IBM LABORATORY VIENNA, Austria

A COMPARISON OF TWO APPROACHES TO LANGUAGE DEFINITION AS BASES FOR THE CONSTRUCTION OF PROOFS

C.B.JONES

ABSTRACT:
Two approaches to language definition are considered: the functional approach developed by McCarthy in ref /4/ and the "Vienna Method". Definitions of two simple languages are used to compare proofs for the correctness of their compilers.

LN 25.3.050
3 February 1969

0. INTRODUCTION

The aim of this note is to consider the merits of language definitions, constructed in two different ways, as bases for the creation of proofs relating to the defined languages. In particular, the problem of proving the correctness of compilers is considered. The starting point is the language used by McCarthy in /1/. We re-define this in the style of /2/ and show two proofs based on the new definition.

The language is then extended so as to tax the methods of definition and proof construction with an important facet of procedural languages. Some observations are made in an attempt to summarise the limited results.

Familiarity with the Vienna Method (e.g. /2/) is assumed, as is an awareness of Recursion Induction (e.g. /3/).

REFERENCES

/1/ McCARTHY, J., PAINTER, J.: Correctness of a Compiler for Arithmetic Expressions.- Stanford University, A.I.-Memo No. 40, April 1966.

/2/ LUCAS, P., LAUER, P., STIGLEITNER, H.: Method and Notation for the Formal Definition of Programming Languages.- IBM Laboratory Vienna, Techn.Rep. TR 25.081, June 1968.

/3/ McCARTHY, J.: A Basis for a Mathematical Theory of Computation.- In: Computer Programming and Formal Systems, P.Braffort & D.Hirschberg (Eds.).

/4/ McCARTHY, J.: A Formal Description of a Subset of ALGOL.- In: Formal Language Description Languages, T.Steel (Ed.) Vienna Conference, September 1964.

1. APPLICATION OF THE VIENNA METHOD TO McCARTHY'S PROOF

The use of some notational aspects of the "VM" has permitted a slight shortening of this proof (the reasons for this are examined at the end of the section). However, the language defined is very simple and section 3 develops a more interesting example.

Transliteration of Proof:

In order to facilitate comparison with the original, its sequence of presentation is followed; transliterated formulae are given their original number preceded by M. Other than the changes which can be attributed to the "VM", conditional expression form has been used to aid readability.

M2 The Source Language

The abstract syntax of source expressions is:

is-expr = is-const ∨ is-var ∨ is-sum
is-sum = (<s-1:is-expr>,<s-2:is-expr>)
is-var, is-const } not defined

The semantics of the source language is given by the function value whose second argument is a "state vector" which is defined by:

is-l-state(ξ) = ({<var:is-value> || is-var(var)})

The function is defined:

M2.1 value(e,ξ) = (is-const(e) ⟶ val(e),
is-var(e) ⟶ e(ξ),
is-sum(e) ⟶ value(s-1(e),ξ) + value(s-2(e),ξ))

for: is-expr(e) and is-l-state(ξ)
where: the function val gives the numerical value of a constant, and is not further defined.

Note: The function c of /1/ is exactly replaced by the application of the variable name as a selector to the state.

We shall use the induction principle of /1/: Suppose Φ is a predicate applicable to expressions, and suppose that for all expressions e we have:

$$\text{is-const}(e) \supset \Phi(e)$$
$$\text{is-var}(e) \supset \Phi(e)$$
$$\text{is-sum}(e) \wedge \Phi(\text{s-1}(e)) \wedge \Phi(\text{s-2}(e)) \supset \Phi(e)$$

then we may conclude that $\Phi(e)$ is true for all expressions e.

M3 The Object Language

The abstract syntax of the object language is:

```
is-program = is-instr-list
is-instr = is-li v is-load v is-sto v is-add
is-li = (<s-t:LI>,<s-arg:is-const>)
is-load = (<s-t:LOAD>,<s-adr:is-adr>)
is-sto = (<s-t:STO>,<s-adr:is-adr>)
is-add = (<s-t:ADD>,<s-adr:is-adr>)
```

where: is-adr is true for any valid selector of is-state.
We shall use the constructors of /1/ as abbreviations, thus:
mk-load(x) = μ(<s-t:LOAD>,<s-adr:x>)
The meaning of the object language is given by the step and outcome functions whose second argument is a "state vector" defined by:

```
is-state = (<s-ac:is-value>,<s-v-pt:is-v-pt>,
            <s-t-pt:is-value-list>)
is-v-pt = ({<var:is-value> || is-var(var)})
```

Note: The functions a and c of /1/, applied to machine states, are replaced by μ functions and selectors using appropriate composite selectors.

We shall abbreviate s-elem(i)∘s-t-pt(ξ) to i∘s-t-pt(ξ) throughout the sequel.

The semantics of the object language are given by:

M3.11 step(s,η) = (is-li(s) ⟶ μ(η;<s-ac:s-arg(s)>),
is-load(s) ⟶ μ(η;<s-ac : s-adr(s)(η)>),
is-sto(s) ⟶ μ(η;<s-adr(s):s-ac(η)>),
is-add(s) ⟶ μ(η;<s-ac:s-adr(s)(η) + s-ac(η)>))

M3.12 outcome(p,η) = (is-<>(p) ⟶ η,
T ⟶ outcome(tail(p),step(head(p),η)))

for: is-program(p) and is-state(η)

Ref. /1/ states the following lemma without proof:

M3.13 outcome(p1⌢p2,η) = outcome(p2,outcome(p1,η))

M4 The Compiler

Note: The mapping function of /1/, "loc", can be thought of as mapping a variable name, say v, onto the composite selector for this value in is-state, i.e. loc(v,map) = v∘s-v-pt.

Our statement of the relationship of the initial object machine and language "state vectors", called η and ξ respectively, becomes:

M4.1 s-v-pt(η) = ξ

The compiler is defined by:

M4.2 compile(e,t) =
(is-const(e) ⟶ <mk-li(val(e))>,
is-var(e) ⟶ <mk-load(e∘s-v-pt)>,
is-sum(e) ⟶ compile(s-1(e),t)⌢<mk-sto(t)>⌢
compile(s-2(e),t+1)⌢<mk-add(t)>)

for: is-expr(e) and is-integer(t)

Partial equality of objects (i.e. equality for a limited set of selectors) is shown by:

$$\xi_1 = \{\alpha_1,\alpha_2,\ldots,\alpha_n\}\ \xi_2 = (\alpha_1(\xi_1) = \alpha_1(\xi_2) \wedge \alpha_2(\xi_1) = \alpha_2(\xi_2) \wedge \vdots \alpha_n(\xi_1) = \alpha_n(\xi_2))$$

Partial equality satisfies the following relations:

M4.3 $\xi_1 = \xi_2$ is equivalent to $\xi_1 = \{s \mid (s(\xi_1) \neq \Omega) \vee (s(\xi_2) \neq \Omega)\}\ \xi_2$

M4.4 if $A \subset B$ and $\xi_1 =_B \xi_2$ then $\xi_1 =_A \xi_2$

M4.5 if $\xi_1 =_A \xi_2$ then $\mu(\xi_1;\langle x:\alpha\rangle) =_{A\cup\{x\}} \mu(\xi_2;\langle x:\alpha\rangle)$

M4.6 if $x \notin A$ then $\mu(\xi;\langle x:\alpha\rangle) =_A \xi$

M4.7 if $\xi_1 =_A \xi_2$ and $\xi_2 =_B \xi_3$ then $\xi_1 =_{A\cap B} \xi_3$

The correctness of the compiler is stated in:

M4.8 Theorem 1

If η and ξ are states and language states respectively, such that M4.1 holds, then

outcome (compile(e,t),η) = $\{$s-ac,s-v-pt$\}\ \mu(\eta;\langle$s-ac:value(e,ξ)$\rangle$)

for: is-expr(e) and is-integer(t).

M5 Proof of Theorem 1

For reference we state, without proof, an axiom of μ functions:

1.1 s-1($\mu(\xi;\langle$s-2:$\alpha\rangle)$) = (s-1 = s-2 $\longrightarrow \alpha$, T $\longrightarrow$ s-1(ξ))

Case I: is-const(e)

outcome(compile(e,t),η) = outcome($\langle$mk-li(val(e))$\rangle$,η) M4.2
= step(mk-li(val(e)),η) M3.12
= $\mu(\eta;\langle$s-ac:s-arg(mk-li(val(e)))$\rangle$) M3.11
= $\mu(\eta;\langle$s-ac:val(e)$\rangle$) 1.1
= $\mu(\eta;\langle$s-ac:value(e,ξ)$\rangle$) M2.1
= $\{$s-ac,s-v-pt$\}\mu(\eta;\langle$s-ac:value(e,ξ)$\rangle$) M4.3,M4.4

Case II: is-var(e)

outcome(compile(e,t),η)

= outcome(<mk-load(e∘s-v-pt)>,η) M4.2

= step(mk-load(e∘s-v-pt),η) M3.12

= $\mu(\eta;$<s-ac:s-adr(mk-load(e∘s-v-pt))(η)>) M3.11

= $\mu(\eta;$<s-ac:e∘s-v-pt(η)>) 1.1

= $\mu(\eta;$<s-ac:e(ξ)>) M4.1

= $\mu(\eta;$<s-ac:value(e,ξ)>) M2.1

= {s-ac,s-v-pt}$\mu(\eta;$<s-ac:value(e,ξ)>) M4.3,M4.4

Case III:is-sum(e)

outcome(compile(e,t),η)

= outcome(compile(s-1(e),t)⌢<mk-sto(t)>⌢compile(s-2(e),t+1)⌢
<mk-add(t)>,η) M4.2

= outcome(<mk-add(t)>,outcome(compile(s-2(e),t+1),
outcome(<mk-sto(t)>, outcome(compile(s-1(e),t),η))))
M3.13

Now we introduce some notation:

let v = value(e,ξ)
v_1 = value(s-1(e),ξ)
v_2 = value(s-2(e),ξ)
then v = v1 + v2 M2.1

let ξ_1 = outcome(compile(s-1(e),t),η)
ξ_2 = outcome(<mk-sto(t)>, ξ_1)
ξ_3 = outcome(compile(s-2(e),t+1),ξ_2)
ξ_4 = outcome(<mk-add(t),ξ_3>)

so that ξ_4 = outcome(compile(e,t),η), we wish to prove

ξ_4 = {s-ac,s-v-pt}$\mu(\eta;$<s-ac:v>)

We have

ξ_1 = outcome(compile(s-1(e),t),η)
= {s-ac,s-v-pt}$\mu(\eta;$<s-ac:v1>) I.H.

Now ξ_2 = outcome(<mk-sto(t)>,ξ_1)
= $\mu(\xi_1;$<t∘s-t-pt:s-ac(ξ_1)>) M3.12,M3.11

Next ξ_3 = outcome(compile(s-2(e),t+1),ξ_2)
= {s-ac,s-v-pt,t∘s-t-pt}$\mu(\xi_2;$<s-ac:v2>) I.H.[1]

Finally
ξ_4 = outcome(<mk-add(t)>,ξ_3)
= $\mu(\xi_3;$<s-ac:t∘s-t-pt(ξ_3) + s-ac(ξ_3)>) M3.12,M3.11
but t∘s-t-pt(ξ_3) = t∘s-t-pt(ξ_2) = s-ac(ξ_1) = v1 1.1
and s-ac(ξ_3) = v2 1.1

∴ ξ_4 = $\mu(\xi_3;$<s-ac:v>)
now ξ_3 = {s-v-pt}ξ_2 = {s-v-pt}ξ_1 = {s-v-pt}η 1.1

∴ ξ_4 = {s-ac,s-v-pt}$\mu(\eta;$<s-ac:v>) M4.5,M4.1

This concludes the proof

Comment on differences:

We can now tabulate the differences, between the above and the original, alongside the changes in the definition which brought them about:

a) The abstract syntax is defined by a single set of predicates. This eliminates the relationships between synthetic and analytic syntaxes listed in M3.1 - M3.4 (these are now properties of objects and selectors).

b) A predefined list notation is used. This eliminates M3.5 - M3.7.

c) The state vectors are defined as objects. This immediately yields M3.8 - M3.10 as properties of the μ function. It also allows some steps of case III of the proof to be omitted, without loss of clarity, because of the ease of tracing back through μ operations using 1.1.

---

[1] The Induction Hypothesis can be applied because ξ_2 = {s-v-pt}η

2. PROOF OF THEOREM 1 BY RECURSION INDUCTION

The formulation of the problem in the above notation prompts a proof by recursion induction (see /3/). This section can be thought of as an alternative M5, it would also remove the need for the Induction Principle of M2 and the notion of partial equality of M4.

Theorem 1 Given η and ξ are machine and language state vectors respectively such that:

2.1 s-v-pt(η) = ξ

show: outcome(compile(e,t),η) = {s-ac,s-v-pt}$\mu(\eta;$<s-ac:value(e,ξ)>)

Part 1: with respect to s-v-pt

s-v-pt(step(s,η)) ≠ s-v-pt(η) ⊃ s=mk-sto(v∘s-v-pt) |
is-var(v) M3.11

but

~~∀~~e.elem(i)° compile (e,t) ≠ mk-sto(v∘s-v-pt) |
is-var(v) ∧ is-integer(i) M4.2

therefore

s-v-pt(outcome(compile(e,t),η)) = s-v-pt(η)

Now: s-v-pt($\mu(\eta;$<s-ac:value(e,ξ)>)) = s-v-pt(η) 1.1

Therefore they are equal, which proves part I.

Lemma Before tackling part 2 we prove a lemma which shows that s-ac of the state after evaluating the second part of an expression, would have been the same if the second subexpression had been evaluated first.

s-v-pt(η_1) = s-v-pt(η_2) ⊃
s-ac(outcome(compile(e,t),η_1)) = M3.11,M3.12
s-ac(outcome(compile(e,t),η_2)) M4.2

since

s-v-pt(outcome(compile(e,t)^mk-sto(t),η)) =
s-v-pt(η) ditto

then

2.2 η_2 = outcome(compile(e,t)⌢mk-sto(t),η) ⊃
s-ac(outcome(compile(e_2,t),η_2)) = s-ac(outcome(compile(e_2,t),η_1)

Part 2 : with respect to s-ac, the theorem becomes

s-ac(outcome(compile(e,t),η)) = value(e,ξ)

We consider both of these as functions of e and η over the domain is-expr(e), is-state(η) and is-integer(t). We use the equation:

f(e,η,t) = (is-const(e) ⟶ val(e),
is-var(e) ⟶ e∘s-v-pt(η),
is-sum(e) ⟶ f(s-1(e),η,t) + f(s-2(e),η,t+1))

This function must terminate because there can be no infinite chain of non identity selectors (i.e. the result of a finite number of applications of s-1 is an object of type is-const or is-var).

Now value(e,ξ) M2.1
= (is-const(e) ⟶ val(e),
is-var(e) ⟶ e(ξ),
is-sum(e) ⟶ value(s-1(e),ξ) + value(s-2(e),ξ))

= (is-const(e) ⟶ val(e),
is-var(e) ⟶ e∘s-v-pt(η), 2.1
is-sum(e) ⟶ value(s-1(e),ξ) + value(s-2(e),ξ))

Clearly [1] this satisfies f.

---

1) f'(e,η,t) = value(e,ξ) where η and ξ are related by 2.1
= (is-const(e) ⟶ val(e),
is-var(e) ⟶ e∘s-v-pt(η),
is-sum(e) ⟶ f'(s-1(e),η,t) + f'(s-2(e),η,t+1))

From M4.2 and the distributive law for condition expressions:

$$\text{s-ac}(\text{outcome}(\text{compile}(e,t),\eta)) =$$
$$\text{s-ac}(\text{is-const}(e) \longrightarrow \text{outcome}(\langle\text{mk-li}(\text{val}(e))\rangle,\eta)$$
$$\text{is-var}(e) \longrightarrow \text{outcome}(\langle\text{mk-load}(e\circ\text{s-v-pt})\rangle,\eta),$$
$$\text{is-sum}(e) \longrightarrow \text{outcome}(\langle\text{mk-add}(t)\rangle,\xi_3)) \qquad \text{M3.13}$$

where $\xi_3 = \text{outcome}(\text{compile}(\text{s-2}(e),t+1),\xi_2)$
$\xi_2 = \text{outcome}(\langle\text{mk-sto}(t)\rangle,\xi_1)$
$\xi_1 = \text{outcome}(\text{compile}(\text{s-1}(e),t),\eta)$

by M3.12 and M3.11 we get:

$$\text{s-ac}(\text{is-const}(e) \longrightarrow \mu(\eta;\langle\text{s-ac:val}(e)\rangle),$$
$$\text{is-var}(e) \longrightarrow \mu(\eta;\langle\text{s-ac:e}\circ\text{s-v-pt}(\eta)\rangle),$$
$$\text{is-sum}(e) \longrightarrow \mu(\xi_3;\langle\text{s-ac:t}\circ\text{s-t-pt}(\xi_3) + \text{s-ac}(\xi_3)\rangle))$$

Now since $t\circ\text{s-t-pt}(\text{outcome}(\text{compile}(\text{s-2}(e),\ t+1),\xi_2)) =$
$t\circ\text{s-t-pt}(\xi_2)$ M4.2

$t\circ\text{s-t-pt}(\xi_3) = t\circ\text{s-t-pt}(\xi_2)$
$= t\circ\text{s-t-pt}(\text{outcome}(\langle\text{mk-sto}(t)\rangle,\xi_1))$
$= \text{s-ac}(\xi_1)$ M3.12,M3.11

and $\text{s-ac}(\xi_3) = \text{s-ac}(\text{outcome}(\text{compile}(\text{s-2}(e),t+1),\xi_2))$
$= \text{s-ac}(\text{outcome}(\text{compile}(\text{s-2}(e),t+1),\eta))$ 2.2

we get, by the distributive law:

$$(\text{is-const}(e) \longrightarrow \text{val}(e),$$
$$\text{is-var}(e) \longrightarrow e\circ\text{s-v-pt}(\eta),$$
$$\text{is-sum}(e) \longrightarrow \text{s-ac}(\text{outcome}(\text{compile}(\text{s-1}(e),t),\eta)) +$$
$$\text{s-ac}(\text{outcome}(\text{compile}(\text{s-2}(e),t+1),\eta)))$$

Clearly this also satisfies f, which concludes the proof.

3. AN EXTENDED LANGUAGE

The language defined in Section 1 was such that it was defined, both in the original and with use of objects and selectors, in a purely functional way. A central part of the Vienna Method is its use of a machine and we now extend the language we are defining to illustrate this basic difference between a purely functional notation and the Vienna Method. These definitions will be used as bases for construction of proofs in the next section.

The language of /1/ did not permit any "side-effects" to occur. (Thus the order of evaluation of sub-expressions can safely be left undefined in M2.1). We now define a language of expressions which include embedded assignment. We assume the language fixes the order of evaluation [1]. The abstract syntax of the language is:

```
is-expr = is-const v is-var v is-assign v is-sum
is-assign = (<s-l:is-var>,<s-r:is-expr>)
is-sum = (<s-1:is-expr>,<s-2:is-expr>)
```

We first define the semantics using the functional approach. It appears to be necessary [2] to show the state dependancies explicitly, either by functions returning states as well as values or by defining two different functions as shown here. The language is defined by two functions of an expression and a state: value returns a single number result and eval returns a, possibly modified, state.

---

[1] Thus the Vienna Method definition will use only a control stack, not a tree.

[2] The definition of /4/ includes the possibility of states altering during execution and is handled by passing the new state to a recursive call of the semantic function. The other argument contains a statement number which is used as the value of a bound variable to select the appropriate successor instruction. However, this technique does not appear to be applicable to recursive evaluation (e.g. that used for expressions).

3.1 value(e,ξ) = (is-const(e) ⟶ val(e),
is-var(e) ⟶ e(ξ),
is-assign(e) ⟶ value(s-r(e),ξ),
is-sum(e) ⟶ value(s-1(e),ξ) +
value(s-2(e),eval(s-1(e),ξ)))

3.2 eval(e,ξ) = (is-const(e) ⟶ ξ,
is-var(e) ⟶ ξ,
is-assign(e) ⟶ μ(eval(s-r(e),ξ);
<s-1(e):value(s-r(e),ξ)>),
is-sum(e) ⟶ eval(s-2(e), eval(s-1(e),ξ)))

for: is-expr(e) and is-l-state(ξ)

We have used μ functions and selectors, rather than the a & c functions of McCarthy, for ease in Section 4.

The "V.M." notation allows the more direct definition:

3.3 value(e) =

```
is-const(e) ⟶
   PASS:val(e)
is-var(e) ⟶
   PASS:e(ξ)
is-assign(e) ⟶ assign(s-1(e),v);
                 v:value(s-r(e))
is-sum(e) ⟶ add(v1,v2);
              v2:value(s-2(e));
               v1:value(s-1(e))
```

3.4 assign(var,val) =

```
PASS:val
  ξ :μ(ξ;<var:val>)
```

3.5 add(v1,v2) =

```
PASS:v1 + v2
```

4. PROOFS BASED ON EXTENDED LANGUAGE

In this section we define a new compiler, whose domain is is-expr as defined in Section 3, which creates programs for the machine of M3. We then use the definitions of the last section to prove the correctness of the compiler. Theorem 2 uses recursion induction to prove the compiler correct for 3.1 and 3.2. Theorem 3 proves the compiler correct from 3.3 - 3.5 by induction. The ULD definition is not directly suited to a RI proof because of the hidden state argument. Clearly, in this case, the function definition can easily be formed from the "V.M." definition.

The compiler is defined by:

4.1 compile(e,t) =

$$
\begin{aligned}
&(\text{is-const}(e) \longrightarrow \langle \text{mk-li}(\text{val}(e))\rangle,\\
&\ \text{is-var}(e) \longrightarrow \langle \text{mk-load}(e\circ \text{s-v-pt})\rangle,\\
&\ \text{is-assign}(e) \longrightarrow \text{compile}(\text{s-r}(e),t)^{\frown}\\
&\qquad\qquad \langle \text{mk-sto}(\text{s-l}(e)\circ \text{s-v-pt})\rangle,\\
&\ \text{is-sum}(e) \longrightarrow \text{compile}(\text{s-1}(e),t)^{\frown}\langle \text{mk-sto}(t\circ \text{s-t-pt})\rangle^{\frown}\\
&\qquad\qquad \text{compile}(\text{s-2}(e),t+1)^{\frown}\langle \text{mk-add}(t\circ \text{s-t-pt})\rangle)
\end{aligned}
$$

Theorem 2:

Given:

4.2 $\text{s-v-pt}(\eta) = \xi$

Show:

$\text{s-ac}(\text{outcome}(\text{compile}(e,t),\eta)) = \text{value}(e,\xi)$

and $\text{s-v-pt}(\text{outcome}(\text{compile}(e,t),\eta)) = \text{eval}(e,\xi)$

We consider all four of these as functions of e and η over the ranges is-expr(e) and is-state(η). We shall show the pairs satisfy:

$$
\begin{aligned}
f(e,\eta) = (&\text{is-const}(e) \longrightarrow \text{val}(e),\\
&\text{is-var}(e) \longrightarrow e\circ \text{s-v-pt}(\eta),\\
&\text{is-assign}(e) \longrightarrow f(\text{s-r}(e),\eta),\\
&\text{is-sum}(e) \longrightarrow f(\text{s-1}(e),\eta) + f(\text{s-2}(e),g(\text{s-1}(e),\eta)))
\end{aligned}
$$

$$g(e,\eta) = (\text{is-const}(e) \longrightarrow \text{s-v-pt}(\eta),$$
$$\text{is-var}(e) \longrightarrow \text{s-v-pt}(\eta),$$
$$\text{is-assign}(e) \longrightarrow \mu(g(\text{s-r}(e),\eta);$$
$$\langle \text{s-l}(e) : f(\text{s-r}(e),\eta)\rangle),$$
$$\text{is-sum}(e) \longrightarrow g(\text{s-2}(e),g(\text{s-1}(e),\eta)))$$

Functions f and g must terminate because the chains of selectors must be finite.

Using 4.2, 3.1 and 3.2 satisfy f and g directly.

By the distributive law and 4.1 we can write

4.3 $\text{outcome}(\text{compile}(e,t),\eta) =$
$(\text{is-const}(e) \longrightarrow \text{outcome}(\langle \text{mk-li}(\text{val}(e))\rangle,\eta),$
$\text{is-var}(e) \longrightarrow \text{outcome}(\langle \text{mk-load}(e \circ \text{s-v-pt})\rangle,\eta),$
$\text{is-assign}(e) \longrightarrow \text{outcome}(\langle \text{mk-sto}(\text{s-l}(e) \circ \text{s-v-pt})\rangle,\xi_4),$
$\text{is-sum}(e) \longrightarrow \text{outcome}(\langle \text{mk-add}(t \circ \text{s-t-pt})\rangle,\xi_3))$

where

$$\left.\begin{aligned}
\xi_4 &= \text{outcome}(\text{compile}(\text{s-r}(e),t),\eta)\\
\xi_3 &= \text{outcome}(\text{compile}(\text{s-2}(e),t+1),\xi_2)\\
\xi_2 &= \text{outcome}(\langle \text{mk-sto}(t \circ \text{s-t-pt})\rangle,\xi_1)\\
\xi_1 &= \text{outcome}(\text{compile}(\text{s-1}(e),t),\eta)
\end{aligned}\right\} \quad \text{M3.13}$$

Now by M3.12 and M3.11 we can write:

$\text{s-ac}(\text{outcome}(\text{compile}(e,t),\eta)) =$
$(\text{is-const}(e) \longrightarrow \text{s-ac}(\mu(\eta;\langle \text{s-ac}:\text{val}(e)\rangle)),$
$\text{is-var}(e) \longrightarrow \text{s-ac}(\mu(\eta;\langle \text{s-ac}:e \circ \text{s-v-pt}(\eta)\rangle)),$
$\text{is-assign}(e) \longrightarrow \text{s-ac}(\mu(\xi_4;\langle \text{s-l}(e) \circ \text{s-v-pt}:\text{s-ac}(\xi_4)\rangle)),$
$\text{is-sum}(e) \longrightarrow \text{s-ac}(\mu(\xi_3;\langle \text{s-ac}:t \circ \text{s-t-pt}(\xi_3) + \text{s-ac}(\xi_3)\rangle)))$

Now since t is not changed by outcome(compile(s-2(e),t+1),ξ_2):

M4.2

$$t\circ s\text{-}t\text{-}pt(\xi_3) = t\circ s\text{-}t\text{-}pt(\xi_2)$$
$$= t\circ s\text{-}t\text{-}pt(outcome(\langle mk\text{-}sto\ (t\circ s\text{-}t\text{-}pt)\rangle,\xi_1))$$
$$= s\text{-}ac(\xi_1) \qquad M3.12, M3.11$$

and outcome(compile(s-2(e),t+1),ξ_2) relies only on s-v-pt(ξ_2)

$$s\text{-}ac(\xi_3) = s\text{-}ac(outcome(compile(s\text{-}2(e),t+1),s\text{-}v\text{-}pt(\xi_2)))$$
$$= s\text{-}ac(outcome(compile(s\text{-}2(e),t+1),s\text{-}v\text{-}pt(\xi_1))) \quad M3.11$$

Therefore we get, by 1.1

$$\begin{aligned}
&(is\text{-}const(e) \longrightarrow val(e),\\
&\ is\text{-}var(e) \longrightarrow e\circ s\text{-}v\text{-}pt(\eta),\\
&\ is\text{-}assign(e) \longrightarrow s\text{-}ac(outcome(compile(s\text{-}r(e),t),\eta)),\\
&\ is\text{-}sum(e) \longrightarrow s\text{-}ac(outcome(compile(s\text{-}1(e),t),\eta)) +\\
&\qquad\qquad s\text{-}ac(outcome(compile(s\text{-}2(e),t+1),\\
&\qquad\qquad\qquad s\text{-}v\text{-}pt(outcome(compile(s\text{-}1(e),t),\eta)))
\end{aligned}$$

which clearly satisfies f.

Reverting to 4.3, we use M3.12 & M3.11 to write:

$$\begin{aligned}
&s\text{-}v\text{-}pt(outcome(compile(e,t),\eta)) =\\
&\quad (is\text{-}const(e) \longrightarrow s\text{-}v\text{-}pt(\mu(\eta;\langle s\text{-}ac:val(e)\rangle)),\\
&\quad\ is\text{-}var(e) \longrightarrow s\text{-}v\text{-}pt(\mu(\eta;\langle s\text{-}ac:e\circ s\text{-}v\text{-}pt(\eta)\rangle)),\\
&\quad\ is\text{-}assign(e) \longrightarrow s\text{-}v\text{-}pt(\mu(\xi_4;\langle s\text{-}1(e)\circ s\text{-}v\text{-}pt:s\text{-}ac(\xi_4)\rangle)),\\
&\quad\ is\text{-}sum(e) \longrightarrow s\text{-}v\text{-}pt(\mu(\xi_3;\langle s\text{-}ac:t\circ s\text{-}t\text{-}pt(\xi_3) +\\
&\qquad\qquad\qquad s\text{-}ac(\xi_3)\rangle)))
\end{aligned}$$

Now $s\text{-}v\text{-}pt(\xi_3) = s\text{-}v\text{-}pt(outcome(compile(s\text{-}2(e),t+1),\xi_2))$
$= s\text{-}v\text{-}pt(outcome(compile(s\text{-}2(e),t+1),\xi_1))$ M3.11

Therefore using 1.1, we get:

(is-const(e) ⟶ s-v-pt(η),
is-var(e) ⟶ s-v-pt(η),
is-assign(e) ⟶ μ(s-v-pt(outcome(compile(s-r(e),t),η));
<s-l(e)∘s-v-pt:s-ac(outcome(compile
(s-r(e),t),η))>),
is-sum(e) ⟶ s-v-pt(outcome(compile(s-2(e),t+1),
s-v-pt(outcome(compile(s-1(e),t),η))))

This satisfies g, which concludes the proof of Theorem 2.

Theorem 3

Notation: In order to talk about the results of the execution of Instructions we use the notation R[instr(args),ξ] to represent the result passed by instr, when executed in state ξ with arguments args; and S[instr(args,ξ)] to represent the value of the state after this execution.

4.4 Given s-v-pt(η) = ξ show:

s-ac(outcome(compile(e,t),η)) = R[value(e),ξ]

and s-v-pt(outcome(compile(e,t),η)) = S[value(e),ξ]

Case I: is-const(e)

outcome(compile(e,t),η) = outcome(<mk-li(val(e))>,η) 4.1
= μ(η;<s-ac:val(e)>) M3.12,M3.11

Therefore

s-ac(outcome(compile(e,t),η)) = val(e) 1.1
but R[value(e),ξ] = val(e) 3.3

and s-v-pt(outcome(compile(e,t),η)) = s-v-pt(η) 1.1
but S[value(e),ξ] = s-v-pt(η) 4.4

This proves case I.

Case II: is-var(e)

outcome(compile(e,t),η)) = outcome(<mk-load(e∘s-v-pt)>,η) 4.1
= $\mu(\eta;$<s-ac:e∘s-v-pt(η)>) M3.12,M3.11

Therefore

s-ac(outcome(compile(e,t),η)) = e∘s-v-pt(η) 1.1

but R[value(e),ξ] = e(ξ) 3.3
= e∘s-v-pt(η) 4.4

and s-v-pt(outcome(compile(e,t),η)) = s-v-pt(η) 1.1

but S[value(e),ξ] = ξ 3.3
= s-v-pt(η) 4.4

This proves case II.

Case III: is-assign(e)

outcome(compile(e,t),η) = outcome(<mk-sto(s-l(e)∘s-v-pt)>,ξ_4) 4.1

where: ξ_4 = outcome(compile(s-r(e),t),η) M3.13

Thus $\mu(\xi_4;$<s-l(e)∘s-v-pt:s-ac(ξ_4)>) M3.12,M3.11

Now s-ac(outcome(compile(s-r(e),t),η)) = R[value(s-r(e)),ξ] } IND
which we shall call vr
s-v-pt(outcome(compile(s-r(e),t),η)) = S[value(s-r(e)),ξ] } HYP
which we shall call ξ_r

So s-ac(outcome(compile(e,t),η)) = vr 1.1

but R[value(e),ξ] = R[assign(s-l(e),vr),ξ_r] 3.3
= vr 3.4

And s-v-pt(outcome(compile(e,t),η)) = μ(s-v-pt(ξ_4);<s-l(e):s-ac(ξ_4)>)
= $\mu(\xi_r;$<s-l(e):vr>)

but S[value(e),ξ] = S[assign(s-l(e),vr),ξ_r] 3.3
= $\mu(\xi_r;$<s-l(e):vr>) 3.4

This proves case III.

Case IV: is-sum(e)

outcome(compile(e,t),η) = outcome(<mk-add(t∘s-t-pt)>,ξ_3) 4.1

where ξ_3 = outcome(compile(s-2(e),t+1),ξ_2) M3.13

ξ_2 = outcome(<mk-sto(t∘s-t-pt)>,ξ_1)

ξ_1 = outcome(compile(s-1(e),t),η)

Thus $\mu(\xi_3;$<s-ac:t∘s-t-pt(ξ_3) + s-ac(ξ_3)>) M3.12,M3.11

Now s-ac(outcome(compile(s-1(e),t),η)) = R[value(s-1(e)),ξ] } IND

which we shall call vs1

s-v-pt(outcome(compile(s-1(e),t),η)) = S[value(s-1(e)),ξ] } HYD

which we shall call ξ_{s1}

$\xi_2 = \mu(\xi_1;$<t∘s-t-pt:s-ac(ξ_1)>) M3.12,M3.11

Therefore

4.5 s-v-pt(outcome(<mk-sto(t∘s-t-pt)>,ξ_1)) = ξ_{s1} 1.1

and t∘s-t-pt(outcome(<mk-sto(t∘s-t-pt)>,ξ_1)) = vs1 1.1

Now, since s-v-pt(ξ_2) = ξ_{s1}

s-ac(outcome(compile(s-2(e),t+1),ξ_2)) = R[value(s-2(e)),ξ_{s1}] } 4.5 IND

which we shall call vs 2

s-v-pt(outcome(compile(s-2(e),t+1),ξ_2)) = S[value(s-2(e)),ξ_{s1}] } HYD

which we shall call ξ_{s2}

also t∘s-t-pt(outcome(compile(s-2(e),t+1),ξ_2)) = vs1

So s-ac(outcome(compile(e,t),η)) = t∘s-t-pt(ξ_3) + s-ac(ξ_3) 1.1

= vs1 + vs2

R[value(e),ξ] = R[add(vs1,vs2),ξ_{s2}] 3.2

= vs1 + vs2 3.5

s-v-pt(outcome(compile(e,t),ξ)) = ξ_{s2}

S[value(e),ξ] = S[add(v1,v2), ξ_{s2}] 3.3

= ξ_{s2} 3.5

This proves case IV, which concludes the proof of Theorem 3.

5. SUMMARY

In addition to showing the convenience of the "V.M." for defining languages, this note has considered its use as a basis for proof work. The single abstract syntax and use of formal objects in a definition provide a base whose properties are already known, and thus shorten proofs of this type.

The underlying concept of Vienna Method definitions is that of a machine. Although such definitions are well suited to defining algorithmic languages, proofs using them as a basis are less direct since the initial steps must be deductions about the effect of instructions. In particular, the useful Recursion Induction technique, which relies on manipulation of (conditional) expressions, is no longer directly applicable.

could however, do induction on entire
states - including $\subseteq$ which points to
what still has to be done