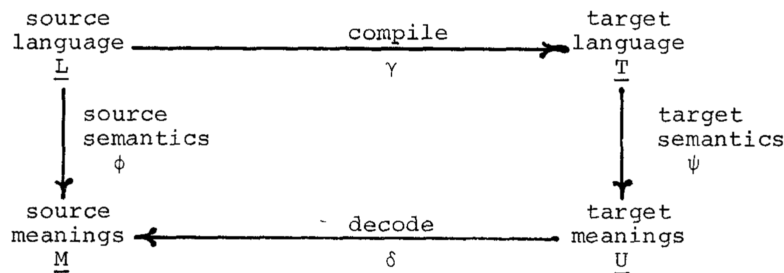
Advice on Structuring Compilers and Proving Them Correct †

F. Lockwood Morris

Syracuse University

The purpose of this paper is to advise an approach (and to support that advice by discussion of an example) towards achieving a goal first announced by John McCarthy: that compilers for higher-level programming languages should be made completely trustworthy by proving their correctness. The author believes that the compiler-correctness problem can be made much less general and better-structured than the unrestricted program-correctness problem; to do so will of course entail restricting what a compiler may be.

The essence of the present advice is that a proof of compiler correctness should be the proof that a diagram of the form



commutes. (The symbols in the diagram refer to the example to be given below.) It is of course not very startling to regard a compiler as a function assigning target language programs to source language programs; however, the rest of the diagram does have a non-vacuous content: it insists that 'semantics' should be an actual function assigning definite mathematical objects ('meanings') to programs--usually partial functions of some fairly complicated type. That is, it rejects interpretive forms of semantic definition, such as those provided by the Vienna definition method, or by sets of reduction rules of some form, which only provide for any given pair of program and input data some prescription for finding the output (if any). The bottom, 'decoding' side of the diagram is merely an allowance for the highly probable eventuality that the meanings of target language programs are not identical with, but only represent in some adequate way the meanings of the source language programs from which they were compiled.

There is one crucial elaboration to what has just been said: it is further advised that the corners of the diagram should be made not merely sets but algebras and that the arrows should be homomorphisms. Specifically, <u>heterogeneous</u> (universal) <u>algebras</u>, as defined in (1), seem to be the appropriate tool. Briefly, a heterogeneous algebra has a number of sets of elements of different types, its <u>phyla</u>, which are connected by a family of operations, each taking a finite number of arguments of specified types and producing a result of specified type. For a homomorphism to exist between two heterogeneous algebras, their operations must have corresponding argument and result types under a suitable pairwise correspondence of the phyla, and for a set of functions between the corresponding phyla to constitute a homomorphism, it must respect the operations in the same way as does a conventional homomorphism, e.g. of groups. That is, a homomorphism $\phi: \underline{A}\rightarrow\underline{A}'$ of heterogeneous algebras, where S, T, ... , U, V are any phyla of $\underline{A}$, S′, T′, ..., U′, V′ the corresponding phyla of $\underline{A}'$, and where $O: S\times T\times\ldots\times U\rightarrow V$, $O': S'\times T'\times\ldots\times U'\rightarrow V'$ are any corresponding operations of $\underline{A}$ and of $\underline{A}'$, must satisfy the equation

$$\phi(O(s,t,\ldots,u)) = O'(\phi(s),\phi(t),\ldots\phi(u))$$

† Work supported by ARPA contract no. DAHCO4-72-C-0003.

for all $s \varepsilon S$, $t \varepsilon T$, ..., $u \varepsilon U$. Sets $S_0 \subseteq S$, $T_0 \subseteq T$, ..., $U_0 \subseteq U$ are said to be generating sets for $\underline{A}$ if every element of every phylum is a finite combination of elements of S_0, T_0, ..., U_0.

The bulk of the present paper will be devoted to putting some flesh on the maxims just given by applying them to the statement and partial proof of a simple compiler correctness problem. We begin at once by introducing an example programming language. (No claim whatever will be made for it as a well-designed language; it is intended merely to comprise an assortment of familiar "features" which the author thinks he knows how to describe algebraically.)

To give an intuitive idea of what is modelled, we present first a possible concrete syntax for our language. Similarities between this syntax and those of other Algol-like languages are intended to be helpful rather than misleading.

```
<st> ::= continue|<var>:=<ae>|if <be> then <st> else <st>|while <be> do <st>
            (empty, assignment, conditional, and iterative statements)

<ae> ::= <const>|val<var>|(<ae>+<ae>)|<st>res<ae>|let <var> be <ae> in <ae>
            (arithmetic expressions, but including "compound statements" and
             "blocks" with initialized declarations, which are both made to
             deliver values)

<be> ::= <ae> = <ae>|(<be>∧<be>)
            (Boolean expressions)
```

<var> and <const> are simply distinguishable sets, which for typographical convenience we will refer to henceforward as Var and Const.

We can abstractly characterize the heterogeneous algebra $\underline{L}$ which is to model our example language merely by decreeing that it is to be a word algebra, i.e. one in which no equalities hold save between identically-constructed objects, and by naming its phyla, specifying the generating sets, and giving the types of its operations. In modelling a programming language by a heterogeneous algebra, it appears there will generally be needed one phylum for each (abstract) syntactic phrase class. In conformity with this principle, we take $\underline{L}$ to have four phyla:

Var, Stat, Aexp, Bexp

with, respectively, the following generating sets:

Var, {continue}, Const, {}.

There are nine operations (whose names we borrow from the corresponding bits of concrete syntax) of the following types:

val: Var $\rightarrow$ Aexp
:= : Var $\times$ Aexp $\rightarrow$ Stat
while-do: Bexp $\times$ Stat $\rightarrow$ Stat
if-then-else: Bexp $\times$ Stat $\times$ Stat $\rightarrow$ Stat
res: Stat $\times$ Aexp $\rightarrow$ Aexp
$\wedge$: Bexp $\times$ Bexp $\rightarrow$ Bexp
=: Aexp $\times$ Aexp $\rightarrow$ Bexp
+: Aexp $\times$ Aexp $\rightarrow$ Aexp
let-be-in: Var $\times$ Aexp $\times$ Aexp $\rightarrow$ Aexp.

We next proceed to an algebraic specification of semantics for $\underline{L}$, that is to the definition of an algebra $\underline{M}$ of "meanings" and of a homomorphism $\phi: \underline{L} \rightarrow \underline{M}$. We first note that it would be irrelevant to our present purpose to commit ourselves to a particular system of "arithmetic" values. We shall merely assume that they constitute a set A, that there is a given denotation function Value: Const$\rightarrow$A, and a given operation Plus: A$\times$A$\rightarrow$A (in a practical example there would of course be many such operations taking various numbers of arguments); and we shall, where appropriate, assume that the target machine which we develop has the requisite abilities to store elements of A and to perform Plus. This evasion conforms to the usual practice in leaving details of numerical range and operations "to the implementation".

As we shall have to assign meanings to phrases with free variables, it behooves us to define a type "environment":

$$\text{Env} = (\text{Var} \overset{\sim}{\rightarrow} \text{A})$$

of partial functions assigning values to variables, and to make all our meanings functions

of environments. In fact we can also take environments to be the entities affected by assignment (although this dodge would not work for languages with a more developed reference concept, in which creation of assignable places was not co-extensive with declaration of variables). We therefore take the four phyla of $\underline{M}$ to be sets of elements of the following types:

$$\text{Var},\ \text{Env} \overset{\sim}{\to} \text{Env},\ \text{Env} \overset{\sim}{\to} (A \times \text{Env}),\ \text{Env} \overset{\sim}{\to} (\underline{2} \times \text{Env})$$

("$\underline{2}$" denotes the two-element set of truth-values).

We have still to define the operations in $\underline{M}$, and the effect of ϕ on the generating sets of $\underline{L}$. It is a fundamental result of universal algebra that when we have done so we will have specified the homomorphism ϕ uniquely, and moreover, in case the domain is a word algebra (as $\underline{L}$ is) that any so-specified ϕ actually exists (see, for example, (2) where this result is called the "unique extension lemma"). As a concession to readability, we shall combine the specifications of ϕ and of the $\underline{M}$-operations in a conventional style of recursive function definition, following the notation of (3). Observe in particular the use of the notation * for a special form of composition: if $q\colon X \to (X' \times Y)$, $p\colon X' \to (Y \to Z)$, then

$$p*q =_{df} \lambda x.p(q(x)_1)(q(x)_2).$$

The semantics:

$\phi\llbracket v\rrbracket = v$ for $v \varepsilon \text{Var}$
$\phi\llbracket \underline{\text{continue}}\rrbracket(e) = e$ for $e \varepsilon \text{Env}$ (here and subsequently)
$\phi\llbracket \overline{c}\rrbracket(e) = \text{Value}(c),e$ for $c \varepsilon \text{Const}$
$\phi\llbracket \underline{\text{val}}\ v\rrbracket(e) = e(\phi\llbracket v\rrbracket),e$
$\phi\llbracket v{:=}r\rrbracket = (\lambda a\lambda e.\lambda w.w{=}\phi\llbracket v\rrbracket \to a, e(w))*\phi\llbracket r\rrbracket$
$\phi\llbracket \underline{\text{while}}\ q\ \underline{\text{do}}\ s\rrbracket = Y(\lambda f.(\lambda b.b \to f \circ \phi\llbracket s\rrbracket, \lambda e.e)*\phi\llbracket q\rrbracket)(\Omega)$
(here Y is the minimal fixpoint combinator, $\Omega\colon \text{Env} \overset{\sim}{\to} \text{Env}$ is the totally undefined function)
$\phi\llbracket \underline{\text{if}}\ q\ \underline{\text{then}}\ s_1\ \underline{\text{else}}\ s_2\rrbracket = (\lambda b.b \to \phi\llbracket s_1\rrbracket, \phi\llbracket s_2\rrbracket)*\phi\llbracket q\rrbracket$
$\phi\llbracket s\ \underline{\text{res}}\ r\rrbracket(e) = \phi\llbracket r\rrbracket(\phi\llbracket s\rrbracket(e))$
$\phi\llbracket q_1 \wedge q_2\rrbracket = (\lambda b.b \to \phi\llbracket q_2\rrbracket, (\lambda e.\underline{\text{false}}, e))*\phi\llbracket q_1\rrbracket$
$\phi\llbracket r_1{=}r_2\rrbracket = (\lambda a_1.(\lambda a_2\lambda e.a_1{=}a_2, e)*\phi\llbracket r_2\rrbracket)*\phi\llbracket r_1\rrbracket$
$\phi\llbracket r_1{+}r_2\rrbracket = (\lambda a_1.(\lambda a_2\lambda e.a_1{+}a_2, e)*\phi\llbracket r_2\rrbracket)*\phi\llbracket r_1\rrbracket$
$\phi\llbracket \underline{\text{let}}\ v\ \underline{\text{be}}\ r_1\ \underline{\text{in}}\ r_2\rrbracket = (\lambda a_1\lambda e_1.((\lambda a_2\lambda e_2.a_2,(\lambda w.w{=}\phi\llbracket v\rrbracket \to e_1(w), e_2(w)))$
$*\phi\llbracket r_2\rrbracket)(\lambda w.w{=}\phi\llbracket v\rrbracket \to a_1, e_1(w)))*\phi\llbracket r_1\rrbracket.$

Note well that although the definition of ϕ (and others to follow) reads like a program in a ligh-level language, and may be so interpreted, yet it must be a program of a rigidly restricted kind if it is to define a homomorphism. Each line of the definition (save those defining ϕ outright on the generating sets) is specifying an operation in $\underline{M}$. It follows that each $\phi(O(s,t,\ldots,u))$ must be defined to depend solely upon $\phi(s)$, $\phi(t)$, $\ldots$, $\phi(u)$, and not upon any other functions of $s,t,\ldots,u$.

We now turn to the definition of a target language $\underline{T}$ for our example, and of a compiling homomorphism $\gamma\colon \underline{L} \to \underline{T}$. Note in passing, by way of motivation, that a straightforward "syntax-directed" compiler in the real world does bear a coarse resemblance to a program for computing a homomorphism: it follows the phrase structure of its input, and (unlike an interpreter) gets done with any finite program in a finite, indeed roughly linear, amount of time.

Roughly speaking, the programs of $\underline{T}$ will be flowcharts, to be interpreted (to no one's surprise) on a stack machine; the operations of $\underline{T}$ will be ones which stitch flowcharts together into bigger flowcharts. Two difficulties arise, however. The first is caused by the existance of $\underline{L}$-phrases with free variables - we cannot expect to compile such phrases into runnable flowcharts. Our solution here is similar to that used to define the semantics of $\underline{L}$: we introduce a set of symbol tables or "maps".

$$\text{Map} = (\text{Var} \xrightarrow[\text{one-to-one}]{\sim} \mathbb{N})$$

(it will turn out that the natural-number values of maps will represent distances down from the top of the stack), and we will make each phylum of $\underline{T}$ be a set of functions from Map to some suitable class of flowcharts.

The second difficulty arises quite generally in "translating" from one (language represented by an) algebra to another: in general the given second algebra will not be a homomorphic image of the first; what is required is to construct, or "derive" - see the discussion of derived operations in (2) - the operations of the actual target algebra from the given operations by composition and the use of constant operands. In our case, realism demands that we define an underlying flowchart algebra $\underline{T}_0$, not biased towards the compilation of $\underline{L}$-programs, and then derive the operations of $\underline{T}$ as compounds of $\underline{T}_0$-operations.

We make the following rather mysterious definition of $\underline{T}_0$:

1) For every finite set D, there is a separate phylum of $\underline{T}_0$, which we may call the D-flowcharts. The intuitive idea is that elements of D label points in each D-flowchart at which it may be attached to others. In particular there will be a phylum of {S,H}-flowcharts (S for "start", H for "halt") with one entry point and one exit point. Note that S and H are not meant for variables of any kind, but simply as two particular objects - the nineteenth and eighth letters of the alphabet.

2) For any three sets D, E, F and any three functions d: $D\to\mathbb{N}$, e: $E\to\mathbb{N}$, f: $F\to\mathbb{N}$, there is a binary operation O of $\underline{T}_0$ producing an F-flowchart from a D-flowchart and an E-flowchart. Intuitively O "works" by collapsing into one point each set of points of the D- and E-flowcharts whose labels map to a single integer, and then relabelling some of these collapsed points with the elements of F to give an F-flowchart. ($\mathbb{N}$ is used merely as a convenient large-enough set.) For example, end-to-end composition of {S,H}-flowcharts is given by O_1, defined as follows:

$$f_1(S) = d_1(S) = 0$$

$$f_1(H) = e_1(H) = 2$$

$$d_1(H) = e_1(S) = 1$$

(The actual numeric values on the right are plainly giving no useful information; we will suppress them from here on.) The author would be the first to agree that the $\underline{T}_0$-operations are cumbersome and peculiar. They have the merit, however, of permitting the creation of flowcharts with loops.

3) $\underline{T}_0$ contains at least the following primitive flowcharts, or "instructions":

For each $n\in\mathbb{N}$, {S,H}-flowcharts

L_n (for loading the nth stack element to the top of the stack)

and

St_n (for removing the top of stack and storing it n positions down);

for each $a\in A$, an {S,H}-flowchart

$Limm_a$ ("load immediate" - for putting a on the stack);

one special {S,H}-flowchart

Add (for replacing the top two stack elements by their sum);

and one special {S,T,F}-flowchart

Equal (for deleting the top two stack elements and exiting at T if they were equal, otherwise at F).

There are also empty flowcharts having all their labels at the same point.

For the phyla of our derived target language $\underline{T}$, homologous with $\underline{L}$, we will take Var again, two sets of partial functions of type (Map$\dashrightarrow$ {S,H}-flowcharts) to correspond to Stat and Aexp, and one set of partial functions of type (Map$\dashrightarrow$ {S,T,F}-flowcharts) to correspond to Bexp. We may now proceed, in the same style as for ϕ and $\underline{M}$, to give recursive equations defining the effect both of the compiling homomorphism $\overline{\gamma}$: $\underline{L}\to\underline{T}$ and of the operations in $\underline{T}$. The remaining definitions of needed $\underline{T}_0$-operations are collected at the end.

We make one auxiliary definition:

$$\text{Push}(m)(v) = m(v)+1 \qquad \text{for } m\in\text{Map},\ v\in\text{Var}.$$

Now the compiler:

$\gamma[\![v]\!] = v$ for $v \varepsilon Var$
$\gamma[\![\underline{continue}]\!](m)$ = the empty $\{S,H\}$-flowchart
for $m\varepsilon Map$ (here and subsequently)
$\gamma[\![c]\!](m) = Limm_{Value(c)}$ for $c\varepsilon Const$

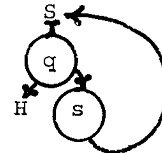
$\gamma[\![\underline{val}\ v]\!](m) = L_{m(v)}$
$\gamma[\![v{:=}r]\!](m) = O_1(\gamma[\![r]\!](m), St_{m(v)})$
$\gamma[\![\underline{while}\ q\ \underline{do}\ s]\!](m) = O_2(\gamma[\![q]\!](m), \gamma[\![s]\!](m))$

$\gamma[\![\underline{if}\ q\ \underline{then}\ s_1\ \underline{else}\ s_2]\!](m) = O_4(O_3(\gamma[\![q]\!](m), \gamma[\![s_1]\!](m)), \gamma[\![s_2]\!](m))$
$\gamma[\![s\ \underline{res}\ r]\!](m) = O_1(\gamma[\![s]\!](m), \gamma[\![r]\!](m))$
$\gamma[\![q_1 \wedge q_2]\!](m) = O_5(\gamma[\![q_1]\!](m), \gamma[\![q_2]\!](m))$

$\gamma[\![r_1{=}r_2]\!](m) = O_1(\gamma[\![r_1]\!](m), O_1(\gamma[\![r_2]\!](Push(m)), Equal))$
$\gamma[\![r_1{+}r_2]\!](m) = O_1(\gamma[\![r_1]\!](m), O_1(\gamma[\![r_2]\!](Push(m)), Add))$
$\gamma[\![\underline{let}\ v\ \underline{be}\ r_1\ \underline{in}\ r_2]\!](m) = O_1(\gamma[\![r_1]\!](m), O_1(\gamma[\![r_2]\!](\lambda w.w{=}v{\rightarrow}0, Push(m)(w)), St_0))$
(This is a trick: St_0 has the effect of deleting the stack element below the top one.)

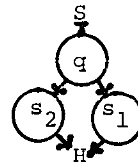
We now define $O_2 - O_5$, but we append pictures of their intended effects for those who understandably may not wish to puzzle out the equations:

O_2: $f_2(S) = d_2(S) = e_2(H)$
$f_2(H) = d_2(F)$
$e_2(S) = d_2(T)$



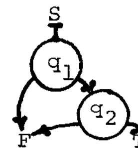
O_3: $f_3(X) = d_3(S)$
$f_3(Y) = d_3(F)$
$e_3(S) = d_3(T)$
$f_3(Z) = e_3(H)$

O_4: $f_4(S) = d_4(X)$
$f_4(H) = d_4(Z) = e_4(H)$
$e_4(S) = d_4(Y)$



(O_3 produces $\{X,Y,Z\}$-flowcharts, which are immediately consumed by O_4)

O_5: $f_5(S) = d_5(S)$
$f_5(F) = d_5(F) = e_5(F)$
$e_5(S) = d_5(T)$
$f_5(T) = e_5(T)$



We next turn to the description of $\underline{U}_0$, the algebra of meanings for $\underline{T}_0$. Intuitively speaking, we want to interpret our flowcharts by running them on a machine whose set of states, Σ, consists of the "stacks" of elements of A, that is of all finite sequences with values in A. (For $\sigma\varepsilon\Sigma$, we will consider $\sigma(0)$ to be the top stack element, and so on down.) For each phylum of D-flowcharts, we take its phylum of meanings to consist of binary relations from $D\times\Sigma$ to itself. (We denote the set of all such relations by $(D\times\Sigma \longrightarrow\!\!\!\!\times\ D\times\Sigma)$). Thus, intuitively, the meaning of a flowchart will tell us just what <initial state, final state> pairs it can compute from each entrance to each exit. (The omission of any formal distinction between entrances and exits is perhaps unnatural, but seems to

facilitate the treatment.)

Corresponding to each $\underline{T}_0$-operation

O: D-flowcharts × E-flowcharts → F-flowcharts

determined by functions d,e,f, we decree an operation Q of $\underline{U}_0$,

$$Q: (D\times\Sigma \rightarrow\!\!\!\times D\times\Sigma)\times(E\times\Sigma \rightarrow\!\!\!\times E\times\Sigma) \rightarrow (F\times\Sigma \rightarrow\!\!\!\times F\times\Sigma)$$

defined as follows:

$$Q(R_1,R_2) = f\times I;(d^{-1}\times I;R;d\times I \cup e^{-1}\times I;R_2;e\times I)^{+};f^{-1}\times I.$$

Here $\times$, $;$, $\cup$, $^{-1}$, $^{+}$ denote respectively relational Cartesian product, composition, union, converse, and ancestral, and I denotes the identity relation on Σ. The intuitive content of this formula is that to compute in a flowchart sewn together from pieces is to compute by turns in the pieces, jumping back and forth as often as one pleases at the stitches.†

In particular, we denote by $Q_1,\ldots,Q_5$ the $\underline{U}_0$-operations corresponding to $O_1,\ldots,O_5$ in $\underline{T}_0$.

By way of apology for the mysteriousness of $\underline{T}_0$ and $\underline{U}_0$, it may be said that they are intended to conform to an algebraic model of flowcharts and machines introduced by Landin in (4) and further developed in (5). In that model, individual flowcharts, and the machines on which they run, are viewed as algebras of a special type; meanings (relations computed by flowcharts running on any machine) are given by a uniform algebraic product operation. In (5), explicit constructions on flowchart-algebras are defined, corresponding pretty closely to the $\underline{T}_0$-operations of the present paper, and a formula essentially equivalent to that postulated above as giving the connection between the $\underline{T}_0$-operations and the $\underline{U}_0$-operations is there derived from the properties of the algebraic product.

Having defined the operations in $\underline{U}_0$, we may now specify a semantic homomorphism $\psi_0: \underline{T}_0 \to \underline{U}_0$ merely by giving its effect on the generating sets of $\underline{T}_0$, that is on the individual instructions. We do so as follows (for brevity, let $a\cdot\sigma$ denote the result of prefixing the element a to the stack σ).

$\psi_0(L_n)$: $X,\sigma \mapsto Y,\tau$ iff $X = S$ & $Y = H$ & $\tau = \sigma_n\cdot\sigma$

$\psi_0(St_n)$: $X,\sigma \mapsto Y,\tau$ iff $X = S$ & $Y = H$ & $\tau_i = \sigma_{i+1} (i\neq n)$ & $\tau_n = \sigma_0$

$\psi_0(Limm_a)$: $X,\sigma \mapsto Y,\tau$ iff $X = S$ & $Y = H$ & $\tau = a\cdot\sigma$

$\psi_0(Add)$: $X,\sigma \mapsto Y,\tau$ iff $X = S$ & $Y = H$ & $\sigma = \sigma_0\cdot\sigma_1\cdot\rho$ & $\tau = Plus(\sigma_0,\sigma_1)\cdot\rho$

$\psi_0(Equal)$: $X,\sigma \mapsto Y,\tau$ iff $X = S$ & $\sigma = \sigma_0\cdot\sigma_1\cdot\tau$ & $(Y = T$ & $\sigma_0 = \sigma_1$ or $Y = F$ & $\sigma_0 \neq \sigma_1)$

ψ_0(empty {S,H}-flowchart): $X,\sigma \mapsto Y,\tau$ iff $X = S$ & $Y = H$ & $\sigma = \tau$

It is now possible to define the algebra $\underline{U}$ which is to form the lower right-hand corner of our compiler correctness diagram, together with the target semantic homomorphism ψ. Evidently the elements of $\underline{U}$ must follow those of $\underline{T}$ in being functions of symbol tables - to be precise we have the following for the types of the members of the four phyla of $\underline{U}$:

Var

$Map \overset{\sim}{\to} (\{S,H\}\times\Sigma \rightarrow\!\!\!\times \{S,H\}\times\Sigma)$

$Map \overset{\sim}{\to} (\{S,H\}\times\Sigma \rightarrow\!\!\!\times \{S,H\}\times\Sigma)$

$Map \overset{\sim}{\to} (\{S,T,F\}\times\Sigma \rightarrow\!\!\!\times \{S,T,F\}\times\Sigma)$.

We of course want ψ to reflect the action of the underlying flowcharts-to-meanings homomorphism ψ_0. That is, we can define the desired effect of ψ, considered simply as a mapping, by the equations

$\psi(v) = v$ for $v\in$Var

$\psi(p)(m) = \psi_0(p(m))$ for $m\in$Map, $p\in$ any other $\underline{T}$-phylum.

That this function ψ actually is a homomorphism: $\underline{T}\to\underline{U}$ for an appropriate choice of operations in $\underline{U}$ is a fact of universal algebra, stated for homogeneous algebras in (2) under the name "homomorphism of restrictions lemma". It is convenient to display the necessary $\underline{U}$-operations in the form of a recursive definition of the composite $\psi\circ\gamma$. (Note that we are constructing both $\underline{T}$ and $\underline{U}$ to be epimorphic images of $\underline{L}$ under γ and $\psi\circ\gamma$, so that this form of definition is indeed complete.) In fact, we can obtain the definition of $\psi\circ\gamma$ mechanically from that of γ by simply replacing each $\underline{T}_0$-constant Inst by ψ_0(Inst), each O_i by Q_i, and each γ by $\psi\circ\gamma$; however we reproduce the result here for reference:

† To conform more nearly to (5), we should take * here rather than +, i.e. make all relations reflexive; to do so would merely complicate some formulas below.

$\psi \circ \gamma [\![v]\!] = v$
$\psi \circ \gamma [\![\underline{continue}]\!](m) = \psi_o(\text{empty}\{S,H\} \to \text{flowchart})$
$\psi \circ \gamma [\![c]\!](m) = \psi_o(\text{Limm}_{\text{Value}(c)})$
$\psi \circ \gamma [\![\underline{val}\ v]\!](m) = \psi_o(L_{m(v)})$
$\psi \circ \gamma [\![v := r]\!](m) = Q_1(\psi \circ \gamma [\![r]\!](m), \psi_o(St_{m(v)}))$
$\psi \circ \gamma [\![\underline{while}\ q\ \underline{do}\ s]\!](m) = Q_2(\psi \circ \gamma [\![q]\!](m), \psi \circ \gamma [\![s]\!](m))$
$\psi \circ \gamma [\![\underline{if}\ q\ \underline{then}\ s_1\ \underline{else}\ s_2]\!](m) = Q_4(Q_3(\psi \circ \gamma [\![q]\!](m), \psi \circ \gamma [\![s_1]\!](m)), \psi \circ \gamma [\![s_2]\!](m))$
$\psi \circ \gamma [\![s\ \underline{res}\ r]\!](m) = Q_1(\psi \circ \gamma [\![s]\!](m), \psi \circ \gamma [\![r]\!](m))$
$\psi \circ \gamma [\![q_1 \wedge q_2]\!](m) = Q_5(\psi \circ \gamma [\![q_1]\!](m), \psi \circ \gamma [\![q_2]\!](m))$
$\psi \circ \gamma [\![r_1 = r_2]\!](m) = Q_1(\psi \circ \gamma [\![r_1]\!](m), Q_1(\psi \circ \gamma [\![r_2]\!](\text{Push}(m)), \psi_o(\text{Equal})))$
$\psi \circ \gamma [\![r_1 + r_2]\!](m) = Q_1(\psi \circ \gamma [\![r_1]\!](m), Q_1(\psi \circ \gamma [\![r_2]\!](\text{Push}(m)), \psi_o(\text{Add})))$
$\psi \circ \gamma [\![\underline{let}\ v\ \underline{be}\ r_1\ \underline{in}\ r_2]\!](m) = Q_1(\psi \circ \gamma [\![r_1]\!](m),$
$\qquad Q_1(\psi \circ \gamma [\![r_2]\!](\lambda w. w = v \to 0, \text{Push}(m)(w)), \psi_o(St_0)))$

It is now necessary to define the fourth side of the compiler correctness diagram, translating between source-and target-language meanings. It proves more convenient to define an "encoding" function $\varepsilon: \underline{M} \to \underline{U}$ than one in the opposite direction; it will then be necessary as a final step in proving correctness to show that ε has a "decoding" inverse, $\delta: \underline{U} \to \underline{M}$, at least for programs without free variables, which are the only ones we really expect to be able to run. We will define ε simply as a family of mappings (one per phylum); the proof that ε is in fact a homomorphism will constitute the main (inductive) part of the correctness proof.

To get an intuitive grasp of the behavior of ε, it is necessary to realize that stacks in the implementation contain two kinds of entry: values of variables and anonymous intermediate results. Thus given, for example, a function s: Env$\to$Env in the statement phylum of $\underline{M}$, we want $\varepsilon(s)$ to be such that - once provided with a symbol table telling which stack positions represent the current environment - it will change these just as s does but will leave all other positions unchanged. We encapsulate this notion by defining an auxiliary function for changing a stack to bring it into agreement with an environment:

$$\text{Change}(e,m,\sigma) = \lambda i. i \varepsilon \text{range}(m) \to e \circ m^{-1}(i), \sigma_i \qquad (\text{for } e \varepsilon \text{Env}, m \varepsilon \text{Map}, \sigma \varepsilon \Sigma).$$

It is now not too difficult to write the necessary four equations defining ε:

$$\varepsilon(v) = v \qquad \text{for } v \varepsilon \text{ Var}$$
$$\varepsilon(s)(m)(S,\sigma) = H, \text{Change}(s(\sigma \circ m), m, \sigma) \qquad \text{for } s \varepsilon \text{Im}_\phi(\text{Stat})$$
$$\varepsilon(r)(m)(S,\sigma) = (\lambda a \lambda e. H, a \cdot \text{Change}(e,m,\sigma))^* r(\sigma \circ m) \qquad \text{for } r \varepsilon \text{Im}_\phi(\text{Aexp})$$
$$\varepsilon(q)(m)(S,\sigma) = (\lambda b \lambda e. (b \to T, F), \text{Change}(e,m,\sigma))^* q(\sigma \circ m) \qquad \text{for } q \varepsilon \text{Im}_\phi(\text{Bexp})$$

The peculiar "formal parameter" notation (S,σ) is meant to indicate that $\varepsilon(s)(m)$ and the like are partial functions (strictly speaking, they must be considered as the corresponding relations to make all the types fit) which are defined only when the first component of the argument is the letter S.

At this point, although nothing has as yet been proved, the main goal of the paper has been accomplished: to carry out in detail the modelling of a compiler correctness problem within algebra, in what the author believes to be a natural manner. To actually carry through the correctness proof, which we shall do here only in part, is to prove the equation

$$\varepsilon \circ \phi = \psi \circ \gamma \quad .$$

The necessary proof has been structured for us by our algebraic approach; namely we have in the first place to show that we have not merely a function but a homomorphism $\varepsilon: \underline{M} \to \underline{U}$, that is that ε respects each of the nine operations of $\underline{L}$; and in the second place to show directly that the diagram commutes for the elements of the generating sets of $\underline{L}$. It will then follow by the unique extension lemma that $\varepsilon \circ \phi$ and $\psi \circ \gamma$, agreeing on the generating sets, must be the same homomorphism. If we wished to eschew algebraic terminology, we could call these two parts the inductive and base steps of a "proof by structural induction" (but note, this is an induction on the structure of source programs, not of computations). After proving that the diagram commutes, we must still show that for a program p without free variables we actually can invert ε, and recover $\phi(p)$ from $\psi \circ \gamma(p)$.

Proving commutativity for the generating sets is not hard. For the phylum Var, which was only carried along for bookkeeping purposes, we have trivially $\varepsilon\circ\phi[\![v]\!]=v=\psi\circ\gamma[\![v]\!]$. For $c\varepsilon$Const, it is easy to discover from the definitions, taking advantage of the notational convention used in defining ε, that both $\varepsilon\circ\phi[\![c]\!]$ and $\psi\circ\gamma[\![c]\!]$ work out to the relation-valued function

$$\lambda m.\lambda(S,\sigma).H,\ \mathrm{Value}(c)\cdot\sigma$$

Similarly, the definitions yield immediately that

$$\varepsilon\circ\phi[\![\underline{\mathrm{continue}}]\!] = \psi\circ\gamma[\![\underline{\mathrm{continue}}]\!] = \lambda m.\lambda(S,\sigma).H,\sigma.$$

This completes the base of the induction.

The inductive part of the proof verifying that ε respects the operations of $\underline{L}$ - we shall carry out here only for the operations $\underline{\mathrm{val}}$ and :=.

For $\underline{\mathrm{val}}$ we can in fact check commutitivity directly, since we know that all the homomorphisms involved are identities on Var. We compute from the definitions, for arbitrary $v\varepsilon$Var:

$$\begin{aligned}\varepsilon\circ\phi[\![\underline{\mathrm{val}}\ v]\!] &= \varepsilon(\lambda e.e(v),e)\\ &= \lambda m.\lambda(S,\sigma).H,(\sigma\circ m)(v)\cdot\mathrm{Change}(\sigma\circ m,m,\sigma)\\ &= \lambda m.\lambda(S,\sigma).H,(\sigma\circ m)(v)\cdot\sigma\end{aligned}$$

and

$$\begin{aligned}\psi\circ\gamma[\![\underline{\mathrm{val}}\ v]\!] &= \psi(\lambda m.L_{m(v)})\\ &= \lambda m.\lambda(S,\sigma).H,\sigma_{m(v)}\cdot\sigma\end{aligned}$$

which is the same thing.

For :=, what we have to prove (for $v\varepsilon$Var,$r\varepsilon$Aexp) is the equation

$$\varepsilon\circ\phi[\![v:=r]\!] = \psi\circ\gamma[\![v:=r]\!]$$

under the assumption

$$\psi\circ\gamma[\![r]\!] = \varepsilon\circ\phi[\![r]\!]$$

which may be rewritten, from the definitions of ε and ϕ, as the assumption

$$\psi\circ\gamma[\![r]\!] = \lambda m.\lambda(S,\sigma).(\lambda a\lambda e.H,a\cdot\mathrm{Change}(e,m,\sigma))*\phi[\![r]\!](\sigma\circ m) \quad .$$

We may begin by expanding the left-hand side:

$$\begin{aligned}\varepsilon\circ\phi[\![v:=r]\!] &= \varepsilon((\lambda a\lambda e.\lambda w.w{=}v\rightarrow a,e(w))*\phi[\![r]\!])\\ &= \lambda m.\lambda(S,\sigma).H,\ \mathrm{Change}((\lambda a\lambda e.\lambda w.w{=}v\rightarrow a,e(w))*\phi[\![r]\!](\sigma\circ m),m,\sigma).\end{aligned}$$

Starting now from the other end, we work out what $\psi\circ\gamma[\![v:=r]\!]$ comes to. By giving some attention to the action of Q_1, it is not difficult to see that we have, for $m\varepsilon$Map:

$$\psi\circ\gamma[\![v:=r]\!](m):X,\sigma \longmapsto Y,\tau \ \underline{\mathrm{iff}}\ X{=}S\ \&\ Y{=}H\ \&\ \psi\circ\gamma[\![r]\!]:S,\sigma \longmapsto H,a\cdot\rho \ \&\ \tau=(\lambda i.i{=}m(v)\rightarrow a,\rho_i) \quad .$$

But this amounts (taking note of what $\psi\circ\gamma[\![r]\!]$ does) to

$$\psi\circ\gamma[\![v:=r]\!]=\lambda m.\lambda(S,\sigma).H,(\lambda a\lambda e.\lambda i.i{=}m(v)\rightarrow a,\mathrm{Change}(e,m,\sigma)(i))*\phi[\![r]\!](\sigma\circ m)$$

which, considering the definition of Change, we may rewrite as

$$\psi\circ\gamma[\![v:=r]\!]=\lambda m.\lambda(S,\sigma).H,\ \mathrm{Change}((\lambda a\lambda e.\lambda w.w{=}v\rightarrow a,e(w))*\phi[\![r]\!](\sigma\circ m),m,\sigma)$$

and this, as we have seen, is just $\varepsilon\circ\phi[\![v:=r]\!]$.

Supposing the proof that $\varepsilon\circ\phi = \psi\circ\gamma$ complete, we now examine the special case of "runnable" programs - those containing no free variables, and belonging, we assume, to the phylum Aexp. Consider such a program p, and let e_o denote the empty (completely undefined) environment, m_o the empty symbol table, and ζ the empty stack. Then the answer we are after by running p is $\phi(p)(e_o)$-in fact just the first component of this, for the second component will be e_o again. But our statement of commutativity tells us that we can compile the flowchart $\gamma(p)(m_o)$, run it on the target machine starting at (S,ζ) and come out with

$$(\lambda a\lambda e.H,a\cdot\zeta)*\phi(p)(e_o),$$

just in case $\phi(p)(e_o)$ is defined-that is, with the desired element of A as the only stack element.

The omitted parts of the correctness proof which has been prepared for and begun above appear to offer no particular difficulty, save possibly in the case of the $\underline{\mathrm{while}}$ statement, where Y will have to be either proven (by showing its argument to be a

continuous functional) or else defined to be equivalent in this application to $\lambda F.\lambda f.\bigcup_{n\geq 0}(F^{(n)}(f))$. Plainly, however, the proof would be quite tedious when written out in full. To some extent this appears inevitable, since the definitions of the compiler and of the two semantic functions can hardly be made very much shorter than they are, and yet any proof must appeal to every syllable of these definitions. There is a pressing need for notational improvements and, if reliable proofs are to be produced, for a sufficiently expressive (and preferably partly automated) formal system to carry them out in. The present paper, aside from its insistence on homomorphisms everywhere, is basically unsystematic in its approach; the aim has been to uncover some of the mathematical facts underlying the correctness of compilers. The author hopes that efforts such as this one will complement the endeavors of the system-builders, such as Milner's LCF, applied to a compiler-correctness problem in (6), and also Hitchcock and Park's (7); designing good proving languages should be easier if one has an idea first of what one will want to say in them.

References

(1) G. Birkhoff and J.D. Lipson, "Heterogeneous Algebras", J. Combinatorial Theory 8, pp. 115-133 (1970).

(2) R. M. Burstall and P. J. Landin, "Programs and Their Proofs: an Algebraic Approach", Machine Intelligence 4 (1969).

(3) D. Scott and C. Strachey, "Towards a Mathematical Semantics for Computer Languages", Programming Research Group Monograph PRG-6, Oxford Computing Laboratory (1971).

(4) P. J. Landin, "A Program-Machine Symmetric Automata Theory", Machine Intelligence 5 (1970).

(5) F. L. Morris, "Correctness of Translations of Programming Languages", Stanford Computer Science Memo CS 72-303 (1972).

(6) R. Milner and R. Weyhrauch, "Proving Compiler Correctness in a Mechanized Logic", Machine Intelligence 7 (1972).

(7) P. Hitchcock and D. Park, "Induction Rules and Proofs of Termination", Proc. Colloques IRIA Theorie des automates des langages et de la Programmation. Rocquencourt, France, July 1972.