small, the avoidance of duplication in reasoning with AL3 is more significant.

A further area of useful research would be the design of a computer based editing system which would ease generation (e.g. automatically producing versions without assertions) and updating (the numbering of some formulae above indicates a particularly tedious problem).

APPENDIX : Function Index

prefix	section	subject
AL	2.2	Lemma on "annotation"
DA	3.1	Definition: Abstract Syntax
DC	3.2	Definition: Context Conditions
DS	3.3	Definition: States
DF	3.4	Definition: Semantic Functions
MS	4.2	Map: State and Dictionary
MC	4.3	Map: Context Conditions
P	4.4	Proof Notes
L	4.5	Lemmas
MF	4.6	Map: Functions
MG	5	Stage II Map
CC	6.1	Code Sequence: Context Conditions
CF	6.2	Code Sequence: Functions

There now follows an, alphabetical, index to the defined functions:

Section: APPENDIX I

Section: APPENDIX I

Stg	3.3(3)
Stmt	3.1(2)
Tr	3.3(13)
Tr1	4.2(31)
Type	3.1(4)
Val	3.3(2)
Var-ref	3.1(18)
Σ	3.3(1)

This section ended on the previous page.