CHAPTER 8

COMPILER DESIGN

The process of developing programs from their specifications is described in chapter 10. This chapter discusses the special problem of using a denotational language definition in the design of a compiler. In particular, the examples show that a VDM definition can be used to justify a mapping from the source language to sequences of instructions of the object machine. Such a mapping can then be used as a specification of the translation process: the techniques of chapter 10 being applicable to the development of a translator. One can see such a compiler design approach as isolating the use of the semantics to the first stage. As normal with such proposals, the top-down description is an over-simplification: it provides a documentation structure rather than a constraint on thinking. The examples are taken from the language of chapter 4. The proofs are not given at a very formal level.

This chapter is a rewritten version of [Jones 78c]. Other relevant references are [McCarthy 67a, Lucas 68a, Jones 71a] (this last paper contains further references to the early Vienna work in this area), [Jones 76a, Milne 76a, Morris 73a] and [Mosses 76a].

CONTENTS

8.1 Introduction.....255
8.2 Expression Evaluation - Language.....257
8.3 Expression Evaluation - Target Machine.....259
8.4 Expression Evaluation - Code Sequences.....263
8.5 Location References - Language.....266
8.6 Location References - Target Machine.....267

8.1 INTRODUCTION

A definition of a language is written in terms of semantic objects (e.g. state) which are abstract in the sense that they possess only properties which are essential to the semantics. The first task in compiler design is to choose how these objects are to be represented on the target machine. Just as in conventional "data refinement" the representation is likely to have extra information, necessary to manipulate data efficiently, which are not present in the abstraction. In the ideal case, the representation can be related to the abstraction using a "retrieve function" (chapter 10 contains a detailed explanation of data refinement proofs).

The semantics of the language can be redefined in terms of the chosen representation. The proof of correctness of this alternative definition is given by relating it to the original semantics as shown in Fig. 1.

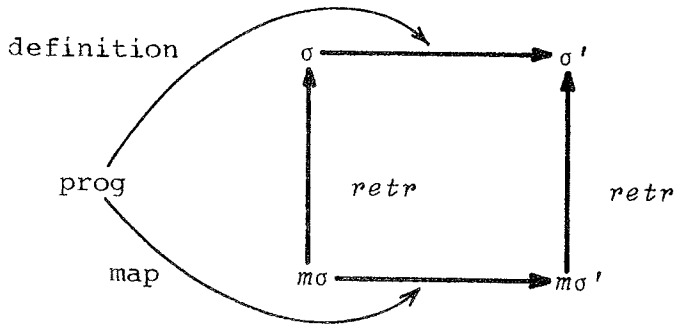

Fig. 1: First Stage of Compiler Correctness

The original definition maps abstract programs into state transformations (σ, σ' are members of the abstract states). The chosen state representation ($m\sigma$, $m\sigma'$ as typical members) is related to the abstract state by a retrieve function (*retr*). It is necessary to establish that this diagram commutes in the sense that the state transformation on the representation states is the same, when viewed under the retrieve function, as that on the abstract states. This form of justification should be decomposed to treat one language "concept" at a time.

At the end of the first stage, it is known what transformations are to be made in the target machine for particular language constructs. The next task is to choose sequences of target machine instructions which realize these transformations. Given an understanding of the semantics of the object machine instructions, the correctness of a translator specification is illustrated in Fig. 2.

This shows that the state transformations which are required on the object machine should be identical with the effect of the sequences of object machine instructions generated. Once again, such an argument must be decomposed.

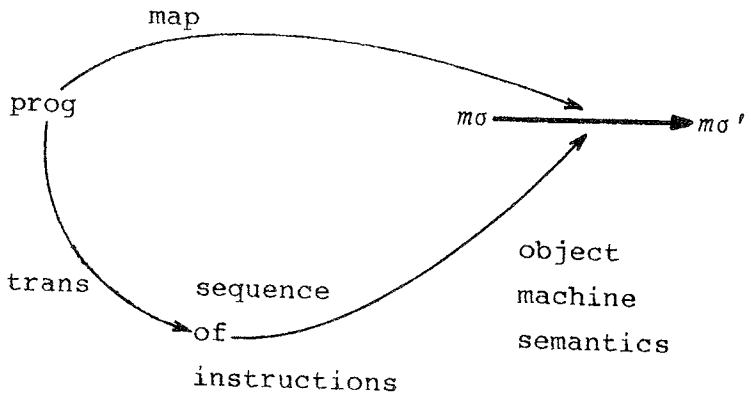

Fig. 2: Translator Correctness

The understanding of the object machine would not normally have to be recorded in a complete formal semantics. However, for an unconventional machine, it may be necessary to record assumptions on its architecture in terms of a state.

As with most descriptions of "top-down" design, the above is a simplification. It would clearly be unwise to choose a representation of the state without considering the instruction set of the target machine. Such a lack of foresight could result in choosing transformations which could only be realized inefficiently on the object machine. Clearly, a compiler designer will often sketch "instruction sequences" for sample programs as one of the first attempts to understand the translation problem. The structure described above is, then, one for presenting clear documentation (including correctness argument) rather than a constraint on thinking.

Compiler design documentation thus breaks into three major parts. The design of the object time environment solves the problem of representing abstract state objects (e.g. mappings) on the chosen object machine. This "object machine" may be an extension of the hardware to include commonly required service routines. The choice of the (extended) machine instructions to achieve the required transformations gives rise to a translator specification. The development of the translator itself is not covered here but is discussed in Chapter 9.

The program as so far described would only be practical for very small languages. For an actual programming language, it is essential that the design is decomposed so that different "language concepts" can be treated independently. This desirable separation is not fully formalized. However, the monolithic proof of [Jones 76a] has been decomposed to consider the following concepts separately:

i) Expression evaluation
ii) Location reference
iii) Environment handling
iv) Statement sequencing
v) Input/Output

The first of these concepts has been described by several authors and is presented as the first example below. The example gives the opportunity to discuss the topic of different forms of optimization. Location reference and Environment handling have often been grouped together as in [Henhapl 70a]. The former concept provides the second example in this chapter and shows how an implementation might resolve non-determinacy in the specification. Environment handling (or "reference to automatic variables") has been extensively studied by the Vienna group (see [Lucas 68b, Henhapl 70a, Jones 71a]). Statement sequencing is discussed in [Zimmerman 70a] & [Bjørner 78a]. Input/Output -- as always the poor relation -- is not normally the subject of papers because of the reliance on details of the chosen operating system interface.

8.2 EXPRESSION EVALUATION - LANGUAGE

The denotational semantics of expression evaluation avoids the issue of allocation of space for intermediate values. This is quite proper in the semantics, but it is an important issue which must be resolved in the design of the object time state. The parts of the language of chapter 4 which are of concern here are given by the following abstract syntax:

Assign :: *Varref* *Expr*
Expr = *Infixexpr* | *Rhsref*
Infixexpr :: *Expr* *Op* *Expr*
Op = *Intop* | *Boolop* | *Comparisonop*
Rhsref :: *Varref*
Varref :: *Id* [*Expr*$^+$]
Scalartype = INT | BOOL

The relevant context conditions from chapter 4 are also assumed, as is the auxiliary function *SCTP*. Notice that two different ways of isolating this language concept are being used. The detailed list of operators is not discussed. This relies on the tacit assumption that the target machine has an adequate set of instructions. If, for example, there were no division instruction, the control of temporaries for the expanded code would probably have to be considered here. Another "interface" to the expression evaluation concept is the calculation of addresses for elements of arrays (see below): although this can be set aside, the implications for the use of temporaries must be considered here. It is for this reason that the use of array references on the left of assignment statements is included in the language fragment.

The relevant semantic objects (again taken from chapter 4) are:

$$
\begin{array}{lcl}
STATE & :: & \text{STR}:STORE \ \ldots \\
STORE & = & SCALARLOC \xrightarrow{m} [SCALARVALUE] \\
SCALARVALUE & = & Bool \mid Int \\
ENV & = & Id \xrightarrow{m} DEN \\
DEN & = & LOC \mid \ldots \\
LOC & = & ARRAYLOC \mid SCALARLOC \\
ARRAYLOC & = & Nat^{+} \xrightarrow{m} SCALARLOC^{+}
\end{array}
$$

Elements of *ARRAYLOC* are one-one mappings and their domains form a rectangle:

$$rect: \ Nat^{*} \rightarrow (Nat^{*})\text{-}set$$

The relevant semantic functions are:

$$M[mk\text{-}Assign(vr,e)](\rho) \triangleq$$
$$\quad \underline{def}\ l : Mloc[vr](\rho);$$
$$\quad \underline{def}\ v : M[e](\rho);$$
$$\quad \text{STR} := \underline{c}\ \text{STR} + [l \mapsto v]$$

$$M[mk\text{-}Infixexpr(e1,op,e2)](\rho) \triangleq$$
$$\quad \underline{def}\ v1 : M[e1](\rho);$$
$$\quad \underline{def}\ v2 : M[e2](\rho);$$
$$\quad \underline{return}(M[op](v1,v2))$$

$$M[mk\text{-}Rhsref(vr)](\rho) \triangleq (\underline{def}\ l : Mloc[vr](\rho);\ contents(l))$$

```
Mloc[mk-Varref(id,sscl)](ρ) ≜
    if sscl=nil then return(ρ(id))
    else let aloc = ρ(id) in
         def esscl : <M[sscl[i]](ρ) | i∈indsesscl>;
         if ¬(esscl∈domaloc) then error else return(aloc(esscl))

contents(l) ≜ def v : (c STR)(l);
              if v=nil then error else return(v)
```

There are two places above where a program can be in error in a way which cannot be detected statically. In both cases, the design chosen below does not detect the error. This is a design decision and has nothing to do with the development method. Those who consider the decision too dangerous to be acceptable might choose to outline an alternative design.

8.3 EXPRESSION EVALUATION - TARGET MACHINE

The representation of store (as a byte string) is discussed in the second example below. Here this problem is avoided by showing the *STORE* in its abstract form. What cannot be postponed is providing space for the temporaries which are implied by *let vi* in the semantics. In a typical von Neumann computer such temporaries are most conveniently stored in registers. The initial development assumes that there are "enough" such registers and the problems posed by a paucity are discussed as optimization. With some care in the description, it is possible to leave open the possibility that the same registers are used for other purposes (e.g. address calculation). The use of registers is, however, assumed to be on "stack" basis.

The state of the target machine is then:

```
STATE_m :: STR:  STORE  ...
           REGS: (Regno ⫛ SCALARVALUE)
```

In order to defer decisions about address calculation, a function is assumed which computes locations given store references and registers:

```
sr:  STOREREF × (Regno ⫛ SCALARVALUE) → SCALARLOC
```

In a machine with registers and store there are likely to be a range of instructions. Taking something like an IBM System/360 as a model, there might be register to register, register to store, and store to store instructions.

As is observed above, the specific instructions available have a great influence on the way in which the (alternative) definition based on target machine states is defined. Had there been three address instructions available, a different transformation would have been defined. The case distinctions given here are rather coarse and give rise to somewhat pessimistic code; the subject of optimization is considered below. A function for choosing new registers is defined:

$$newreg: Regno\text{-}set \to Regno$$
$$newreg(rs)=r \supset \neg(r \in rs)$$

The semantic functions for the machine state might be (*tgi* indicates that a target is provided and that an integer expression is being evaluated - cf. [Jones 76a] for all cases):

$$Mc[mk\text{-}Assign(vr,e)](\delta,rs) \triangleq$$
$$\quad \underline{def}\ (ref,rs') : MCloc[vr](\delta,rs);$$
$$\quad \underline{if}\ SCTP[vr] = \text{INT}\ \underline{then}\ \underline{let}\ nr = newreg(rs')\ \underline{in}$$
$$\qquad MCtgi[e](\delta,nr,rs');$$
$$\qquad \underline{def}\ l : sr(ref,\underline{c}\ \text{REGS});$$
$$\qquad \text{STR} := \underline{c}\ \text{STR} + [l \mapsto (\underline{c}\ \text{REGS})(nr)]$$
$$\quad \underline{else}\ \ldots$$

$$MCtgi[mk\text{-}Infixexpr(e1,op,e2)](\delta,r,rs) \triangleq$$
$$\quad \underline{let}\ nr = newreg(rs \cup \{r\})\ \underline{in}$$
$$\quad MCtgi[e1](\delta,r,rs);$$
$$\quad MCtgi[e2](\delta,nr,rs \cup \{r\});$$
$$\quad \text{REGS} := \underline{c}\ \text{REGS} + [r \mapsto M[op]((\underline{c}\ \text{REGS})(r),(\underline{c}\ \text{REGS})(nr))]$$

$$MCtgi[mk\text{-}Rhsref(vr)](\delta,r,rs) \triangleq$$
$$\quad \underline{def}\ (ref,rs') : MCloc[vr](\delta,rs);$$
$$\quad \underline{def}\ l : sr(ref,\underline{c}\ \text{REGS});$$
$$\quad \text{REGS} := \underline{c}\ \text{REGS} + [r \mapsto contents(l)]$$

These functions use a dictionary which provides statically determinable information about identifiers:

$$\delta \in DICT = Id \xrightarrow{m} DICTINF$$

The retrieve function which relates the semantic objects here with those of the definition is of type:

$$retr: DICT \times STATE_m \to ENV \times STATE$$

with:

$$retr(\delta,\sigma m) \triangleq (retrENV(\delta, REGS(\sigma m)), retrSTATE(\sigma m))$$

$$retrENV: \quad DICT \times (Regno \xrightarrow{m} SCALARVALUE) \to ENV$$
$$retrSTATE: \; STATE_m \to STATE$$

The details of these functions need not be given here. The overall result to be shown for assignment statements is:

$$\begin{aligned}((\rho,\sigma) = retr(\delta,\sigma m)) \supset (&\underline{let}\ \sigma m' = MC[assn](\delta,rs)(\sigma m)\ \underline{in}\\ &\underline{let}\ \sigma' = M[assn](\rho)(\sigma)\ \underline{in}\\ &(\sigma' = retrState(\sigma m')) \wedge regs(rs,\sigma m,\sigma m'))\end{aligned}$$

where *regs* is a predicate which checks that two states agree over a given register set:

$$regs(rs,\sigma m,\sigma m') = \mathrm{REGS}(\sigma m)|rs = \mathrm{REGS}(\sigma m')|rs$$

This predicate is used to define the stack nature of register use.

Consulting the *M* and *MC* semantics for assignment statements, it is clear that something must be assumed about the computation of locations. In addition to the obvious properties requiring that Fig. 1 commutes and the location "corresponds" the overall proof requires that no registers are "freed":

$$\begin{aligned}(\rho,\sigma)=retr(\delta,\sigma m) \supset (&\underline{let}\ (\sigma m', ref, rs') = MCloc[vr](\delta,rs)(\sigma m)\ \underline{in}\\ &\underline{let}\ (\sigma', l) = Mloc[vr](\rho)(\sigma)\ \underline{in}\\ &\sigma' = retrSTATE(\sigma m') \wedge rs \subseteq rs' \wedge\\ &l = sr(ref, \mathrm{REGS}(\sigma m') \mid rs') \wedge regs(rs,\sigma m,\sigma m'))\end{aligned}$$

This assumption documents one of the interfaces of the expression evaluation language concept. A related property about expression evaluation

itself is required to prove the overall result about assignment. It is:

$$((\rho,\sigma)=retr(\delta,\sigma m) \wedge \neg(r\in s) \wedge (SCTP[e]=\underline{INT})) \supset$$
$$(\underline{let}\ (\sigma m',vm) = MCtgi[e](\delta,r,rs)(\sigma m)\ \underline{in}$$
$$\underline{let}\ (\sigma',v) = M[e](\rho)(\sigma)\ \underline{in}$$
$$\sigma'=retrState(\sigma m') \wedge v=\mathrm{REGS}(\sigma m')(r) \wedge regs(rs,\sigma m,\sigma m'))$$

From this, and the fact that:

$$rs' \supseteq rs$$

the assignment result follows without difficulty.

The proof of the required result about expressions can be performed by structural induction over the syntactic structure of *Expr*. For elements of *Infixexpr*, the proof relies on the preservation of the registers given the hypotheses:

$$\neg(r\in rs)$$
$$\neg(nr \in rs\cup\{r\})$$

and the type matching which results from the context conditions.

For elements of *Rhsref*, the assumption about *MCloc* and an obvious property of *retrSTATE* are required.

As indicated above, the transformations shown, and thus the implied code, are "pessimistic". With small expressions the superfluous "LOAD" instructions become very wasteful. Consider the following example:

$$MC[x:=a+b](\ldots) \triangleq$$
$$(\underline{def}\ refx : MCloc[x](\ldots);$$
$$\underline{def}\ refa : MCloc[a](\ldots);$$
$$\underline{def}\ la : sr(refa, \underline{c}\ REGS);$$
$$\mathrm{REGS} := \underline{c}\ \mathrm{REGS} + [r \mapsto contents(la)]$$
$$\underline{def}\ refb : MCloc[b](\ldots);$$
$$\underline{def}\ lb : sr(refb, \underline{c}\ \mathrm{REGS});$$
$$\mathrm{REGS} := \underline{c}\ \mathrm{REGS} + [nr \mapsto contents(lb)];$$
$$\mathrm{REGS} := \underline{c}\ \mathrm{REGS} + [r \mapsto (\underline{c}\ \mathrm{REGS})(r)+(\underline{c}\ \mathrm{REGS})(nr)];$$
$$\underline{def}\ lx : sr(refx, \underline{c}\ \mathrm{REGS});$$
$$\mathrm{STR} := \underline{c}\ \mathrm{STR} + [l \mapsto (\underline{c}\ \mathrm{REGS})(r)]$$

obviously the sequence *corresponding to:*

RS(*LOAD*,nr,refb); RR(*ADD*,r,nr)

should be combined to use:

*RS(*ADD*,r,refb)*

This is an example of a class of optimization which can be defined in a natural way over the structure of a program. The necessary case distinction can be defined as follows:

```
MCtgi[mk-Infixexpr(e1,op,e2)](δ,r,rs) ≙
  MCtgi[e1](δ,r,rs);
  if e2∈Infixexpr then
   (let nr = newreg(rs∪{r}) in
    MCtgi[e2](δ,nr,rs∪{r});
    REGS   := c REGS + [r1 ↦ M[op]((c REGS)(r1),(c REGS)(r2)])])
  else
   (def (ref,rs') : MCloc[e2](dict,rs);
    def l  :  sr(ref,c REGS);
    REGS   := c REGS + [r ↦ M[op]((c REGS)(r),(c STR)(l))])
```

There are a number of examples of this class of optimization: [Jones 76a] shows a similar approach to handling comparison in conditional statements and the ubiquitous: $x := x + c$ type of assignment could be handled in the same way.

There are other classes of optimization which are better handled by an extra, equivalence preserving, pass. An example of this class is the allocation of actual registers to the "virtual" registers allocated by *MC*. The development of such translator structure is discussed in Chapter 9.

8.4 EXPRESSION EVALUATION - CODE SEQUENCES

The previous section redefined the semantics of (a part of) the language on a more baroque state. The correctness argument shows that this and the original language semantics correspond. It is now possible to proceed to the step invisaged in Fig. 2. In this simple example the transformations match obvious machine instructions. Thus:

```
Instr = RR | RS | SS
RR    :: (LOADREGISTER | Op)   Regno       Regno
RS    :: (LOAD | STORE | Op)   Regno       Storeref
SS    :: (MOVE | ...)          Storeref    Storeref
```

The semantics of the register-to-register and register-to-store instructions is given by:

```
MC[mk-RR(op,r1,r2)] △
  if op = LOADREGISTER then
          REGS := c REGS + [r1 ↦ (c REGS)(r2)]
  else if op ∈ Intop then
          REGS := c REGS + [r1 ↦ M[op]((c REGS)(r1),(c REGS)(r2))]
  else ...

MC[mk-RS(op,r,sref)] △
   def l : sr(sref,c REGS);
   if op = STORE then STR := c STR + [l ↦ (c REGS)(r)]
   else if op = LOAD then REGS := c REGS + [r ↦ contents(l)]
   else ...
```

It is a straightforward task to rewrite to *MC* semantics using these operations. (But the reader should bear in mind the warning given above that a careless choice of transformations may only be realizable by very inefficient sequences of instructions.) For example, the final version of *MCtg* becomes:

```
Ttgi[mk-Infixexpr(e1,op,e2)](δ,r,rs) △
    MCtgi[e1](δ,r,rs);
    if e2 ∈ Infixexpr then
      (let nr = newreg(rs∪{r}) in
       MCtgi[e2](δ,nr,rs∪{r});
       MC[mk-RR(op,r,nr)])
    else
      (def (ref,rs') : MCloc[e2](δ,rs);
       MC[mk-RS(op,r,ref)])
```

The correctness of this transition *Ttgi* can be checked by expansion to agree with *MCtgi*.

Having reached the stage of machine-like operations, it is now possible

to reinterpret the *T* functions: rather than mapping programs into functions, they can now be viewed as mapping into sequences of instructions. It should be observed that there are two sorts of case distinctions made in the *T* function. The static distinction like $e2 \in Infixexpr$ and dynamic distinction like the value of the Boolean expression in a conditional statement. The static distinctions depend on the text alone and govern a form of macro-expansion into the required instruction sequences. This view results in a specification of a mapping from (abstract) programs to sequences of instructions.

The total translator should, of course, not rely on programs being given in a convenient tree form. The task of relating the forms to something like "reverse Polish" is a simple case of data refinement. Suppose that the concrete syntax for the internal form of expression is:

Iexpr ::= *Iinfexpr* | *Ivarref*
Iinfixexpr ::= *Iexpr* *Iexpr* *Op*
Ivarref ::= ...

The relevant retrieve function:

$retrExpr : Iexpr \to Expr$

This can be defined by a standard stack algorithm. In order to sketch how this would work a function

$convert: (Ivarref \mid Op)^* \times Expr^* \to Expr$

is defined which must initially be called with a first argument which satisfies the concrete syntax of *Iexpr* and a second argument which is an empty list.

```
convert(tokenl,stack) Δ
  if hdtokenl∈Op then
      let e2 = hdstack    in
      let e1 = hdtlstack  in
      convert(tltokenl,<mk-Expr(e1,hdtokenl,e2)>^tltlstack)
  else
      convert(tltokenl,<hd tokenl>^stack)
```

If the actual infix expressions were to be taken as input, one of the

"two stack" algorithms could be used to express the retrieve function. In this case there would be more than one representation for each abstract tree. For an indication of how this method of documentation an interface works for a large language like PL/I, see [Weissenböck 75a].

8.5 LOCATION REFERENCES - LANGUAGE

This second example of relating design to the semantic definition concerns the computation of locations. It illustrates both data refinement and the resolution of non-determinacy in the semantics.

The part of the language of interest concerns the declaration of variables. The relevant abstract syntax definition from chapter 4 is:

Block :: *s-dclm:(Id* $\overrightarrow{m}$ *Type)* ...
Type :: *Scalartype* [*Expr*$^+$]
Scalartype = INT | BOOL

The relevant semantic objects are:

STORE = *SCALARLOC* $\overrightarrow{m}$ [*SCALARVALUE*]
ARRAYLOC = *Nat*$^+$ $\overrightarrow{m}$ *SCALARLOC*

where members of *ARRAYLOC* are one-to-one mappings whose domain forms a rectangle of natural numbers. The relevant semantic functions are:

```
M[mk-Block(dclm, ...)](ρ) ≜ ...
   def ρ' : ρ + ([id ↦ M[dclm(id)](ρ) | id∈domdclm] ∪ ... );
   always (let sclocs = ... in
           STR        := c STR \ sclocs; ...) in ...

M[mk-Type(sctp,bdl)](ρ) ≜
   if bdl=nil then (def l∈SCALARLOC - dom c STR;
                    STR := c STR ∪ [l ↦ nil];
                    return(l))
   else (def ebdl : ...;
         if ... then error
         else (let al∈ARRAYLOC s:t: domal = rect(ebdl) ∧
                                    is-disjl(<rngal,dom cSTR>);
                   STR := c STR ∪ [scl ↦ nil | scl∈rngal];
                   return(al)))
```

```
Mloc[mk-Varref(id,sscl)](ρ) Δ
    if sscl=nil then return (ρ(id))
    else (let aloc = ρ(id) in
          def esscl : <M[sscl[i]](ρ) | i∈indssscl>;
          if ¬(esscl∈domaloc) then error else return(aloc(esscl)))
```

Locations are constrained only in some respects by this *M* semantics. Care is taken in the semantics of *Type* to define a non-deterministic choice of *SCALARLOC*s. Although particular implementations will presumably provide a deterministic algorithm for selecting locations, the algorithm is not precisely determined by the semantics. (Strictly, this has lifted the whole definition from denotations which are functions to relations as denotations. The fact that the actual choice does not affect the outcome of a program could be formally proved.) The advantage of this freedom can now be felt. Here the chosen implementation reuses locations in a "stack" style; alternative implementations could always select new locations or or could perform dynamic garbage collection. Each such implementation could be shown to be a different specialization of the choice given in the *M* semantics. If, on the other hand, the semantics had prescribed a choice function, it would be necessary - for some implementations - to become involved in a cumbersome equivalence proof.

8.6 LOCATION REFERENCES - TARGET MACHINE

The standard von Neumann machine architecture does not support mappings as general as those used in "Store". The representation chosen here is of a byte string. Furthermore, not all *Scalarvalues* require the same amount of space. Here, integers are allocated four bytes (aligned on a four-byte boundary) and Boolean values one (whole!) byte. Arrays are mapped so that left-hand indices vary most rapidly. Array locations are represented by a base value and a list of multipliers. The bytes are used as a stack with a stack pointer (*PTR*) to indicate the next, unused, byte. Scalar locations are modelled by natural numbers which, in machine terms, are indices into the byte string.

By defining only constraints on the use of store (beyond the stack pointer), it is possible to have freedom for other information to be inserted in the stack. This freedom could be used to store a "display" and/or a "static chain" (cf.[Henhapl 70a]). Another implementation decision made here is to ignore the checking of subscript range. The semantics defines

it to be on *error* if the subscript is out of bounds -- this leaves an implementation free to do as it pleases. (A different implementation which avoided the danger of computing addresses outside on array could equally well be proved correct.) Thus:

$STATE_m$ :: STR: $Byte^*$ ...
PTR: Nat
$SCALARLOC$ = Nat
$ARRAYLOC_m$ :: s-base: $Nat0$
s-mults: Nat^+

There are obvious constraints which prohibit variables from overlapping. In a normal data refinement proof (cf. chapter 10) the next step would be to relate the representation to the abstraction by a retrieve function. In the current example, this is not possible because the representation lacks some of the information in the abstraction. In particular, the maximum bound information is not stored with the values. This information would have been required only to support the elided test. There are two possible ways of handling such a problem: [Jones 76a] followed [Lucas 68a] in using "ghost variables". Here, a relation between abstraction and representation is used rather than a retrieve function:

$Rstore \subseteq (SCALARLOC \xrightarrow{m} SCALARVALUE) \times Byte^+$
$Rstore(sm,bl) \triangleq sm = [l \mapsto value(l,type(l),bl) \mid l \in \underline{dom}\, sm]$

$type: SCALARLOC \to Scalartype$
$value: SCALARLOC \times Scalartype \times Byte^* \to SCALARVALUE$

$Raloc(arloc,mk\text{-}Arraylocm(b,ml)) \triangleq$
$\quad arloc = [sscl \mapsto arref(b,ml,sscl) \mid sscl \in \underline{dom}\, arloc]$

$arref(b,ml,sscl) \triangleq b + sum(1,\underline{len}\, ml,ml[i]+sscl[i])$

The semantics, expressed in terms of $STATE_m$ etc. is:

$MC[mk\text{-}Block(dclm,...)](\rho) \triangleq$
...
def $oldp$: *c* PTR;
def ρ' : $\rho + ([id \mapsto MC[dclm(id)](\rho) \mid id \in \underline{dom}\, dclm] \cup ...)$;
always (PTR := $oldp$; ...)
in (...)

```
MC[mk-Type(sctp,bdl)](ρ) Δ
   if bdl=nil then if sctp=BOOL
                   then (def l∈Nat s:t: l ≥ c PTR;
                         PTR := l+1;
                         return(l))
                   else (def l∈Nat s:t: l ≥ c PTR ∧ mod(l,4)=0;
                         PTR := l+4;
                         return(l))
   else (def ebdl : ...;
         if sctp = BOOL then (def l∈Nat s:t: l ≥ c PTR - 1;
                              let ml = mults(ebdl,1) in
                              PTR    := 1+arref(1,ebdl,ml)+1;
                              return mk-Arrayloc(l,ml))
         else ... )
```

Notice that the check on valid bounds has been omitted. Furthermore, there should strictly be a check on the exhaustion of store - this has not been shown here. The other functions are:

```
mults(ubdl,m) Δ if len ubdl=1 then <m>
                else (let tml=mults(tlubdl,m) in
                      <hdtlubdl * hdtml> ^ tml)
```

for:

```
MCloc[mk-Varref(id,sscl)](ρ)
```

in the scalar case, the address must be located and the array case the base address and multipliers used by *arref* to compute the address of the element. The addresses and multipliers would actually be located via a "display".

The *MC* semantics can be seen to conform to the constraints in the *M* semantics from the following observations:

- locations chosen beyond the stack pointer (PTR) are disjoint from existing locations

- consistency (non-overlap etc.) is preserved

- array locations correspond (under *Raloc*) to one:one mappings with rectangular domains.