STANFORD ARTIFICIAL INTELLIGENCE PROJECT
MEMO AIM-174

STAN-CS-72-303

AD-A954 771

CORRECTNESS OF TRANSLATIONS OF PROGRAMMING LANGUAGES -- AN ALGEBRAIC APPROACH

BY

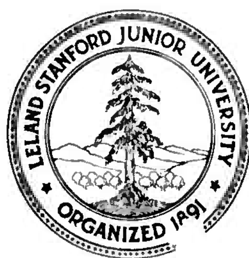
FRANCIS LOCKWOOD MORRIS

SUPPORTED BY
NATIONAL SCIENCE FOUNDATION
AND
ADVANCED RESEARCH PROJECTS AGENCY
ARPA ORDER NO. 457

AUGUST 1972

APPROVED FOR PUBLIC RELEASE;
DISTRIBUTION IS UNLIMITED (A)

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY



DTIC FILE COPY

DTIC
S ELECTE D
JUN 1 7 1985
G

M.S.F.

85 06 13 012

STANFORD ARTIFICIAL INTELLIGENCE PROJECT
MEMO AIM-174

COMPUTER SCIENCE DEPARTMENT
REPORT CS-303

AUGUST 1972

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A/1	

UNANNOUNCED

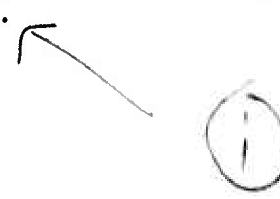
CORRECTNESS OF TRANSLATIONS OF PROGRAMMING LANGUAGES -- AN ALGEBRAIC APPROACH

Francis Lockwood Morris

Abstract

Programming languages and their sets of meanings can be modelled by general operator algebras; semantic functions and compiling functions by homomorphisms of operator algebras. A restricted class of individual programs, machines, and computations can be modelled in a uniform manner by binary relational algebras. These two applications of algebra to computing are compatible: the semantic function provided by interpreting ("running") one binary relational algebra on another is a homomorphism on an operator algebra whose elements are binary relational algebras.

Using these mathematical tools, proofs can be provided systematically of the correctness of compilers for fragmentary programming languages, each embodying a single language "feature". Exemplary proofs are given for statement sequences, arithmetic expressions, Boolean expressions, assignment statements, and while statements. Moreover, proofs of this sort can be combined to provide (synthetic) proofs for, in principle, many different complete programming languages. One example of such a synthesis is given.



This research is supported in part by the National Science Foundation under grant number NSF GJ-776 and in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD-183).

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency of the U.S. Government.

Reproduced in the U.S.A. Available from the National Technical Information Service. Springfield, Virginia 22151.

Table of Contents

O. Introduction, Notation, and Organization 1

I. Modelling Computing Devices and Computations by Binary Relational Algebras 11

II. Simulation and Categories 16

III. The Relation Computed by a BRA 26

IV. Semantics of Programming Languages 29

V. Compilers are Homomorphisms 36

VI. Semantics of BRAs is a Homomorphism 45

VII. The Compiler Composition Theorem 54

VIII. The General Plan for Simple Proofs of Compiler Correctness 65

IX. Proofs for Examples SS, BE, AE 69

X. Stores and Assignment 81

XI. While Statements 94

XII. An Exemplary Synthesis 100

XIII. Conclusion 116

References 124

0. Introduction, Notation, and Organization

The aim of this work is to contribute to the mathematical theory of programming language semantics and of translations from one programming language to another, and in particular to bring within nearer reach the feasibility of proving the correctness of rules (compilers) for performing such translations.

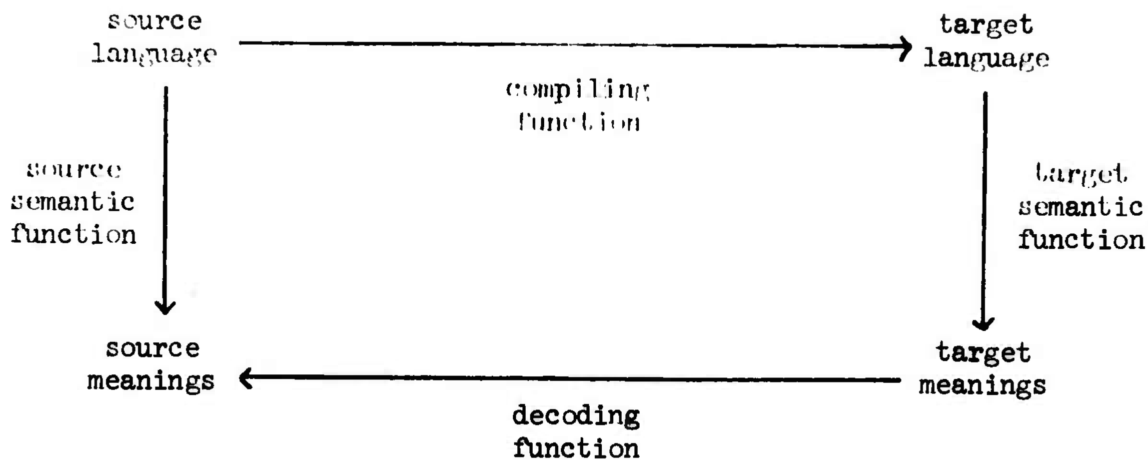
McCarthy [McC 62] appears to have been the first to have drawn attention to the possibility and desirability of making such proofs, and the approach to be followed here agrees with the essential points of that which he proposed: that a program should be regarded as denoting a partial function; that a compiler is a program-valued function of programs whose responsibility is to give a result denoting the same partial function as did its argument ("same", that is, after making allowance for any necessary encoding of program data and decoding of program results); and that in all reasonable cases both the denotations of programs and the effects of compilers can be given by definitions which follow the abstract syntactic structure of the source language programs, yielding the denotation or translation of a complex program by some operation on the denotations (respectively translations) of its syntactic constituents.

The development of general techniques for proving properties of algorithms, in particular the equivalence of two algorithms, as initiated by McCarthy [McC 63] and carried on by Floyd, Manna, and many others, has provided us with a sufficiently powerful theory to make us expect that if we are given an algorithm for compiling from one language to another, together with algorithms (interpreters) for finding the result

of a program applied to data in both languages, and algorithms for transforming between source and target data representations, we should in principle be able to find a proof of the correctness of the compiler (supposing it to be in fact correct). Such a proof was actually carried out by Painter [Paint 67] for a small language. Encouraging as his result is, it may be said that it also provides a warning that the bane of computer programming -- the natural incomprehensibility of large and even medium-sized programs -- carries over in full force to proofs about programs.

It is natural to hope that if we can find more structure in the compiler-correctness problem than in the general equivalence-of-algorithms problem, we may be able to give to compiler verification the characteristics of a typical area of applied mathematics, in which small results with intuitive content are proved once and applied many times, and to liberate it from the incomprehensibility and duplication of effort so characteristic of programming. The main goal of this thesis is to demonstrate that this hope can to a considerable extent be realized.

In the author's opinion, the essential first step in structuring the compiler-correctness is to reject the view that the semantics of a programming language may be given by any algorithm for getting a result from program and data together, and instead to demand the literal assignment of <u>meanings</u> of whatever mathematical type may be appropriate -- generally functions or relations of some kind -- directly to programs and program constituents. Having made this decision, then even regarding a language and the associated meanings merely as sets, we have that proving a compiler correct is always proving the commutativity of a square diagram of the form:



Insofar as possible, we would like to consider the mathematical properties of the functions indicated by the four arrows, rather than properties of any particular programs for computing them. To succeed in this aim we will have to be able to give mathematical descriptions, rather than descriptions by programs, of these functions. (And indeed, in applications where the correctness of an actual compiling program must be proved, an additional step, ignored in this thesis, will be needed: a verification that the compiler does compute the compiling function which we have discovered to be correct.)

We can do better than to regard the corners of the above diagram merely as sets. We shall make extensive application of the idea of Burstall and Landin [Burs 69], that it is possible to recognize an algebraic structure in the source language, and to impose corresponding structure on the other corners of the diagram, in such a way that the arrows become not merely functions but homomorphisms, and many results depending on an induction on the structure of programs can be obtained in a uniform way as applications of basic results of (universal) algebra about the existence and uniqueness of homorphisms.

A second application of algebra to the theory of computation of which heavy use will be made in what follows comes from Landin [Land 70]. The idea here is that each single program can be regarded as an algebra, of which the operations may be taken as partial functions, or somewhat more generally as relations; likewise any suitable interpreting machine for a language can be regarded as an algebra of the same sort. The possible computations of a given program on a given machine then become a product algebra derived from the program- and machine-algebras.

Landin's idea will be developed here only for the case that all operations of program- and machine-algebras can be taken to be binary relations (or unary functions). So restricted, its applicability is certainly much more limited than is that of the programming-language-as-algebra model; nevertheless, it makes possible a uniform treatment of a surprisingly large class of examples. (Our terminology will be to apply the abbreviation "BRA" -- for binary relational algebra -- to these program- and machine-algebras, and the phrase "operator algebra" to general language- and meaning-algebras.)

The specific techniques of compiler verification we shall develop will be primarily applicable to a situation in which the source language is an arbitrary operator algebra, but the target language, as well as being in its entirety an operator algebra, will also have its individual programs be binary relational algebras. It is not claimed that all instances of compiling can be adequately modelled by this scheme; rather our restriction on target language-algebras biases us towards modelling compilation into low-level languages. The modelling of target programs by BRAs has a claim to be called natural to the extent to which we

believe that the ultimate fate of any program is to be obeyed by a sequential machine (a concept which does not exclude the sort of non-deterministic machine underlying Dijkstra's concept of co-operating sequential processes).

The development here of some properties of binary relational algebras will be cast in the framework of very elementary category theory. The exposition of those elements of category theory we shall need is intended to be completely self-contained, and in any case does not go beyond, or even up to, the limits of what is contained in [MacL 67]. It is perhaps necessary to defend the introduction of terminology from an area of mathematics which may be unfamiliar to most readers. The author is convinced that the concepts of simulation and of running a program-algebra on a machine-algebra are essentially category-theoretic notions (an insight which he owes to [deB 69] and which also appears in somewhat disguised form in [Burs 72]) and believes that recognition of this fact will make them better understandable. All the same, the reader who prefers to skip Chapter II, from Definition II.2 onwards, and to ignore all mention of categories and functors thereafter should find all the results of later chapters stated in "plain language" and, except in Chapter VI, should still be able to follow the proofs.

Notation

A great deal of the notation used in what follows is adapted from MacLane and Birkhoff, <u>Algebra</u> [MacL 67].

Sets: Upper-case Roman and Greek letters are used for names of sets. For set constants, we use curly brackets and commas to give a set in extension, as $\{1,2,3\}$, and curly brackets with a vertical bar separating bound variable from characteristic property to give a set in intension, as $\{x \mid 0 < x < 4\}$.

Operations on sets: The signs $\cap$, $\cup$, $-$, and $\subseteq$ denote set intersection, union, difference, and inclusion, respectively. We also use $\cap$, $\cup$, $\subseteq$ between relations of the same type, meaning that the set operation is to be performed on their graphs. Set membership is written $\in$. Cartesian product and disjoint union of sets are denoted respectively by $\times$ and $\dot{\cup}$.

Logical operations: Within formulas, we denote the usual truth-functional operations by the underlined words and, or, not, implies, iff, quantification by for all, for some, and truth values by true, false.

Functions: By either of $f: A \to B$ or $A \xrightarrow{f} B$ we express that f is a function with domain A and codomain B. Note that two functions are to be considered as equal if and only if they have the same domain, same codomain, and same graph.

We also use the notations $f: A \to B$ and $A \xrightarrow{f} B$ when A and B are objects in any category and f is a morphism between them (a situation which includes the ordinary notion of a homomorphism between operator algebras).

Partial functions: By $f: A \stackrel{\sim}{\to} B$ we express that f is a partial function with domain A and codomain B.

Application: If f is a function or partial function, we denote the result of its application to the argument x by $f(x)$, f_x, or even, if no confusion can arise, by simply $f\ x$. (However, subscripts, together with primes, tildes, and the like, are also sometimes used merely as distinguishing marks.)

Relations: By $f: A \twoheadrightarrow B$, we express that f is a binary relation on A to B, i.e., that it has domain A, codomain B, and graph any subset of the Cartesian product $A \times B$.

We may also use the notation $(A \to B)$ (similarly $(A \tilde{\to} B)$, $(A \twoheadrightarrow B)$) in isolation to denote the set of all functions (similarly, partial functions, relations) on A to B. It follows that the sign $:$, as used so far (but beware, not subsequently) could be displaced in favor of the sign ϵ.

We consider that $(A \to B) \subseteq (A \tilde{\to} B) \subseteq (A \twoheadrightarrow B)$. The notation described in the following paragraphs as being for relations therefore applies equally to partial functions and functions.

If $f: A \twoheadrightarrow B$, $a \epsilon A$, $b \epsilon B$, then we express by

$$f: a \mapsto b$$

that f relates a to b, i.e., that the pair $\langle a,b \rangle$ is an element of the graph of f, or what can be also expressed, if f is at least a partial function, by $f(a) = b$.

Composition: If $f: X \twoheadrightarrow Y$, $g: Y \twoheadrightarrow Z$, then we denote by either $g \circ f$ or $f;g$ indifferently the composite defined by

$$f;g : X \twoheadrightarrow Z$$

with

$$f;g : x \mapsto z \text{ \underline{iff} \underline{for} \underline{some} } y \in Y,\ f : x \mapsto y \text{ \underline{and} } g : y \mapsto z \ .$$

If f and g are partial functions we have

$$g \circ f(x) = g(f(x)) = (f;g)(x) \ .$$

Iteration and converse: If $f: A \nrightarrow A$, then f^k denotes the k-fold composition of f with itself. This notation is consistently extended to all integral powers by taking f^0 to be the identity relation on A , and f^{-k} to be the relational converse of f^k , i.e., $f^{-k}: a' \mapsto a$ iff $f^k: a \mapsto a'$. The notation f^{-1} to denote the converse of f is also used for relations with arbitrary (possibly different) domain and codomain.

Insertions: With every instance of set inclusion, such as $A \subseteq B$, there is associated a unique function $i: A \to B$ with $i: a \mapsto a$; this is called the insertion of A into B . We may name the insertion in passing by writing, e.g., $i: A \subseteq B$.

Restriction: If $i: A \subseteq B$, and $f: B \nrightarrow C$, we denote by $f \upharpoonright A$ the composite $f \circ i$, and call this the restriction of f to A . Conversely, if $D \subseteq C$, we denote $(f^{-1} \upharpoonright D)^{-1}$ more briefly by $f \downharpoonright D$, and call this the cut-down of f to D . Moreover we shall write $f \rceil D$ as shorthand for $(f \upharpoonright D) \downharpoonright D$; when no confusion can arise we also refer to this combined restriction and cut-down simply as restriction.

Special sets and functions:

$\mathbb{N}$ denotes the natural numbers $\{0,1,2,\ldots\}$.

$\underline{k}$ denotes a standard set of k elements, namely $\{0,1,\ldots,k-1\}$; in particular, $\underline{1}$ denotes $\{0\}$. Exception: at some places the

explicit convention will be made that $\underline{2}$ denotes the set $\{\underline{true}, \underline{false}\}$.

The identity function (equality relation) on a set A to itself is denoted by 1_A .

Other alphabets: Underlined Roman capitals (as, $\underline{A}$) will be used for general operator algebras; underlined curly capitals (as, $\underline{\mathcal{A}}$) for binary relational algebras (BRAs). Plain curly capitals will be used for categories and functors (as, $\mathcal{A}$), except for the following five special functors (defined in Chapters II and III): $\otimes$, $\oplus$, Σ , $\mathcal{A}$nc , ac .

Functions and homomorphisms will be denoted by lower-case letters, generally Roman for the former and Greek for the latter. Note, however, that our definition of an algebra will make it a function from its set of operators to its set of operations, so that an applicative notation such as for example $\underline{A}_\omega$ will indicate the operation which the operator ω denotes in $\underline{A}$.

Unanalyzed set elements will be named by any convenient symbols, for example x , S , ω , + .

Organization

The first half of the thesis, Chapters I-VII, is theory; the second half is application.

In particular, Chapters I-III develop the theory of binary relational algebras as models for the concepts of program, machine, and computation. Chapters IV, V, and VI introduce the concept of operator algebra as a

common model for a programming language and its set of meanings, and present examples of modelling both programming language semantics and compilers (in Chapters IV and V respectively) by homomorphisms of operator algebras. Chapter VI assimilates the notion of the meaning of a program obtained by regarding it as a BRA to the notion of semantic homomorphism. In Chapter VII a theorem is proved which will allow a correct compiler for a complex language to be assembled from compilers for simple languages each of which embodies only a single "feature".

The second half of the thesis, Chapters VIII - XII, is devoted to exemplary proofs of compiler correctness. Chapter VIII may be considered a second introduction; it sets out the method to be followed in the applications. There are three proofs in Chapter IX, one each in X and XI -- all these for simple, one-feature languages -- , and in Chapter XII the theorem from VII is applied to obtain a correctness proof for a (somewhat) complex language.

The languages considered in Chapter IX are referred to as Examples SS, AE, and BE; they will already have been introduced under these names, and their semantics and compilers defined, to illustrate the development in Chapters IV and V.

The following is a suggested strategy for a first reading of the thesis: Skim Chapters I - VII very lightly, attending only to the informal exposition and the examples. Read Chapter VIII as soon as possible; then study one or more of the proofs in Chapters IX - XI, referring back to Chapters I - VI for explanations of concepts as the need for them becomes apparent. Defer Chapters VII and XII until last.

I. Modelling Computing Devices and Computations by Binary Relational Algebras

The present chapter will give the basic definitions, with examples, of the algebraic model of computation due to Landin [Land 70].

Definition I.1: A Binary Relational Algebra (BRA) $\mathcal{A}$ is a function $\mathcal{A}: \Gamma \to (|\mathcal{A}| \nrightarrow |\mathcal{A}|)$ associating with each element (operator) of a set Γ a binary relation (operation) on a set $|\mathcal{A}|$ (the carrier of $\mathcal{A}$).

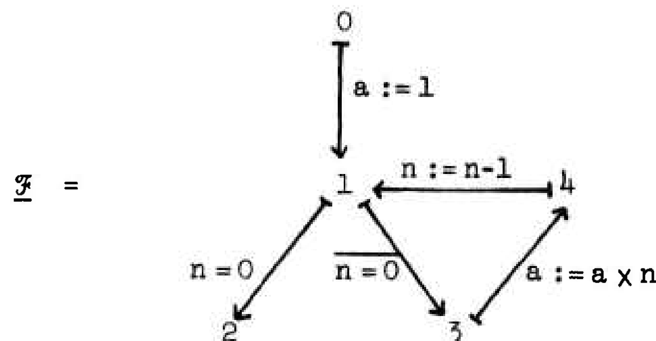
To get hold of the right end of the stick, one may do well to keep in mind from the start the informal notion of a computation of $\mathcal{A}$ as being a sequence (finite or infinite) $a_0 \underset{\gamma_0}{\mapsto} a_1 \underset{\gamma_1}{\mapsto} a_2 \vdash \cdots$, such that $\mathcal{A}_{\gamma_i}: a_i \mapsto a_{i+1}$.

Example I.1: A finite-state machine (not that the restriction to finitude has any particular significance for us) is normally taken to be a function $M: I \times Q \to Q$ with Q the set of states and I the alphabet. We can just as well view this as a BRA $\mathcal{M}: I \to (Q \to Q)$. The relations here are special in two ways: by being (partial) functions, and by being total. We distinguish the first of these special properties by its own name: a unary function algebra (UFA) is a BRA of which all the relations are partial functions.

Example I.2: A flowchart is ordinarily written as a graph with labeled nodes: some nodes (function boxes) with one exit edge; others (predicate boxes) with two labeled exits; others still (halts) with none. We wish,

following a device of Karp's [Karp 59], to keep the same shaped graph, but transfer the labels from the nodes to their outgoing edges (from each predicate π we must somehow derive two distinct labels, say π and $\bar{\pi}$, for the two departing edges). We then have a BRA, with the set of nodes as carrier, mapping each label to the set of ordered pairs of nodes connected by a so-labelled edge. It will by its construction observe the rule that each node is related to at most two others, and if to as many as two, then by virtue of a pair of complementary predicate labels.

There is, as observed by Landin [Land 70], a one-to-one correspondence between BRAs and edge-labelled directed graphs without duplicate edges. In fact, by taking some liberties with the "barred arrow" notation by which $x \overset{f}{\mapsto} y$ indicates that $y = f(x)$, we can claim edge-labelled directed graphs, with the edges drawn as barred arrows, as ready-made notation for presenting small finite BRAs. E.g., if we write:



we indicate that $\underline{\mathfrak{F}}$ is a BRA with carrier $\underline{5}$, and with at least the set of operators $\{n = 0,\ a := 1,\ \overline{n = 0},\ a := a \times n,\ n := n-1\}$ each denoting a relation : $\underline{5} \nrightarrow \underline{5}$ -- as it happens each of these operations relates only one pair of carrier elements -- and possibly with other operators as well, all denoting the empty relation.

Running one BRA on another:

Given BRAs which in some way model a program and a machine, one would expect them to determine the computations obtainable by running the program on the machine. We shall make the seemingly over-general definition of an operation $\otimes$ for producing from two BRAs a third which, as will be seen, has a claim to be called their Cartesian product.

We first define the Cartesian product of two relations: if $R: A \twoheadrightarrow C$, $S: B \twoheadrightarrow D$, then $R \times S: a,b \twoheadrightarrow c,d$ iff $R: a \mapsto c$ and $S: b \mapsto d$. I.e., if we consider the graphs (denoted here by an over-bar) of the relations: $\bar{R} = \{\langle a,c\rangle \mid R: a \mapsto c\}$; $\bar{S} = \{\langle b,d\rangle \mid S: b \mapsto d\}$, then the graph of $R \times S$ is just the set-theoretic Cartesian product of the two graphs, with each component quadruple rearranged: whereas

$\bar{R} \times \bar{S} = \{\langle\langle a,c\rangle,\langle b,d\rangle\rangle \mid \langle a,c\rangle \in \bar{R}$ and $\langle b,d\rangle \in \bar{S}\}$,

$\overline{R \times S} = \{\langle\langle a,b\rangle,\langle c,d\rangle\rangle \mid \langle a,c\rangle \in \bar{R}$ and $\langle b,d\rangle \in \bar{S}\}$.

We are now ready to define a product on BRA's:

Definition I.2: If $\mathcal{A}$ and $\mathcal{B}$ are each BRAs with operator set Γ , then $\mathcal{A} \otimes \mathcal{B}: \Gamma \to (|\mathcal{A}| \times |\mathcal{B}| \overset{\bullet}{\twoheadrightarrow} |\mathcal{A}| \times |\mathcal{B}|)$ is given by $(\mathcal{A} \otimes \mathcal{B})_\gamma = \mathcal{A}_\gamma \times \mathcal{B}_\gamma$, for $\gamma \in \Gamma$.

Intuitively, the computations of $\mathcal{A} \otimes \mathcal{B}$ are just those common to $\mathcal{A}$ and $\mathcal{B}$: the product can do whatever both its factors can do.

Example I.3: Finite state machines as above; take an input sequence, e.g., the string pqr , to be an UFA $\bullet \overset{p}{\mapsto} \bullet \overset{q}{\mapsto} \bullet \overset{r}{\mapsto} \bullet$ (we use dots to indicate arbitrary, distinct carrier elements when there is no occasion to name them); then the product of a string with a machine gives all

the computation sequences generated by presenting the string to the various states of the machine.

Example I.4: Flowcharts and their interpretations. An interpretation I is an assignment of unary functions and predicates over a domain (of states) to function and predicate labels; take an UFA $\underline{\mathcal{J}}$ with the domain as carrier, $\underline{\mathcal{J}}: f \mapsto I_f$; $\underline{\mathcal{J}}: p \rightarrow \{x \mapsto x \mid I_p(x)\}$; $\mathcal{J}: \bar{p} \rightarrow \{x \mapsto x \mid \underline{\text{not}}\ I_p(x)\}$. Then if $\underline{\mathcal{P}}$ is a flowchart UFA, as above, the product $\underline{\mathcal{P}} \otimes \underline{\mathcal{J}}$ relates each $\langle$node, state$\rangle$ pair to its successor pair.

The products in Examples 3 and 4 have a property stronger than being UFA's: each element of the carrier is related to at most one other element, even considering all the relations together. We may make the definitions:

Definition I.3: $\Sigma \underline{\mathcal{A}} = \bigcup_{\gamma \in \Gamma} \underline{\mathcal{A}}_\gamma$, for any BRA $\underline{\mathcal{A}}$ with operator set Γ .

Definition I.4: A BRA $\underline{\mathcal{A}}$ is monogenic iff $\Sigma \underline{\mathcal{A}}$ is a partial function, i.e., iff $\Sigma \underline{\mathcal{A}} \in (|\underline{\mathcal{A}}| \rightharpoonup |\underline{\mathcal{A}}|)$.

Generally speaking, the idea of deterministic computation will be modelled by monogenic BRAs; however, most of the theory will apply equally to mono- and poly-genic BRAs, and so may throw some light on (one notion of) non-deterministic computation.

Example I.5: Turing Machines. Define $\underline{\mathcal{T}}^F$, the Turing machine-BRA with alphabet F , to be an UFA, $\underline{\mathcal{T}}^F: F \times F \times \{\underline{\text{left}}, \underline{\text{right}}\} \rightarrow (\text{Tape} \rightharpoonup \text{Tape})$,

where Tape $= (\mathbb{N} \rightharpoonup F) \times F \times (\mathbb{N} \to F)$, as follows:

$$\underline{T}^F_{f,g,\underline{left}}: \alpha, f, \beta \mapsto \alpha \circ \sigma, \alpha(0), \beta \circ \rho \cup \{0 \mapsto g\}$$

$$\underline{T}^F_{f,g,\underline{right}}: \alpha, f, \beta \mapsto \alpha \circ \rho \cup \{0 \mapsto g\}, \beta(0), \beta \circ \sigma$$

where

$$\sigma: \mathbb{N} \to \mathbb{N}: n \mapsto n+1$$

and

$$\rho: \mathbb{N} \rightrightarrows \mathbb{N}: n+1 \mapsto n \quad .$$

A (nondeterministic) program $\underline{P}^F$ for $\underline{T}^F$ is an arbitrary finite UFA $\underline{P}^F: F \times F \times \{\underline{left}, \underline{right}\} \to (Q \rightrightarrows Q)$, with Q a finite set of states.

A product $\underline{P}^F \otimes \underline{T}^F$ is then a particular non-deterministic Turing machine with alphabet F as ordinarily defined.

II. Simulation and Categories

In keeping with our idea that a BRA defines the class of computation sequences it can perform, we want a notion that of two BRAs $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{B}$ simulates $\mathcal{A}$ if $\mathcal{B}$ can perform any computation that $\mathcal{A}$ can. Formally we define:

Definition II.1: A simulation of a BRA $\mathcal{A}$ by a BRA $\mathcal{B}$ with common set of operators Γ is a function $\varphi\colon |\mathcal{A}| \to |\mathcal{B}|$ with the property, for all $\gamma\epsilon\Gamma$,

$$\mathcal{A}_\gamma;\varphi \subseteq \varphi;\mathcal{B}_\gamma$$

or equivalently,

$$\varphi^{-1};\mathcal{A}_\gamma;\varphi \subseteq \mathcal{B}_\gamma$$

or equivalently,

$$\mathcal{A}_\gamma \subseteq \varphi;\mathcal{B}_\gamma;\varphi^{-1}$$

(these equivalences because φ is a function, and so we have $\varphi^{-1};\varphi \subseteq 1_{|\mathcal{B}|}$ and $1_{|\mathcal{A}|} \subseteq \varphi;\varphi^{-1}$.

We hasten to observe that there are some very silly simulations, e.g., the BRA $\underline{1}^\Gamma$, with

$$|\underline{1}^\Gamma| = \{0\} \quad , \quad \underline{1}^\Gamma_\gamma\colon 0 \mapsto 0$$

simulates all BRAs with operator set Γ.

Depending on the particular application, we shall generally want to prove more than just simulation, for example, that φ is invertible and $\mathcal{B}$ is monogenic, before we think we have a simulation in the intuitive, useful sense that the simulating object will really "do the work" of the simulated one.

Categories:

We next introduce a few of the notions of category theory, in order to facilitate the development of the theory of BRAs and simulations.

Definition II.2: A concrete category $\mathcal{C}$ is a class of objects $Obj(\mathcal{C})$ together with two maps: one, $U_{\mathcal{C}}$, associating with each $X \in Obj(\mathcal{C})$ a set (the underlying set, or carrier) $U_{\mathcal{C}}(X)$, and a second, $Mor_{\mathcal{C}}$, associating with each pair of objects X,Y of $\mathcal{C}$ a set of functions (the morphisms) with domain $U_{\mathcal{C}}(X)$ and codomain $U_{\mathcal{C}}(Y)$, (briefly, $Mor_{\mathcal{C}}(X,Y) \subseteq (U_{\mathcal{C}}(X) \to U_{\mathcal{C}}(Y))$), and satisfying the following two axioms:

CC1: For all $X \in Obj(\mathcal{C})$, $1_{U(X)} \in Mor_{\mathcal{C}}(X,X)$,

CC2: For all $X,Y,Z \in Obj(\mathcal{C})$, if $\varphi \in Mor_{\mathcal{C}}(X,Y)$ and $\psi \in Mor_{\mathcal{C}}(Y,Z)$ then $\psi \circ \varphi \in Mor_{\mathcal{C}}(X,Z)$.

We extend the ordinary notation for functionality by writing $\varphi: X \to Y$ to express that $\varphi \in Mor_{\mathcal{C}}(X,Y)$, and we denote $1_{U(X)}$ by 1_X. The simplest example of a category is, of course, the category $\mathfrak{Ens}$ of sets, where U is simply the identity function, and $Mor(X,Y)$ is the set of all functions from X to Y. Another easy example is the category $\mathfrak{Reln}$ of binary relations on sets to themselves. If $R: A \nrightarrow A$ is such a relation, we take $U_{\mathfrak{Reln}}(R) = A$, and $Mor_{\mathfrak{Reln}}(R,S)$ (supposing $U(S) = B$) to be the set of all functions $\varphi: A \to B$ satisfying $R;\varphi \subseteq \varphi;S$ (or equivalently, since φ is a function, satisfying either of $\varphi^{-1};R;\varphi \subseteq S$ or $R \subseteq \varphi;S;\varphi^{-1}$).

The axioms for a concrete category are easily verified for $\mathfrak{Reln}$.

Our aim is to discover that the BRAs with any given set Γ of operators form a category, $\mathcal{B}ra^{\Gamma}$ with simulations as morphisms. However, to facilitate the subsequent development, we shall allow this fact to emerge as a special case of the following general construction for building new categories from old:

Proposition II.1. If $\mathcal{C}$ is a concrete category, and S is a set, then $\mathcal{C}^{[S]}$ is a concrete category, where $\mathcal{C}^{[S]}$ is defined as follows: The objects of $\mathcal{C}^{[S]}$ are functions $\underline{A}: S \to \mathrm{Obj}(\mathcal{C})$, with the special property that for all $s,t \in S$, $U(\underline{A}(s)) = U(\underline{A}(t)) =$ (by definition) $U(\underline{A})$; i.e., an admissible function has for all the objects in its range of values the same underlying set. The morphisms of $\mathcal{C}^{[S]}$ between two objects $\underline{A}$ and $\underline{B}$ are all those functions $\varphi: U(\underline{A}) \to U(\underline{B})$ with the property that for all $s \in S$, $E \in \mathrm{Mor}_{\mathcal{C}}(\underline{A}(s),\underline{B}(s))$. $\mathcal{C}^{[S]}$ is readily verified to be a category.

The category $\mathcal{B}ra^{\Gamma}$ of all BRAs with operator set Γ, and with simulations as morphisms, is now seen to be exactly the category $\mathcal{R}eln^{[\Gamma]}$.

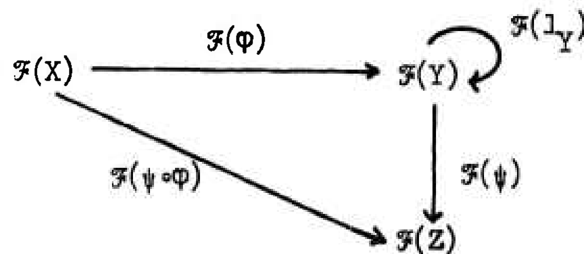
We next define the important notion of a functor, which is essentially a homomorphism of categories.

Definition II.3: A (covariant) functor $\mathcal{F}: \mathcal{C} \to \mathcal{D}$ from a category $\mathcal{C}$ to a category $\mathcal{D}$ consists of two mappings (the same letter $\mathcal{F}$ is by convention used for both), one (called the object function of $\mathcal{F}$) giving for each $X \in \mathrm{Obj}(\mathcal{C})$ an $\mathcal{F}(X) \in \mathrm{Obj}(\mathcal{D})$, and the other (the mapping function) giving for each $\varphi: X \to Y$ an $\mathcal{F}(\varphi): \mathcal{F}(X) \to \mathcal{F}(Y)$, such that the following two axioms are satisfied:

F1: $\mathcal{F}(1_X) = 1_{\mathcal{F}(X)}$

F2: $\mathcal{F}(\psi \circ \varphi) = \mathcal{F}(\psi) \circ \mathcal{F}(\varphi)$.

Pictorially, the axioms require that any instance of the following diagram should commute:



The following easily verified proposition will provide a convenient method for obtaining functors from a category of BRAs to itself.

Proposition II.2: Any functor $\mathcal{F}: \mathcal{C} \to \mathcal{D}$ yields a functor $\mathcal{F}^{[S]}: \mathcal{C}^{[S]} \to \mathcal{D}^{[S]}$ given by $\mathcal{F}^{[S]}(\underline{X})(s) = \mathcal{F}(\underline{X}(s))$ and $\mathcal{F}^{[S]}(\varphi) = \mathcal{F}(\varphi)$.

We shall want functors of more than one argument, and for this purpose we introduce the Cartesian product of categories. Intuitively (and even formally in the theory of abstract categories, which bear just the same relation to concrete categories as abstract groups do to permutation groups) the product $\mathcal{C} \times \mathcal{D}$ of categories $\mathcal{C}$ and $\mathcal{D}$ has for objects ordered pairs $\langle C,D \rangle$ of an object C from $\mathcal{C}$ and an object D from $\mathcal{D}$, and for morphisms ordered pairs $\langle \varphi,\psi \rangle: \langle C,D \rangle \to \langle C',D' \rangle$, where $\varphi: C \to C'$ and $\psi: D \to D'$, and with the obvious component-wise composition. To get an isomorphic concrete category, we must fiddle with this construction slightly; it is good enough to take the disjoint union

$U(C) \dot{\cup} U(D)$ for $U(\langle C,D\rangle)$, and if $U(C) \xrightarrow{i} U(\langle C,D\rangle) \xleftarrow{j} U(D)$ are the injections into the disjoint union, i', j' similarly for $\langle C',D'\rangle$, then instead of the ordered pair of functions $\langle \varphi,\psi\rangle$, we may take as a substitute concrete morphism that function in $(U(\langle C,D\rangle) \to U(\langle C',D'\rangle))$ with graph $(i^{-1};\varphi;i') \cup (j^{-1};\psi;j')$.

A <u>bifunctor</u>, or functor of two arguments is now an ordinary functor defined on a Cartesian product category.

It is easy to check the following distributive law:

<u>Proposition II.3</u>: $(\mathcal{C} \times \mathcal{D})^{[S]} \cong \mathcal{C}^{[S]} \times \mathcal{D}^{[S]}$.

(To be pedantic, isomorphism $(\cong)$ of two categories may be defined as the existence of an invertible functor between them.) From this we have immediately that the statement analogous to Proposition II.2 is valid for bifunctors.

We shall have occasion to define a number of functors on concrete categories. Most of them will be related to functors on the category of sets, in such a way that the labor of proving them functors can be greatly reduced by the use of the following proposition:

<u>Proposition II.4</u>: Let $\mathcal{F}: \mathcal{E}ns \to \mathcal{E}ns$ be a functor on the category of sets to itself. Let $\mathcal{C}$, $\mathcal{D}$ be concrete categories, and suppose $\mathcal{S}$ is a function: $\mathrm{Obj}(\mathcal{C}) \to \mathrm{Obj}(\mathcal{D})$ with the properties

(i) $U(\mathcal{S}(X)) = \mathcal{F}(U(X))$

(ii) For $\varphi: X \to Y$ in $\mathcal{C}$, $\mathcal{F}(\varphi): \mathcal{S}(X) \to \mathcal{S}(Y)$ in $\mathcal{D}$.

Then $\mathcal{S}$ (with the mapping function of $\mathcal{F}$ as mapping function) is a functor on $\mathcal{C}$ to $\mathcal{D}$; we choose to call $\mathcal{S}$ a <u>specialization</u> of $\mathcal{F}$.

Proof: (ii), with the definition of $\mathscr{B}$'s mapping function, gives us that $\mathscr{B}(\varphi): \mathscr{B}(X) \to \mathscr{B}(Y)$ whenever $\varphi: X \to Y$. So we have only to check

(a) $\mathscr{B}(1_X) = 1_{\mathscr{B}(X)}$ and

(b) $\mathscr{B}(\psi \circ \varphi) = \mathscr{B}(\psi) \circ \mathscr{B}(\varphi)$.

But $\mathscr{B}(1_X) = \mathscr{F}(1_{U(X)}) = 1_{\mathscr{F}(U(X))} = 1_{U(\mathscr{B}(X))} = 1_{\mathscr{B}(X)}$, and $\mathscr{B}(\psi \circ \varphi) = \mathscr{F}(\psi \circ \varphi) = \mathscr{F}(\psi) \circ \mathscr{F}(\varphi) = \mathscr{B}(\psi) \circ \mathscr{B}(\varphi)$. ∎

For a first application of Proposition II.4, we define the bifunctor $\otimes : \mathfrak{Reln} \times \mathfrak{Reln} \to \mathfrak{Reln}$ as a specialization of the Cartesian product functor on sets. (Following [MacL 67], the mapping function of the Cartesian product functor $\times: \mathfrak{Ens} \times \mathfrak{Ens} \to \mathfrak{Ens}$ is given by: If $\varphi: A \to A'$, $\psi: B \to B'$, then $\varphi \times \psi: A \times B \to A' \times B': \langle a,b \rangle \to \langle \varphi a, \psi b \rangle$.) We have, then, only to specify $R \otimes S: U(R) \times U(S) \twoheadrightarrow U(R) \times U(S)$. We shall simply generalize the definition of the Cartesian product <u>mapping</u> function to read for relations (rather than exclusively for functions):

<u>Definition II.4</u>: If $R: A \twoheadrightarrow A'$, $S: B \twoheadrightarrow B'$, then $R \times S: A \times B \twoheadrightarrow A' \times B' = \{\langle a,b \rangle \mapsto \langle a',b' \rangle \mid R: a \mapsto a' \text{ and } S: b \mapsto b'\}$,

and then use this to define the <u>object</u> function of our bifunctor $\otimes$ on $\mathfrak{Reln}$, namely:

<u>Definition II.5</u>: $R \otimes S =_{df} R \times S$.

We have now to verify that if $\varphi: R \to R'$, $\psi: S \to S'$, then $\varphi \otimes \psi (=_{df} \varphi \times \psi): R \otimes S \to R' \otimes S'$. That is, given that $R;\varphi \subseteq \varphi;R'$, $S;\psi \subseteq \psi;S'$, we must show $(R \times S);(\varphi \times \psi) \subseteq (\varphi \times \psi);(R' \times S')$. But, as is easily seen, $(R \times S);(\varphi \times \psi) = (R;\varphi) \times (S;\psi)$ and similarly for the right-hand side, and moreover $\times$ is monotone for $\subseteq$, hence finally $R \times S;\varphi \times \psi = (R;\varphi) \times (S;\psi) \subseteq (\varphi;R') \times (\psi;S') = \varphi \times \psi;R' \times S'$. ∎

The functor $\otimes^{[\Gamma]}$, which we also write $\otimes$, may be seen to be just the operation of running one BRA on another.

A second bifunctor we shall be wanting is a specialization of the disjoint union of sets; we shall write it $\oplus$. Supposing $i: U(R) \to U(R) \dot{\cup} U(S)$ and $j: U(S) \to U(R) \dot{\cup} U(S)$ are the injections which copy $U(R)$ and $U(S)$ into their disjoint union, then we specify $R \oplus S: U(R) \dot{\cup} U(S) \nrightarrow U(R) \dot{\cup} U(S)$ by

<u>Definition II.5</u>: $\quad R \oplus S = i^{-1};R;i \cup j^{-1};S;j$.

The proof that if $\varphi: R \to R'$ and $\psi: S \to S'$ then $\varphi \dot{\cup} \psi: R \oplus S \to R' \oplus S'$ and that therefore $\oplus$ is a bifunctor on $\mathfrak{Reln}$ (and so by Proposition II.2 on $\mathcal{B}ra^{\Gamma}$) is dual to that given for $\otimes$.

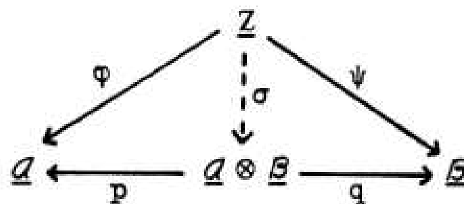
We may observe that $\mathcal{B}ra^{\Gamma}$ has an <u>initial object</u>, i.e., one simulated in a unique way by every object, namely the BRA $\underline{0}$ whose underlying set is empty. Equally, there is a <u>terminal object</u>, one which simulates every object in a unique way; it is the BRA $\underline{1}$ whose underlying set is a singleton, say $\{0\}$, and which maps every operator of Γ to the universal (and also identity) relation $\{0 \mapsto 0\}$.

We may also observe that there are natural isomorphisms for any BRA $\underline{\mathcal{A}}$:

$$\underline{\mathcal{A}} \cong \underline{\mathcal{A}} \otimes \underline{1} \quad , \quad \underline{\mathcal{A}} \cong \underline{\mathcal{A}} \oplus \underline{0} \quad , \quad \underline{0} \cong \underline{\mathcal{A}} \otimes \underline{0} \quad ,$$

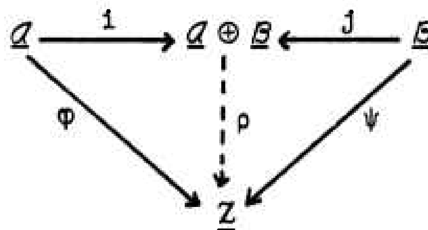
and that up to natural isomorphism, $\oplus$ and $\otimes$ are commutative and associative operations, with $\otimes$ distributive over $\oplus$.

It is now easy to show that the functors $\otimes$ and $\oplus$ have distinguished (and dual) properties; namely $\underline{\mathcal{A}} \otimes \underline{\mathcal{B}}$ provides a <u>product object</u>, and $\underline{\mathcal{A}} \oplus \underline{\mathcal{B}}$ a <u>coproduct object</u>, in the sense of category theory. The property, that $\underline{\mathcal{A}} \otimes \underline{\mathcal{B}}$ is a product object, may be simply stated as follows: any diagram



where p and q are the projection functions $p: \langle a,b \rangle \mapsto a$, $q: \langle a,b \rangle \mapsto b$, can be filled in by a unique morphism at σ so as to commute. Since we are working with concrete categories, most of this is already proved for us; it is an elementary result (see, e.g. [MacL 67]) that the Cartesian product is the categorical product in the category of sets, that is we know the function $\sigma: x \mapsto \langle \varphi x, \psi x \rangle$ is the one and only way to make the diagram commute as a diagram of functions. All we have to check is that the functions p , q , and σ actually are morphisms (i.e., simulations) in $\mathcal{G}ra^{\Gamma}$. This is not hard, using that $\otimes$ is a functor. Take $1_{\underline{\mathcal{B}}}: \underline{\mathcal{B}} \to \underline{\mathcal{B}}$, $t: \underline{\mathcal{A}} \to \underline{1}$, and $i: \underline{\mathcal{B}} \otimes \underline{1} \cong \underline{\mathcal{B}}$. ($1_{\underline{\mathcal{B}}}: b \mapsto b$, $t: a \mapsto 0$, and $i: \langle b,0 \rangle \mapsto b$.) Then $q = i \circ (t \otimes 1_{\underline{\mathcal{B}}})$; hence $q: \underline{\mathcal{A}} \otimes \underline{\mathcal{B}} \to \underline{\mathcal{B}}$. Similarly, $p: \underline{\mathcal{A}} \otimes \underline{\mathcal{B}} \to \underline{\mathcal{A}}$. To establish σ as a morphism, consider $\Delta: \underline{Z} \to \underline{Z} \otimes \underline{Z}$, $\Delta: z \mapsto \langle z,z \rangle$ which is readily seen to be a morphism. Then $\sigma = (\varphi \otimes \psi) \circ \Delta$; hence $\sigma: \underline{Z} \to \underline{\mathcal{A}} \otimes \underline{\mathcal{B}}$.

The proof that $\underline{\mathcal{A}} \oplus \underline{\mathcal{B}}$ gives a coproduct object, that is that the diagram



can be filled in uniquely at ρ so as to commute (where i and j are the injections into the disjoint union) is dual to that for $\underline{\mathcal{A}} \otimes \underline{\mathcal{B}}$ given above.

The product and coproduct properties have an obvious potential for proofs of the existence of a simulation of one BRA by another: if we want to show that a product $\underline{\mathcal{A}} \otimes \underline{\mathcal{B}}$ simulates a BRA $\underline{Z}$, we need only find simulations of $\underline{Z}$ by the factors of $\underline{\mathcal{A}} \otimes \underline{\mathcal{B}}$ separately; this might be easier, since $\underline{\mathcal{A}}$ and $\underline{\mathcal{B}}$ individually will each be less deterministic than their product. Dually, to simulate a coproduct BRA, it is enough to find a BRA which simulates both of its cofactors.

We shall here parenthetically indicate the connection between our notion of BRA and Burstall's [Burs 72] category-theoretic model of programs.

It turns out that there are two fairly natural ways of regarding an individual BRA as a functor, although these entail departing from the convention which we observe elsewhere of dealing only with concrete categories. First of all, corresponding to any alphabet Γ, there is a category $\mathcal{J}$ with one object (say 0) such that $\mathcal{J}$ is isomorphic to Γ^*, the free monoid with generating set Γ. The morphisms of $\mathcal{J}$ (necessarily all $: 0 \to 0$) correspond to the elements of Γ^*, that is, to the finite strings of elements of Γ; composition of morphisms corresponds to concatenation of strings. Also, there is a category $\mathcal{E}$nsrel with its objects all sets, but with relations for its morphisms (rather than functions, as with $\mathcal{E}$ns); composition of morphisms is relational composition. Now it may be seen that to any Γ-BRA $\underline{\mathcal{B}}$ there corresponds bi-uniquely a functor $\underline{\mathcal{B}}^+ : \mathcal{J} \to \mathcal{E}$nsrel, with object function

given by $\underline{\mathcal{B}}^+ : 0 \mapsto |\underline{\mathcal{B}}|$, and with mapping function uniquely determined by the requirement that $\underline{\mathcal{B}}^+ : \gamma \mapsto \underline{\mathcal{B}}_\gamma$ for $\gamma \in \Gamma$.

On the other hand, any unlabelled directed graph H can be made to correspond bi-uniquely to a category $\mathcal{H}$ whose objects are the nodes of H , and whose morphisms are the finite paths in H . A Γ-BRA $\underline{\mathcal{B}}$ may be regarded as a labelling of the edges of some directed graph H with labels from Γ ; hence $\underline{\mathcal{B}}$ may equally well be regarded as a functor $\underline{\mathcal{B}}^- : \mathcal{H} \to \mathcal{S}$, i.e., as a labelling of the paths in H with words from Γ^* .

Our method, when we have a program-BRA $\underline{\mathcal{A}}$ interpreted by (running on) a machine-BRA $\underline{\mathcal{B}}$, will be to model the situation by forming the product $\underline{\mathcal{A}} \otimes \underline{\mathcal{B}}$. Burstall, less symmetrically, models the interpreted program by the composite functor $\underline{\mathcal{B}}^+ \circ \underline{\mathcal{A}}^-$.

III. The Relation Computed by a BRA

Informally, what we would like to say is that the relation computed by a BRA is (some suitable restriction of) its accessibility relation, which relates two elements of the BRA's carrier just in case the second can be reached from the first in some finite number of steps, by the use of any of the operations. To develop this notion formally, we first observe that the "fuse operators" function Σ defined above (Definition I.3) is a functor $\Sigma\colon \mathcal{B}ra^{\Gamma} \to \mathcal{R}eln$, indeed a specialization of the identity functor. To verify this, we need only check that

$$\underline{\text{for all}}\ \gamma\in\Gamma\ ,\quad \mathcal{A}_{\gamma};\varphi \subseteq \varphi;\mathcal{B}_{\gamma}$$

implies

$$\Big(\bigcup_{\gamma\in\Gamma} \mathcal{A}_{\gamma}\Big);\varphi \subseteq \varphi;\Big(\bigcup_{\gamma\in\Gamma} \mathcal{B}_{\gamma}\Big)\ ,$$

which it does, because relational composition distributes over union, and union is monotone with respect to $\subseteq$.

We next define a functor $\mathcal{A}nc\colon \mathcal{R}eln \to \mathcal{R}eln$ (for ancestral), a specialization of the identity functor, by:

Definition III.1: $\mathcal{A}nc(R) = \bigcup_{i\geq 0} R^{(i)}$.

To show that $\mathcal{A}nc$ is a functor, suppose $\varphi\colon Q \to R$ in $\mathcal{R}eln$ with $Q\colon A \nrightarrow A$, $R\colon B \nrightarrow B$; i.e., $Q \subseteq \varphi;R;\varphi^{-1}$. Then for each $k \geq 0$, we have $Q^{(k)} \subseteq (\varphi;R;\varphi^{-1})^{(k)}$ by the monotonicity of composition for $\subseteq$ and because, φ being a function, for $k = 0$ we have $1_A \subseteq \varphi;1_B;\varphi^{-1}$.

But since φ is a function, also $\varphi^{-1};\varphi \subseteq 1_B$, hence $Q^{(k)} \subseteq \varphi;R^{(k)};\varphi^{-1}$. So by the definition of $\mathcal{A}nc$, and the monotonicity of $\bigcup$ for $\subseteq$, we have $\mathcal{A}nc(Q) \subseteq \varphi;\mathcal{A}nc(R);\varphi^{-1}$, that is $\varphi\colon \mathcal{A}nc(Q) \to \mathcal{A}nc(R)$.

As $\mathcal{A}$nc always produces a transitive and reflexive relation, we could just as well say that it is a functor on $\mathcal{R}$eln to $\mathcal{T}$reln, where $\mathcal{T}$reln is the full subcategory of $\mathcal{R}$eln got by restricting the class of objects to transitive and reflexive relations only.

We now define the functor ac: $\mathcal{B}ra^{\Gamma} \to \mathcal{T}$reln (for <u>acc</u>essibility) as the composition:

<u>Definition III.2</u>: ac = $\mathcal{A}$nc $\circ \Sigma$.

We also give:

<u>Definition III.3</u>: If $\mathcal{A}$ is a BRA, with sets $S \subseteq |\mathcal{A}|$, $T \subseteq |\mathcal{A}|$, then $ac(\mathcal{A}) \upharpoonleft S \downharpoonright T$ is called the <u>relation computed by $\mathcal{A}$ from S to T</u> (or, if $S = T$, the <u>relation computed by $\mathcal{A}$ on S</u>).

We are now ready to state an almost obvious result.

<u>Proposition III.1</u>: If a BRA $\mathcal{A}$ is monogenic, with $S \subseteq |\mathcal{A}|$, $T \subseteq |\mathcal{A}|$, and in addition $\Sigma\mathcal{A} \upharpoonleft T = 0$ (we are here using 0 ambiguously to denote any empty relation. The equation simply says that the elements of T are all "dead ends".), then the relation computed by $\mathcal{A}$ from S to T is a partial function.

<u>Proof</u>: Let $i: S \subseteq A$ and $j: T \subseteq A$ be the insertions, and let $q = \Sigma\mathcal{A}$. The hypothesis on $\Sigma\mathcal{A} \upharpoonleft T$ now reads $j;q = 0$. Then since $\mathcal{A}$ is monogenic, q^k is a partial function for each $k \geq 0$. Let $r = i;ac(\mathcal{A});j^{-1}$, then we have to show $r^{-1};r \subseteq 1_T$.

$$r^{-1};r = \bigcup_{n \geq 0} \bigcup_{m \geq 0} j;q^{-m};i^{-1};i;q^{n};j^{-1}$$

$$\subseteq \bigcup_{n \geq 0} \bigcup_{m \geq 0} j;q^{-m};q^{n};j^{-1} .$$

If $m = n$, $j;q^{-m};q^{n};j^{-1} \subseteq j;j^{-1}$ (since q^{m} a partial function)

$$= 1_T \ .$$

If $m < n$, $j;q^{-m};q^{n};j^{-1} \subseteq j;q;q^{n-m-1};j^{-1}$

$$= 0;q^{n-m-1};j^{-1} = 0 \ .$$

If $m > n$, $j;q^{-m};q^{n};j^{-1} \subseteq j;q^{-m+n+1};q^{-1};j^{-1}$

$$= j;q^{-m+n+1};0 = 0 \ .$$

Hence, $r^{-1};r \subseteq \bigcup_{n \geq 0} \bigcup_{m \geq 0} 1_T = 1_T$, as required. ∎

IV. Semantics of Programming Languages

The purpose of this chapter is to announce the intention, and justify it with some examples, to model programming languages by operator algebras, and to give their semantics by homomorphisms to other operator algebras.

Definition IV.1: An Ω-algebra, or operator algebra of signature Ω, is a function $\underline{A}: \Omega \to \bigcup_{k \geq 0} |\underline{A}|^k \to |\underline{A}|$, associating with each operator ω in a set Ω a k-ary function or operation (k is called the arity of ω) on a set $|\underline{A}|$ (called the carrier of $\underline{A}$) to itself.

Without formally complicating this definition, we shall assume that the signature of an algebra somehow carries the arities along with it, so that when we speak of two Ω-algebras, we shall always assume that each operator in Ω denotes operations of the same arity in both. If we were to be perfectly rigorous, we might adopt some such device as making the signature a function which assigned arities to operators.

Defintion IV.2: A homomorphism $\varphi: \underline{A} \to \underline{B}$, where $\underline{A}$, $\underline{B}$ are Ω-algebras, is a function $\varphi: |\underline{A}| \to |\underline{B}|$ with the property, for each $\omega \in \Omega$ of arity k, $\varphi(\underline{A}_\omega(a_1 \ldots a_k)) = \underline{B}_\omega(\varphi a_1 \ldots \varphi a_k)$. We will sometimes say, especially when introducing a homomorphism and its target algebra simultaneously, that φ carries each operation $\underline{A}_\omega$ to the corresponding $\underline{B}_\omega$.

Plainly for each Ω (with built-in arities) the Ω-algebras form a concrete category with homomorphisms as morphisms, although we shall not exploit this fact.

Definition IV.3: Of two Ω-algebras $\underline{A}$ and $\underline{B}$, $\underline{B}$ is a subalgebra of $\underline{A}$ ($\underline{B} \leq \underline{A}$) if $|\underline{B}| \subseteq |\underline{A}|$, $\underline{B}$ is closed under the operations of $\underline{A}$, and for all $\omega \in \Omega$, $\underline{B}_\omega = \underline{A}_\omega \upharpoonleft |\underline{B}|^k \downharpoonright |\underline{B}|$, where k is the arity of ω.

Definition IV.4: A set $X \subseteq |\underline{A}|$ generates an algebra $\underline{A}$ if the elements of $|\underline{A}|$ are just the finite combinations of elements of X under the operations of $\underline{A}$. It is readily shown that any $X \subseteq |\underline{A}|$ generates the smallest subalgebra of $\underline{A}$ containing X, that is, the intersection of all such subalgebras of $\underline{A}$.

Definition IV.5: An Ω-algebra is the word algebra, or free anarchic algebra, on a set X (symbolized by $\underline{W}_\Omega(X)$) if $\underline{W}_\Omega(X)$ is generated by X and moreover every element of $\underline{W}_\Omega(X)$ has a unique expression as a finite combination of elements of X.

In effect, the elements of $\underline{W}_\Omega(X)$ are just the expressions built on X with the operators in Ω, and may conveniently be taken as some standard set of strings over $X \cup \Omega$, e.g. the Polish postfix expressions.

We now present a basic result of universal algebra, which plays the role of an induction principle in reasoning about algebras and homomorphisms.

Proposition IV.1: (Unique Extension Lemma, Part I.) If $f: X \to |\underline{B}|$ is a function, with $\underline{B}$ an Ω-algebra, then there exists a unique homomorphism $\hat{f}: W_\Omega(X) \to \underline{B}$ with $\hat{f} \upharpoonleft X = f$.

Proof. Suppressing explicit use of induction (as indeed was done in the definition of the word algebra) we see that the unique expression for

each element w of $\underline{W}_\Omega(X)$ as a combination of elements of X gives, via repeated use of the property of $\hat{f}$, that it must be a homomorphism, a unique way in which we must compute $\hat{f}(w)$ as a combination of images of elements of X under f. Because we are never required to compute $\hat{f}(w)$ in two ways, the function $\hat{f}$ certainly exists. ∎

<u>Proposition IV.2</u>: (Unique Extension Lemma, Part II.) If $\underline{A}$ and $\underline{B}$ are Ω-algebras, X generates $\underline{A}$, and $f: X \to |\underline{B}|$, then there is at most one homomorphism $\tilde{f}: \underline{A} \to \underline{B}$ with $\tilde{f} \upharpoonright X = f$.

<u>Proof</u>. Consider the diagram:

$$\begin{array}{ccc} \underline{W}_\Omega(X) & & \\ {\scriptstyle \hat{1}_X}\big\downarrow & \searrow^{\hat{f}} & \\ \underline{A} & \overset{\tilde{f}}{\dashrightarrow} & \underline{B} \end{array}$$

(Here $\hat{1}_X$, $\hat{f}$ are as asserted to exist by Proposition IV.1.) First, any $\tilde{f}$ must satisfy $\tilde{f}\circ\hat{1}_X = \hat{f}$, because $\tilde{f}\circ\hat{1}_X: \underline{W}_\Omega(X) \to \underline{B}$ is a homomorphism agreeing with f on X, and $\hat{f}$ is the unique such homomorphism. Second, $\hat{1}_X$ is a surjection, because X generates $\underline{A}$, hence $\hat{1}_X$ is right cancellable, hence $\tilde{f}\circ\hat{1}_X$ unique yields $\tilde{f}$ (if it exists) unique. ∎

Now for some examples of programming language fragments, with their semantics given by homomorphisms. These examples will be carried on in later chapters.

<u>Example SS</u>: Language $\underline{SS}(V)$ (for "statement sequences") is the free semigroup over the vocabulary V, with one binary operation, concatenation, denoted by the operator $\square$. The algebra of meanings consists of

functions (we could if we chose take partial functions or even relations) on a set Q to itself, with the operation of functional composition. Given any "interpretation" $i: V \to (Q \to Q)$ of the individual statements as functions on Q to itself, we take for the meanings of arbitrary sequences values of the unique extension $\tilde{i}$ of i to a homomorphism of semigroups. The basic property of the free semigroup, analogous to that of the free anarchic algebra, is that this extension is always possible.

Example AE: Language AE (for "arithmetic expressions") is simply $\underline{W}_{\Omega}(X)$ for any set of operators Ω and set X of what we call "variables". The meanings may be in an arbitrary Ω-algebra $\underline{A}$, and the meaning homomorphism is of course uniquely specified by requiring it to extend a given map or "environment" $i: X \to |\underline{A}|$. To call these arithmetic expressions merely follows the example of Burstall and Landin [Burs 69], and reflects the programming tradition that operations of arbitrary arity are commonly available only for use with numbers.

Example BE ("Boolean expressions"): This is just a special case of Example AE which we will want to consider later, got by taking $\Omega = \{\wedge, \vee, \supset, \neg\}$. There are two possible meaning algebras for BE which we shall find of interest. The first is just the two-element Boolean algebra B2, that is the set {true, false} furnished with the classical truth-functions and, or, implies, not. The second is McCarthy's three-valued logic B3, with carrier {true, false, undef}, whose truth functions are most perspicuously defined by first giving a truth table for the conditional operator:

p	if p then q else r
true	q
false	r
undef	undef

and then defining the more traditional connectives as follows:

$$p \wedge q =_{df} \underline{\text{if}}\ p\ \underline{\text{then}}\ q\ \underline{\text{else}}\ \underline{\text{false}}$$

$$p \vee q =_{df} \underline{\text{if}}\ p\ \underline{\text{then}}\ \underline{\text{true}}\ \underline{\text{else}}\ q$$

$$p \supset q =_{df} \underline{\text{if}}\ p\ \underline{\text{then}}\ q\ \underline{\text{else}}\ \underline{\text{true}}$$

$$\neg q =_{df} \underline{\text{if}}\ p\ \underline{\text{then}}\ \underline{\text{false}}\ \underline{\text{else}}\ \underline{\text{true}} \quad .$$

In either case, the construction of a meaning homomorphism procedes just as in Example AE.

An objection which has been raised to algebraic semantics as presented in [Burs 69] is that it is unsatisfying to have the meaning of a program available only conditionally, only after an interpretation for the free variables has been supplied. One would like to find meanings which may be assigned to programs and their parts in isolation from interpretations, environments, or the like -- formal replacements for the idea of "just what we understand by a (piece of) program". This want can indeed be supplied, and that without abandoning the algebraic approach to semantics, as we shall now show.

Given any set X and any Ω-algebra $\underline{A}$, we may define an algebra of functions $\underline{F}$ (depending on X and $\underline{A}$) with carrier $((X \to |\underline{A}|) \to |\underline{A}|)$ and operations given "pointwise" by $\underline{F}_\omega(f_1 \ldots f_k) : i \mapsto \underline{A}_\omega(f_1(i) \ldots f_k(i))$.

There is a natural homomorphism $\Phi\colon \underline{W}_\Omega(X) \to \underline{F}$, namely the unique extension to a homomorphism of the mapping $\varphi\colon X \to |\underline{F}|$ defined by

$$\varphi(x)\colon i \mapsto i(x) \quad .$$

We may also define, for each $i\colon X \to |\underline{A}|$, an "application" function $app_i\colon |\underline{F}| \to |\underline{A}|$, given by $app_i\colon f \mapsto f(i)$. We then have the following easy proposition.

<u>Proposition IV.3</u>: Any homomorphism $\pi\colon \underline{W}_\Omega(X) \to \underline{A}$ factors through $\underline{F}$: namely $\pi = app_{\pi \upharpoonright X} \circ \Phi$.

<u>Proof</u>. First, it is immediate that app_i is a homomorphism $: \underline{F} \to \underline{A}$; we have:

$$app_i\colon \underline{F}_\omega(f_1 \dots f_k) \mapsto \underline{F}_\omega(f_1 \dots f_k)(i)$$

$$= \underline{A}_\omega(f_1(i) \dots f_k(i)) \quad ,$$

by definition of $\underline{F}$. Second, $app_{\pi \upharpoonright X} \circ \Phi$ agrees with π on X :

$$x \overset{\Phi}{\mapsto} \{i \mapsto i_x\} \xmapsto{app_{\pi \upharpoonright X}} \pi(x) \quad .$$

But π is the unique homomorphism that agrees with π on X ; the composition of homomorphisms $app_{\pi \upharpoonright X} \circ \Phi$ must be the same as π . ■

Observe that in the case of languages which permit binding of variables, we may not have the option of supplying a "parameterized" meaning homomorphism for each environment, but may be forced to take functions of environments for meanings if we are to assign a meaning to every phrase.

An informal sketch of a possible algebraic semantics for the λ-calculus will provide an example of this situation. We take for granted the existence of a suitable domain D_∞ of denotations for closed λ-expressions, as constructed by Scott [Scott 69]. The aim here is to present this denotation mapping as a restriction of a homomorphism Φ which assigns meanings to all λ-expressions, closed or with free variables. We take $\underline{\Lambda}(X)$, the λ-expressions with variables from the set X , to be a word algebra, with one binary operation "apply" and a separate unary operation "abstract on x " for each $x \in X$. The algebra of meanings has carrier $(X \to D_\infty) \to D_\infty$. As in Proposition IV.3 Φ , the meaning homomorphism, will be the unique extension of the function $\varphi: X \to ((X \to D_\infty) \to D_\infty)$ given by $\varphi(x): i \mapsto i_x$. Also, as one would expect, the application operation in the algebra of meanings is obtained pointwise from application in D_∞ : $f(g): i \mapsto f_i(g_i)$. For the abstraction operations, however, we require an effect which depends crucially on environments, namely $\text{abstract}_x(f): i \mapsto g$, where $g: d \mapsto f_{i'}$ for $d \in D_\infty$, and finally i' is an environment like i except that d has been bound to x , that is:

$$i': y \mapsto \underline{\text{if}}\ y = x\ \underline{\text{then}}\ d\ \underline{\text{else}}\ i_y \ .$$

We note that Φ carries closed λ-expressions to constant functions in $(X \to D_\infty) \to D_\infty$, i.e., effectively to elements of D_∞ , and so we have our original denotations back again.

V. Compilers are Homomorphisms

We shall model compilers by homomorphisms from one programming language, *qua* operator algebra, to another. It follows that we shall take no interest in compilers-as-programs; we shall from the beginning be satisfied with a mathematical description of the function to be computed by a compiler, which will ordinarily take the form of a specification of the translation for a generating set of the source language. This together with the requirement that the translation be a homomorphism will by the unique extension lemma specify it completely.

The translation functions we shall consider will be of quite a special form. Each will produce, given any element of the source operator algebra (which is to say any phrase of the source language) a result which is a BRA -- intuitively speaking a flowchart for computing on a suitable machine whatever relation is the meaning of the phrase. This means that our target operator algebra will be in every case one whose elements are BRAs. The bulk of this chapter will be devoted to describing a class of operations for building big BRAs from little ones, from which we will be able to select suitable operations for the target algebras of the examples of compilation we shall wish to study.

The example compilers we shall model will all be straightforward and non-optimizing; the operations we require for target operator algebras are intuitively all of a very simple sort: namely, to take all the operands (flowchart fragments compiled from subphrases) and "patch them together", perhaps with an additional constant fragment peculiar to the operation, to give a bigger flowchart fragment as the result of compiling the whole phrase.

Let us take up Example SS again to show what is meant. Our idea is that the compiler for statement sequences should simply carry out the modelling of sequences by straight-line BRAs informally described in Example I.3 . That is, we want a compiling function $\varkappa$ which produces from, e.g., the unit statement sequence f , a BRA which looks like $\overset{\cdot}{\underset{\cdot}{\downarrow}} f$. Similarly, we want $\varkappa(g) = \overset{\cdot}{\underset{\cdot}{\downarrow}} g$ and $\varkappa(f \Box g) = (\cdot \overset{f}{\mapsto} \cdot \overset{g}{\mapsto} \cdot)$. But since we want $\varkappa$ to be a homomorphism, we need to construct a target algebra, call it $\widehat{\underline{SS}}: \{\Box\} \to (D \times D \to D)$, where $D \subseteq Obj(\mathcal{B}ra^{\{f,g,\ldots\}})$, such that $\widehat{\underline{SS}}_{\Box}: \langle \overset{\cdot}{\underset{\cdot}{\downarrow}} f, \overset{\cdot}{\underset{\cdot}{\downarrow}} g \rangle \mapsto (\cdot \overset{f}{\mapsto} \cdot \overset{g}{\mapsto} \cdot)$.

Note that the right-hand sides of these equations do not as yet denote specific BRAs because we have not said what the carriers are. Intuitively this really does not matter -- we are only interested in what BRA we have up to isomorphism. However, as a technical device to assist in defining the requisite operations, we shall, as will be seen shortly, make fixed choices for the "entry and exit" carrier elements -- i.e., those which will serve as points of attachment to other BRAs.

For another example of what we want to get compilers to do, let us take Example AE, our language of arithmetic expressions, i.e., the algebra $\underline{AE} = \underline{W}_{\{+,\ldots\}}(\{x,y,\ldots\})$. Our idea is to compile in effect Polish postfix code for a stack machine, that is we want a homomorphism $\varkappa': \underline{AE} \to \widehat{\underline{AE}}$, where $\widehat{\underline{AE}}: \{+,\ldots\} \to \bigcup_{i \geq 0} (D'^{(i)} \to D')$, with $D' \subseteq Obj(\mathcal{B}ra^{\{+,\ldots,Lx,Ly,\ldots\}})$. Here L stands for "load to the top of the stack"; $+$ or the like is an operator both in the algebra of

BRAs $\widehat{\underline{AE}}$ and in the individual BRAs which are its elements; in the latter use it will turn out to mean, as we might expect, "remove the top two elements of the stack, add them, and replace their sum on the stack". We want $\kappa'(x) = \overset{\cdot}{\underset{\cdot}{\uparrow}}Lx$, $\kappa'(y) = \overset{\cdot}{\underset{\cdot}{\uparrow}}Ly$, and

$\kappa'(x+y) = (\cdot \overset{Lx}{\mapsto} \cdot \overset{Ly}{\mapsto} \cdot \overset{+}{\mapsto} \cdot)$, so we need an operation

$\widehat{\underline{AE}}_+ : D' \times D' \to D'$ such that $\widehat{\underline{AE}}_+(\overset{\cdot}{\underset{\cdot}{\uparrow}}Lx, \overset{\cdot}{\underset{\cdot}{\uparrow}}Ly) = \cdot \overset{Lx}{\mapsto} \cdot \overset{Ly}{\mapsto} \cdot \overset{+}{\mapsto} \cdot$

(and similarly, of course, for any other pair of operands).

In every operator algebra of BRAs which we define, we shall require that each of the elements (BRAs) shall possess, as a subset of its carrier, a certain fixed set (that is, the same for all elements, and for all operator algebras) of "distinguished nodes" which will provide the necessary points of attachments to other BRAs.

Intuitively speaking, an operation in such an algebra which does nothing but patch its arguments together can be completely specified by telling the fate of all the distinguished nodes in the arguments -- that is, what sets of distinguished nodes are to coalesce into single nodes and, of these latter, which are to become the distinguished nodes of the result, and which are to be "undistinguished". If A is the set of distinguished nodes, B is A together with as many new undistinguished nodes as are needed for the resulting BRA, and k is the arity of the operation, then we can convey just the information we need by giving a function $\tilde{p}: \underbrace{A \dot{\cup} \dots \dot{\cup} A}_{k \text{ times}} \to B$. (Note that we must have $A \subseteq B$ to ensure that the result of the operation possesses all distinguished nodes.)

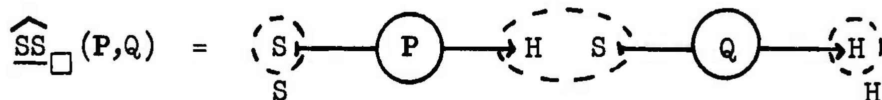
Thus, for the algebra $\widehat{\underline{SS}}$, we take for distinguished nodes the set $\{S,H\}$ (meant to suggest "start", "halt"), we take I to be an undistinguished node, and we expect to be able to characterize $\widehat{\underline{SS}}_{\square}$ by a suitably chosen function $\tilde{p}: \{S,H\} \dot{\cup} \{S,H\} \to \{S,I,H\}$. To be able to distinguish the two copies of $\{S,H\}$ we give names to the injections into their disjoint union:

$$\{S,H\} \xrightarrow{i} \{S,H\} \dot{\cup} \{S,H\} \xleftarrow{j} \{S,H\} ,$$

and we can now define $\tilde{p}$ to have the effect it should by:

$$\tilde{p}: \left\{ \begin{array}{ccc} iS & \mapsto & S \\ iH & \mapsto & I \\ jS & \mapsto & I \\ jH & \mapsto & H \end{array} \right\} .$$

This is cumbersome notation; we have had to invent the names $\tilde{p}$, i , j , I for objects which are of no interest in themselves. We hasten to introduce a more pictorial notation, which specifies the same operation $\underline{SS}_{\square}$ by:



(Note that P and Q are dummy variables, and stand for any BRAs in $\widehat{\underline{SS}}$.) This style of definition we call a <u>construction diagram</u>.

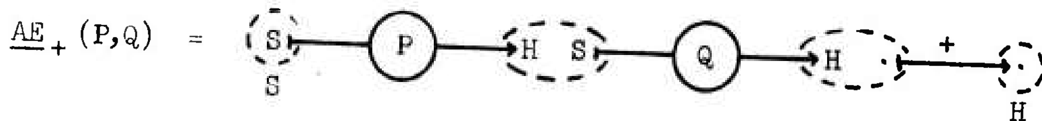
Having defined the algebra $\widehat{\underline{SS}}$, we can now define the compiler $\kappa: \underline{SS} \to \widehat{\underline{SS}}$ by its effect on the generating set $\{f,g,\ldots\}$; this effect should of course be:

$$\kappa : f \mapsto \overset{S}{\underset{H}{\top}} f \quad .$$

A patching-together operation on BRAs with set A of distinguished nodes, which incorporates in its result not only its argument BRAs but also a constant (i.e., characteristic of the operation) BRA $\underline{\mathcal{C}}$, may similarly be determined by a function

$$\tilde{p}: |\underline{\mathcal{C}}| \,\dot{\cup}\, A \,\dot{\cup}\, \ldots \,\dot{\cup}\, A \rightarrow B \quad .$$

We will of course also use construction diagrams to describe operations of this sort. In particular, taking $+$ to be a typical operator of $\widehat{\underline{AE}}$, we define the operation $\widehat{\underline{AE}}_+$ by:



We can immediately read off from this diagram that the set of distinguished nodes for $\widehat{\underline{AE}}$ is again $\{S,H\}$, and that the constant BRA involved is $\cdot \overset{+}{\mapsto} \cdot$. However, the diagram suppresses information of no interest, namely what is the carrier of $\cdot \overset{+}{\mapsto} \cdot$, and what are the two undistinguished nodes introduced by the operation.

We complete the definition of the compiler $\kappa': \underline{AE} \rightarrow \widehat{\underline{AE}}$ by requiring, for $x \in X$,

$$\kappa' : x \mapsto \overset{S}{\underset{H}{\top}} Lx \quad .$$

We shall give the name of "constructions" to the class of operations in algebras of BRAs of which we have just seen two examples. Although we have a convenient notation for describing constructions, we lack as yet an explicit definition of what it is for an operation on BRAs to be a construction. We now prepare the ground for that definition.

It will turn out that for the technical development, especially in Chapter VI, the following property of constructions will be important: *The result of a construction simulates each of its arguments* (and also the constant BRA, if any, which it involves). By the characteristic property of the coproduct, $\oplus$, in any category of BRAs, we see that an instance of a construction which combines a constant BRA $\underline{\mathcal{C}}$ with arguments $\underline{\mathcal{P}}_1,\ldots,\underline{\mathcal{P}}_k$, yielding a result which simulates each of them, may be factored into simulations of $\underline{\mathcal{P}}_1,\ldots,\underline{\mathcal{P}}_k$ and $\underline{\mathcal{C}}$ by $\underline{\mathcal{C}} \oplus \underline{\mathcal{P}}_1 \oplus \cdots \oplus \underline{\mathcal{P}}_k$, followed by a simulation of $\underline{\mathcal{C}} \oplus \underline{\mathcal{P}}_1 \oplus \cdots \oplus \underline{\mathcal{P}}_k$ by that result. This latter simulation is of a kind we shall call a "projection"; considered as a function it merely collapses the appropriate collections of nodes to single nodes. As a morphism of BRAs, it will have the property that the simulating BRA possesses no relation-instances beyond those necessary to make it simulate the coproduct; that is, every relation-instance will be attributable to an antecedent either in $\underline{\mathcal{C}}$ or in one of the $\underline{\mathcal{P}}_i$. This last requirement, together with the choice of $\underline{\mathcal{C}}$ and of the projection function, will suffice to determine the result of the construction uniquely.

Thus we are motivated to define:

Definition V.1: A simulation $p: \underline{\mathcal{A}} \to \underline{\mathcal{B}}$ in $\mathcal{B}ra^{\Gamma}$ is a projection if p is a surjection and for all $\gamma \in \Gamma$, $\underline{\mathcal{B}}_{\gamma} = p^{-1};\underline{\mathcal{A}}_{\gamma};p$.

We are now able to define "construction" and some related terms:

Definition V.2: Given sets $A \subseteq B$, a Γ-BRA $\underline{\mathcal{C}}$, a function $\tilde{p}: |\underline{\mathcal{C}}| \dot{\cup} \underbrace{A \dot{\cup} \cdots \dot{\cup} A}_{k \text{ times}}$, and a set L of Γ-BRAs with the property $A \subseteq |\underline{\mathcal{P}}|$ for all $\underline{\mathcal{P}} \in L$, then an operation $o: L^k \to L$ is the construction derived from $\tilde{p}$ and $\underline{\mathcal{C}}$ if and only if the effect of o on arguments $\underline{\mathcal{P}}_1 \cdots \underline{\mathcal{P}}_k$ is given by the following diagram of morphisms (simulations) in $\mathcal{B}ra^{\Gamma}$:

$$\begin{array}{ccccc} \underline{\mathcal{C}} & \underline{\mathcal{P}}_1 & \cdots & \underline{\mathcal{P}}_k \\ \downarrow i_0 & \downarrow i_1 & \cdots & \swarrow i_k \\ \multicolumn{4}{c}{\underline{\mathcal{C}} \oplus \underline{\mathcal{P}}_1 \oplus \cdots \oplus \underline{\mathcal{P}}_k} \\ \multicolumn{4}{c}{\downarrow p} \\ \multicolumn{4}{c}{o(\underline{\mathcal{P}}_1, \ldots, \underline{\mathcal{P}}_k)} \end{array}$$

where each i_j is the injection into the disjoint union:

$$i_j: |\underline{\mathcal{P}}_j| \to |\underline{\mathcal{C}}| \dot{\cup} |\underline{\mathcal{P}}_1| \dot{\cup} \cdots \dot{\cup} |\underline{\mathcal{P}}_k| ,$$

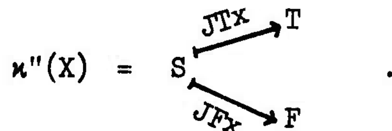
and where $p: |\underline{\mathcal{C}}| \dot{\cup} |\underline{\mathcal{P}}_1| \dot{\cup} \cdots \dot{\cup} |\underline{\mathcal{P}}_k| \to (|\underline{\mathcal{P}}_1|-A) \dot{\cup} \cdots \dot{\cup} (|\underline{\mathcal{P}}_k|-A) \dot{\cup} B$ is the projection (depending on the $\underline{\mathcal{P}}_j$'s as well as on $\tilde{p}$ and $\underline{\mathcal{C}}$) whose effect is:

$$p: a \mapsto \tilde{p}(a) \quad \text{for} \quad a \in |\underline{\mathcal{C}}| \dot{\cup} A \dot{\cup} \cdots \dot{\cup} A ,$$

$$p: x \mapsto x \quad \text{for} \quad x \in (|\underline{\mathcal{P}}_j|-A) .$$

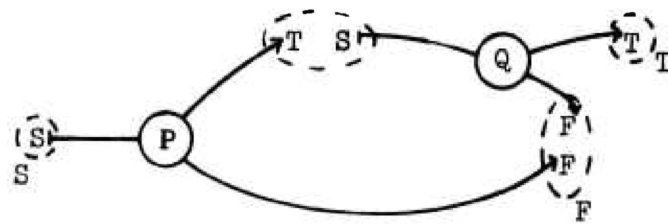
In the case where $\mathcal{C}$ is absent -- i.e., we have a pure patching-together operation -- we still call o a construction, and say that it is derived from $\tilde{p}$. In both cases we also call $\tilde{p}$ the kernel of o .

Example BE provides a more interesting application for our method of defining constructions than those just given. The idea is to compute the value of a Boolean expression in the Lisp (as distinct from Algol) fashion by a series of jumps, one for each occurrence of a variable in the expression. We want a compiler $\kappa'': \underline{W}_{\{\wedge, \vee, \supset, \neg\}}(X) \rightarrow \widehat{\underline{BE}}$. The BRAs in $\widehat{\underline{BE}}$ will have the operator set $\{JTx, JFx \mid x \in X\}$ (standing for "jump if x is true", "jump if x is false"). We specify the effect of κ'' on the generating set by

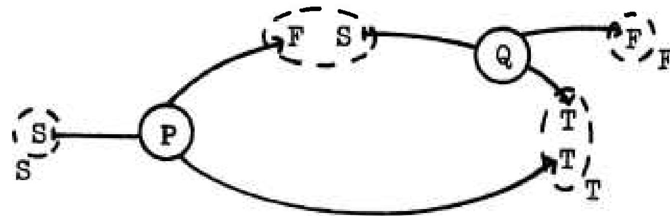


Note that the BRAs we are compiling have three distinguished nodes: a start node, a true exit, and a false exit. The following construction diagrams give the effect of the operations in $\widehat{\underline{BE}}$, and so complete the definition of the compiler κ'' .

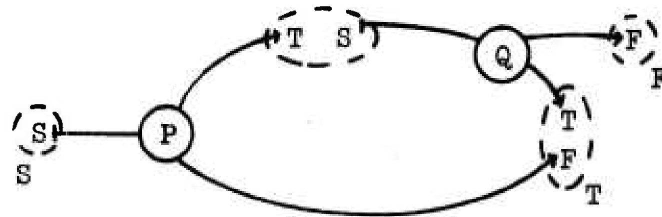
$\widehat{\underline{BE}}_{\wedge}(P,Q)$ =



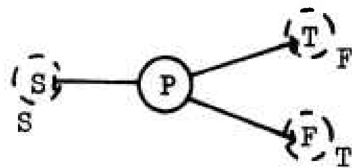
$\widehat{\underline{BE}}_{\vee}(P,Q)$ =



$\widehat{\underline{BE}}_{\supset}(P,Q)$ =



$\widehat{\underline{BE}}_{\neg}(P)$ =

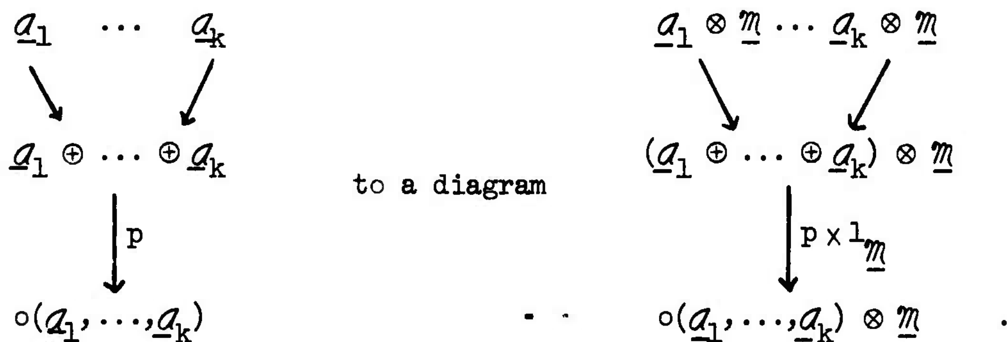


VI. Semantics of BRAs is a Homomorphism

It has been asserted in Chapter IV that the semantics of programming languages can be given by homomorphisms. In the case of the operator algebras of BRAs which will serve as the target languages for our compilers, a potential conflict arises, because we already have a natural way of arriving at the function (or, in general, relation) computed by a BRA (program) under a given interpretation: first run ($\otimes$) the program on the BRA which is the interpretation; second, take the accessibility relation of the product; third, restrict the relation so obtained to whatever set of starting and finishing states we are interested in. The purpose of this section is to set our fears at rest, by showing that each of these three steps, and hence their composition, induces a homomorphism which preserves whatever constructions may have been defined in an algebra of BRAs. What is proved here is similar in content to the main result of Landin's "Program-Machine Symmetric Automata Theory". [Land 70].

Proposition VI.1: If $\circ$ is a k-ary construction (on Γ-BRAs with set of distinguished nodes A) derived from $\tilde{p}: A \dot{\cup} \ldots \dot{\cup} A \to B$, and $\underline{\mathfrak{M}}$ is any Γ-BRA, then the functor $\underline{\mathfrak{P}} \mapsto \underline{\mathfrak{P}} \otimes \underline{\mathfrak{M}}$ provides a homomorphism carrying $\circ$ to the construction $\circ'$ (on Γ-BRAs with set of distinguished nodes $A \times |\underline{\mathfrak{M}}|$) derived from $\tilde{p} \times 1_{\underline{\mathfrak{M}}}$.

Proof. $\underline{\mathfrak{P}} \mapsto \underline{\mathfrak{P}} \otimes \underline{\mathfrak{M}}$, being a functor (it is trivial that fixing one argument of a bifunctor such as $\otimes$ does yield a functor), carries every diagram



Recalling the natural isomorphism

$$(\underline{a}_1 \oplus \cdots \oplus \underline{a}_k) \otimes \underline{m} \cong \underline{a}_1 \otimes \underline{m} \oplus \cdots \oplus \underline{a}_k \otimes \underline{m} ,$$

we see that the latter diagram is an instance of the construction derived from $\tilde{p} \times 1_{\underline{m}}$, and indeed giving the result $o(\underline{a}_1, \ldots, \underline{a}_k) \otimes \underline{m}$ for the arguments $\underline{a}_1 \otimes \underline{m}, \ldots, \underline{a}_k \otimes \underline{m}$, so that we have a homomorphism of constructions. ∎

We may define constructions on relations exactly as on BRAs. ($\mathcal{R}$eln is of course isomorphic to $\mathcal{B}\text{ra}^{\{0\}}$, so that we may always if we wish regard relations as a special case of BRAs.) A projection $p: R \to S$ of relations is simply a surjection $p: |R| \to |S|$ such that $S = p^{-1};R;p$, and again we have that an instance of a construction can be diagrammed as coproduct followed by projection.

In either of the categories $\mathcal{R}$eln and $\mathcal{B}\text{ra}^{\Gamma}$, when we have a diagram of the form

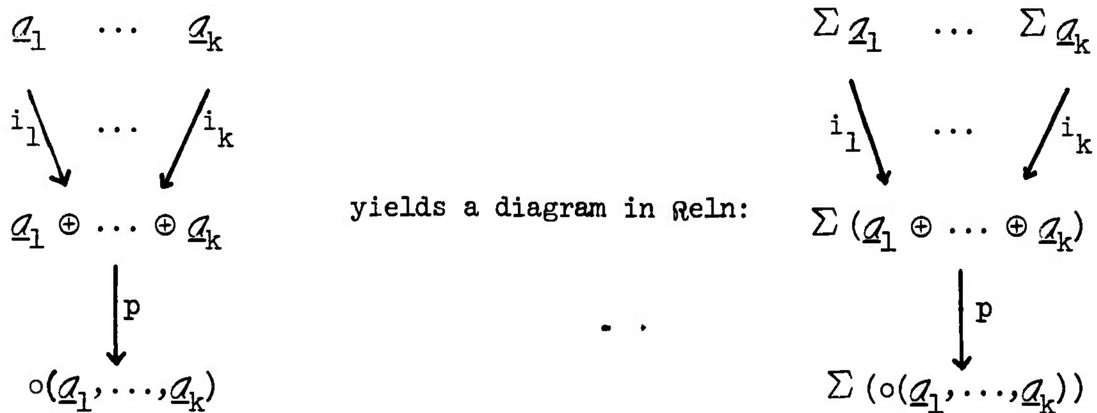
$$
\begin{array}{ccc}
X_1 & \cdots & X_k \\
{\scriptstyle i_1}\searrow & \cdots & \swarrow{\scriptstyle i_k} \\
X_1 \oplus & \cdots & \oplus X_k \\
 & \downarrow{\scriptstyle p} & \\
 & o(X_1,\ldots,X_k) &
\end{array}
$$

with $i_1,\ldots,i_k$ the natural injections into the coproduct, p a projection, we will for brevity denote the composites $p\circ i_1,\ldots,p\circ i_k$ by $p_1,\ldots,p_k$.

We now have the terminology we need to state and prove the remaining two propositions of this chapter.

Proposition VI.2: If o is a construction on Γ-BRAs derived from the kernel function $\tilde{p}$, and o'' is the construction on relations derived from $\tilde{p}$, then ac , the accessibility functor, acts as a homomorphism of operator algebras carrying o to the operation on relations $\mathcal{I}nc\circ o''$.

Proof. Recall that $ac = \mathcal{I}nc\circ\Sigma$; we shall show first that Σ acts as a homomorphism of constructions carrying o to o'' . This is easy; because Σ is a functor whose mapping function is the identity, every diagram in $\mathcal{B}ra^{\Gamma}$ of an instance of o applied to argument BRAs $\mathcal{A}_1,\ldots,\mathcal{A}_k$:



and from the obvious equality of binary relations on $|\underline{a}_1| \dot{\cup} \cdots \dot{\cup} |\underline{a}_k|$:

$$\Sigma(\underline{a}_1 \oplus \cdots \oplus \underline{a}_k) = \Sigma\underline{a}_1 \oplus \cdots \oplus \Sigma\underline{a}_k$$

we conclude that the latter diagram in fact displays an instance of o'' , and hence we have the necessary homomorphism property for Σ :

$$\Sigma(o(\underline{a}_1,\ldots,\underline{a}_k)) = o''(\Sigma\underline{a}_1,\ldots,\Sigma\underline{a}_k) \quad .$$

We have now to derive the homomorphism property for ac , namely:

$$ac(o(\underline{a}_1,\ldots,\underline{a}_k)) = \mathcal{A}nc \circ o''(ac\,\underline{a}_1,\ldots,ac\,\underline{a}_k) \quad .$$

This equation is easily established in one direction:

$$\begin{aligned} ac(o(\underline{a}_1,\ldots,\underline{a}_k)) &= \mathcal{A}nc(\Sigma(o(\underline{a}_1,\ldots,\underline{a}_k))) \\ &= \mathcal{A}nc(o''(\Sigma\underline{a}_1,\ldots,\Sigma\underline{a}_k)) \\ &= \bigcup_{r\geq 0}\left[\bigcup_{1\leq j\leq k} p_j^{-1};\Sigma\underline{a}_j;p_j\right]^{(r)} \\ &\subseteq \bigcup_{r\geq 0}\left[\bigcup_{1\leq j\leq k} p_j^{-1};\left(\bigcup_{s\geq 0}[\Sigma\underline{a}_j]^{(s)}\right);p_j\right]^{(r)} \\ &= \mathcal{A}nc \circ o''(ac\,\underline{a}_1,\ldots,ac\,\underline{a}_k) \quad . \end{aligned}$$

To go the other way, we need to use the fact that the ancestral gives the least reflexive and transitive extension of any relation; hence if we can show:

$$(*) \qquad o''(\mathrm{ac}\ \underline{\mathcal{A}}_1,\dots,\mathrm{ac}\ \underline{\mathcal{A}}_k) \subseteq \mathcal{A}\mathrm{nc}(o''(\Sigma\underline{\mathcal{A}}_1,\dots,\Sigma\underline{\mathcal{A}}_k))$$

we will have, because the right-hand side of (*) is reflexive and transitive and extends the left-hand side, our desired conclusion:

$$\mathcal{A}\mathrm{nc}(o''(\mathrm{ac}\ \underline{\mathcal{A}}_1,\dots,\mathrm{ac}\ \underline{\mathcal{A}}_k)) \subseteq \mathcal{A}\mathrm{nc}(o''(\Sigma\underline{\mathcal{A}}_1,\dots,\Sigma\underline{\mathcal{A}}_k))$$

$$= \mathrm{ac}(o(\underline{\mathcal{A}}_1,\dots,\underline{\mathcal{A}}_k)) \quad .$$

But (*) is just the inequality:

$$\bigcup_{1 \leq j \leq k} \left[p_j^{-1} ; \bigcup_{s \geq 0} (\Sigma\underline{\mathcal{A}}_j)^{(s)} ; p_j \right] \subseteq \bigcup_{r \geq 0} \left[\bigcup_{1 \leq j \leq k} p_j^{-1} ; \Sigma\underline{\mathcal{A}}_j ; p_j \right]^{(r)} ,$$

and this is true by virtue of the inequality:

$$(**) \qquad p_j^{-1};(\Sigma\underline{\mathcal{A}}_j)^{(s)};p_j \subseteq [p_j^{-1};\Sigma\underline{\mathcal{A}}_j;p_j]^{(s)}$$

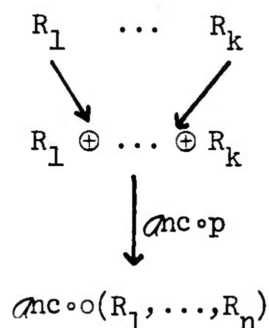
which holds for every pair of values of s and j. Finally, (**) is a consequence of p_j being a function, so that $p_j;p_j^{-1} \supseteq 1_{\underline{\mathcal{A}}_j}$ (this establishes the case $s > 0$) and also $p_j^{-1};p_j \subseteq 1_{o(\underline{\mathcal{A}}_1,\dots,\underline{\mathcal{A}}_k)}$ (this establishes the case $s = 0$). ∎

In the compiler correctness proofs which will follow in Chapters VIII - XII, to obtain the relation computed by a BRA $\underline{\mathcal{B}}$ in an operator algebra of BRAs with a given set A of distinguished nodes, we shall always take the restriction of the accessibility relation of $\underline{\mathcal{B}}$ to a relation on the set of distinguished nodes to itself, that is $(\mathrm{ac}\ \underline{\mathcal{B}}) \rceil A$.

(In the ordinary case that $\underline{\mathcal{B}}$ is the product of a program $\underline{\rho}$ and a machine $\underline{\mathfrak{M}}$, then A will be the Cartesian product of the set of distinguished nodes of $\underline{\rho}$ with the carrier of $\underline{\mathfrak{M}}$.) This will not always give a result in accordance with our intuitive notion of the relation or function computed -- note especially that $(\mathrm{ac}\ \underline{\mathcal{B}}) \upharpoonright A$ must be reflexive -- but it will turn out that by the choice of a suitable decoding homomorphism (from the algebra of relations computed by target language programs back to the algebra of source language meanings) we will be able to obtain the correctness results we expect. Taking this uniform view of what is the relation computed by a BRA will allow the application of the following proposition as the final step in obtaining target language semantic homomorphisms.

Proposition VI.3: If we have an operation $\mathcal{A}nc \circ o$ on reflexive and transitive relations, where o is the construction derived from $\tilde{p}: A \dot{\cup} \dots \dot{\cup} A \to B$, then restriction to A, that is the mapping of relations $R \mapsto R \upharpoonright A$, acts as a homomorphism carrying $\mathcal{A}nc \circ o$ to the operation $(\upharpoonright A) \circ \underline{\mathcal{A}nc} \circ o$.

Proof. The required homomorphism property of $(\upharpoonright A)$, for an arbitrary instance of $\mathcal{A}nc \circ o$ which we may display as the diagram in $\mathcal{T}reln$



is the equality:

$$(\mathcal{A}nc \circ o(R_1,\ldots,R_k)) \upharpoonright A \;=\; (\mathcal{A}nc \circ o(R_1 \upharpoonright A,\ldots,R_k \upharpoonright A)) \upharpoonright A \quad .$$

The inequality $\supseteq$ is immediate, because

$$o(R_1 \upharpoonright A,\ldots,R_k \upharpoonright A) \;=\; o(R_1,\ldots,R_k) \upharpoonright B \quad ;$$

and, since restricting after forming the ancestral rather than before can only produce an enlarged relation, we have

$$\mathcal{A}nc(o(R_1,\ldots,R_k) \upharpoonright B) \;\subseteq\; (\mathcal{A}nc(o(R_1,\ldots,R_k))) \upharpoonright B \quad ;$$

therefore

$$\begin{aligned}(\mathcal{A}nc \circ o(R_1 \upharpoonright A,\ldots,R_k \upharpoonright A)) \upharpoonright A \;&=\; (\mathcal{A}nc(o(R_1,\ldots,R_k) \upharpoonright B)) \upharpoonright A \\ &\subseteq\; (\mathcal{A}nc(o(R_1,\ldots,R_k))) \upharpoonright B \upharpoonright A \\ &=\; (\mathcal{A}nc \circ o(R_1,\ldots,R_k)) \upharpoonright A \quad .\end{aligned}$$

In the other direction we have to show:

$$\left(\bigcup_{i \geq 0} \left[\bigcup_{1 \leq j \leq k} p_j^{-1};R_j;p_j \right]^i \right) \upharpoonright A$$

$$\subseteq \left(\bigcup_{i \geq 0} \left[\bigcup_{1 \leq j \leq k} (p_j^{-1} \downharpoonright A);(R_j \upharpoonright A);(p_j \upharpoonleft A) \right]^i \right) \upharpoonright A \quad .$$

Because p is a projection, and so can neither coalesce a node in any $|R_j| - A$ with any other node, nor map it into B, we can have $(p_i;p_j^{-1})\colon a \mapsto a'$ only if either $a,a' \in A$ or else $i = j$ and $a = a'$. Hence for $b,b'' \in B$, we can have

$$b \xrightarrow{p_i^{-1}} a \xrightarrow{R_i} a' \xrightarrow{p_i} b' \xrightarrow{p_j^{-1}} a'' \xrightarrow{R_j} a''' \xrightarrow{p_j} b''$$

only if $a, a''' \in A$, and either $a', a'' \in A$ and $b' \in B$ as well, or else $i = j$ and $a' = a''$, in which latter case, since R_i is transitive, it must already be the case that $R_i : a \mapsto a'''$. What we have just shown is that:

$$(p_i^{-1};R_i;p_i;p_j^{-1};R_j;p_j) \rceil B \subseteq \Big[(p_i^{-1} \downharpoonleft A);(R_i \rceil A);(p_i \upharpoonleft A);(p_j^{-1} \downharpoonleft A);(R_j \rceil A);(p_j \upharpoonleft A)$$

$$\cup\ (p_i^{-1} \downharpoonleft A);(R_i \rceil A);(p_i \upharpoonleft A)\Big] \rceil B \quad .$$

An analogous computation may be made for as long a composition of any of the k relations $(p_j^{-1};R_j;p_j)$ as we like; hence for each $m \geq 0$ we have:

$$\left(\left[\bigcup_{1 \leq j \leq k} p_j^{-1};R_j;p_j\right]^m\right) \rceil B$$

$$\subseteq \left(\bigcup_{0 \leq i \leq m} \left[\bigcup_{1 \leq j \leq k} (p_j^{-1} \downharpoonleft A);(R_j \rceil A);(p_j \upharpoonleft A)\right]^i\right) \rceil B \quad ,$$

hence the same thing with " $\rceil A$ " in place of " $\rceil B$ ", (since $A \subseteq B$), and hence, taking the union over all non-negative m, the desired inequality. ∎

In the above propositions and proofs, constructions derived also from a constant BRA $\mathcal{C}$ have been left out of account, but this was purely for notational convenience. The proofs dealt solely with single operation instances, and it is plain that nothing changes if one of the $\mathcal{A}_j$ or R_j is made a constant characteristic of the operation,

rather than an argument to it. Combining this observation with Propositions VI.1,2,3 we may assert the

Result: If o is a construction on Γ-BRAs derived from $\tilde{p} : |\underline{\mathcal{C}}| \mathbin{\dot{\cup}} A \mathbin{\dot{\cup}} \ldots \mathbin{\dot{\cup}} A \to B$ and $\underline{\mathcal{C}}$, and if $\underline{\mathfrak{m}}$ is a Γ-BRA, then the mapping $\underline{\rho} \mapsto (ac(\underline{\rho} \otimes \underline{\mathfrak{m}})) \rfloor (A \times |\underline{\mathfrak{m}}|)$ is a homomorphism of operator algebras carrying o to the operation $(\rfloor A \times |\underline{\mathfrak{m}}|) \circ \mathit{Inc} \circ o'$, where o' is the construction on relations derived from $\tilde{p} \times 1_{|\underline{\mathfrak{m}}|}$ and $ac(\underline{\mathcal{C}} \otimes \underline{\mathfrak{m}})$.

In some of the applications of this result which follow, in particular in the proof of the compiler composition theorem in the next chapter, we shall assume that we always have $A \subseteq |\underline{\mathcal{C}}|$, and that

$$\tilde{p} : c \mapsto c \quad \text{for} \quad c \in |\underline{\mathcal{C}}| - A \quad .$$

Letting

$$\tilde{p}^{-} =_{df} \tilde{p} \upharpoonright (A \mathbin{\dot{\cup}} A \mathbin{\dot{\cup}} \ldots \mathbin{\dot{\cup}} A) \quad ,$$

we see that our k-ary construction o is exactly that yielded by the (k+1)-ary construction derived from $\tilde{p}^{-}$ on fixing its first argument to be $\underline{\mathcal{C}}$. In this case we shall say loosely that o is derived from $\tilde{p}^{-}$ and $\underline{\mathcal{C}}$, and it will be strictly correct to say that o is carried to the operation

$$(\rfloor A \times |\underline{\mathfrak{m}}|) \circ \mathit{Inc} \circ o'' \quad ,$$

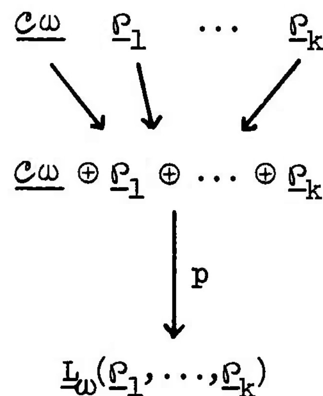
where o'' is the construction on relations derived from $\tilde{p}^{-} \times 1_{|\underline{\mathfrak{m}}|}$ and $(ac(\underline{\mathcal{C}} \otimes \underline{\mathfrak{m}})) \rfloor (A \times |\underline{\mathfrak{m}}|)$.

VII. The Compiler Composition Theorem

The theorem to be proved in this chapter provides the essential tool for compounding simple results about compiler correctness into complex ones. Intuitively, it amounts merely to the observation that if we can compile each of the individual operations of one machine into a program for a second machine (or even, what is a weaker assumption in general, compile each element of the generating set of an algebra of programs for the first machine into a program for the second, and also the program fragments from which the constructions of the first algebra are derived) then we can compile any program (belonging to the algebra) for the first machine into a program for the second.

We consider an arbitrary Ω-algebra $\underline{L}$ of the sort that might be the target algebra of a compiler, that is an algebra of BRAs, say with $|\underline{L}| \subseteq \mathrm{Obj}(\mathcal{B}ra^{\Gamma})$ with a generating set $X = \{\underline{\rho}_i \mid i \in I\}$ for some indexing set I, and with each k-ary operation $\underline{L}_\omega$ (for $\omega \in \Omega$) being a construction derived from a kernel $\tilde{p}_\omega : \underbrace{A \dot{\cup} \ldots \dot{\cup} A}_{k+1} \to B$, and from a constant Γ-BRA $\underline{\mathcal{C}\omega}$. (We are here assuming $A \subseteq |\underline{\mathcal{C}\omega}|$; this turns out to be no restriction in practice. Recall that A is the set of distinguished nodes some of which may be identified from the several operands as specified by $\tilde{p}_\omega$ in the process of "patching together" the argument BRAs of $\underline{L}_\omega$ to form the result. The set B contains all of A, in order to tell us which are the distinguished nodes of the result; B may also contain some other elements, used for images under $\tilde{p}_\omega$ of collections of nodes which are simultaneously to be identified and to become undistinguished.)

We recall that the effect of $\underline{L}_\omega$ on operands $\underline{\mathcal{P}}_1,\ldots,\underline{\mathcal{P}}_k$ is given by the diagram



with p a projection, and $(p \upharpoonleft A \cup \ldots \cup A) \downharpoonright B = \tilde{p}_\omega$.

We suppose, as is our standard practice, that the semantics of $\underline{L}$ are given by running its programs on a machine $\underline{\mathcal{M}}$, with $|\underline{\mathcal{M}}| = M$; that is, we have a semantic homomorphism $\varphi: \underline{L} \to \underline{S}$, where $\varphi: \underline{\mathcal{P}} \mapsto (ac(\underline{\mathcal{P}} \otimes \underline{\mathcal{M}})) \rceil (A \times M)$ (cf. Propositions VI.1,2,3). $\underline{S}$, the semantic algebra, has as elements relations $: A \times M \twoheadrightarrow A \times M$, and receives an induced Ω-algebraic structure as demonstrated in Chapter VI.

We shall now assume the existence of a correct compiler $\varkappa$ applicable to the $\underline{\mathcal{P}}_i$ and to the $\underline{\mathcal{C}\omega}$, producing Δ-BRAs (i.e., BRAs having an arbitrary and in general different operator set Δ) and will prove, subject to a number of hypotheses, that $\varkappa$ can be extended to a correct compiler (homomorphism) from $\underline{L}$ to a language (Ω-algebra) $\underline{K}$ of Δ-BRAs; we denote the extension by $\text{extend}(\varkappa,\underline{L})$. The hypotheses are somewhat lengthy; we shall therefore expound them separately before getting down to the statement of the theorem.

The first two hypotheses are mainly a matter of convenience; a spuriously more general theorem could be had by replacing the set

inclusions which they call for by suitable mappings. The first asserts that the results of $\varkappa$ contain the right carrier elements to permit them to be patched together by the $\tilde{p}_{\omega}$.

Hypothesis 1: For each $\underline{\rho}_i \in X$, $A \subseteq |\varkappa(\underline{\rho}_i)|$, and for each $\omega \in \Omega$, $A \subseteq |\varkappa(\underline{\mathcal{C}\omega})|$.

The second will make the states of our translated programs directly comparable to those of the originals: we may suppose that our Δ-BRAs will be given their semantics by running on a particular Δ-BRA (machine) $\underline{\mathfrak{M}}'$, with $M' =_{df} |\underline{\mathfrak{M}}'|$, and we require that M be a subset of M'; i.e., to give the inclusion mapping a name:

Hypothesis 2: $\mu: M \subseteq M'$.

We also make the abbreviation:

$$\Theta =_{df} 1_A \times \mu: A \times M \subseteq A \times M' \quad .$$

We may define a semantic function ψ for Δ-BRAs, $\psi: \mathrm{Obj}(\mathcal{B}ra^{\Delta}) \to (A \times M' \nrightarrow A \times M')$, by

$$\psi: \underline{\mathcal{B}} \mapsto \mathrm{ac}(\underline{\mathcal{B}} \otimes \underline{\mathfrak{M}}') \upharpoonright (A \times M') \quad .$$

Moreover, thanks to Hypothesis 2, there is an obvious way of "decoding" relations on $A \times M'$ as relations on $A \times M$, namely restricting them; we therefore define the mapping $\delta: (A \times M' \nrightarrow A \times M') \to (A \times M \nrightarrow A \times M)$ by

$$\delta: P \mapsto P \upharpoonright (A \times M) \quad .$$

We may now state the main hypothesis, which asserts that $\varkappa$ correctly translates the bits and pieces of which $\underline{L}$-programs are built.

Hypothesis 3: The diagram

$$\begin{array}{ccc} X \cup \{\underline{C\omega} \mid \omega\in\Omega\} & \xrightarrow{\varkappa} & \mathrm{Obj}(\mathcal{B}ra^{\Delta}) \\ \downarrow \varphi & & \downarrow \psi \\ A\times M \rightarrow\!\!\!\!\times\ A\times M & \xleftarrow{\delta} & A\times M' \rightarrow\!\!\!\!\times\ A\times M' \end{array}$$

commutes.

Unfortunately, Hypothesis 3 does not claim as much as we need; it might be that some of our relations $\psi\circ\varkappa(\underline{\rho})$ relate elements of $A\times M$ to elements of $A\times(M'-M)$ in such a way that when these relations are compounded, "too much" gets computed even from $A\times M$ to itself. We therefore require that all our relations should map $A\times M$ into itself:

Hypothesis 4: For $\underline{\rho}\in X \cup \{\underline{C\omega}\}$,

$$\psi\circ\varkappa(\underline{\rho}) \upharpoonright (A\times M) = (\psi\circ\varkappa(\underline{\rho}) \rfloor (A\times M));\theta$$

or, to say the same thing in more uniform notation:

$$\theta;\psi\circ\varkappa(\underline{\rho}) = \theta;\psi\circ\varkappa(\underline{\rho});\theta^{-1};\theta \quad .$$

We may take advantage of Hypothesis 1 to define our Ω-algebra $\underline{K}$ of Δ-BRAs in a natural way: $\underline{K}$ is generated by the set $\{\varkappa(\underline{\rho}_i) \mid i\in I\}$, and each operation $\underline{K}_{\omega}$, for $\omega\in\Omega$, is the construction derived from $\tilde{p}_{\omega}$ and $\varkappa(\underline{C\omega})$. Informally, we will build up each $\underline{K}$-program in "just the same way" as its corresponding $\underline{L}$-program. The unique extension of $\varkappa \upharpoonright X$ to a homomorphism $\mathrm{extend}(\varkappa,\underline{L}): \underline{L} \rightarrow \underline{K}$ evidently exists; it is the function which carries out the just-described correspondence (but see the remark at the end of this chapter). Having made an Ω-algebra

out of our Δ-BRAs, the semantic function ψ becomes, from the result of Chapter VI, a homomorphism, inducing an Ω-algebraic structure on its codomain; we denote this latter Ω-algebra of relations $: A \times M' \rightarrow\!\!\!\!\times A \times M'$ by $\underline{R}$.

We are now in position to state the result of the present chapter, which asserts the correctness of $\text{extend}(\varkappa, \underline{L})$.

<u>Proposition VII.1</u>. (<u>Compiler Composition Theorem</u>): Given Hypotheses 1, 2, 3, and 4, the diagram

$$\begin{array}{ccc} \underline{L} & \xrightarrow{\text{extend}(\varkappa,\underline{L})} & \underline{K} \\ \varphi \big\downarrow & & \big\downarrow \psi \\ \underline{S} & \xleftarrow{\ \ \delta\ \ } & \underline{R} \end{array}$$

commutes.

<u>Proof</u>. What we have to prove is (i) that the diagram commutes for the generating set X and (ii) that δ is indeed a homomorphism of Ω-algebras (we already know that φ , ψ , and $\text{extend}(\varkappa, \underline{L})$ are). But (i) is just part of Hypothesis 3; only (ii) remains.

To prove (ii) we assume, for arbitrary $\omega \in \Omega$ of arity k , that $\delta: R_j \mapsto S_j \quad (1 \leq j \leq k)$, and endeavor to show:

$$(*) \qquad \delta: \underline{R}_\omega(R_1, \ldots, R_k) \mapsto \underline{S}_\omega(S_1, \ldots, S_k) \quad .$$

As it turns out, in order to prove (*) we need additionally that the property which Hypothesis 4 claims for the generating set of $\underline{R}$ holds for all $R \in |\underline{R}|$, namely:

$$(**) \qquad \Theta;R = \Theta;R;\Theta^{-1};\Theta \quad .$$

Hypothesis 4 gives us the base for our inductive proof of this property. The induction step can proceed simultaneously with the proof of (*); that is, we may additionally assume that (**) holds for each of $R_1,\ldots,R_k$, provided we can then additionally prove that it holds for $\underline{R}_\omega(R_1,\ldots,R_k)$.

As we recall from Chapter VI, we have:

$$\underline{R}_\omega(R_1,\ldots,R_k) \;=\; (ac(o'_\omega(R_1,\ldots,R_k))) \upharpoonright (A \times M')$$

where o'_ω is the construction on relations derived from $\tilde{p}_\omega \times 1_{M'}$ and $(ac(\varkappa(\underline{\mathcal{C}\omega}) \otimes \underline{\mathfrak{M}}')) \upharpoonright (A \times M')$. If we make the abbreviation:

$$R_0 =_{df} (ac(\varkappa(\underline{\mathcal{C}\omega}) \otimes \underline{\mathfrak{M}}')) \upharpoonright (A \times M')$$

then we may diagram this instance of $\underline{R}_\omega$ as:

$$\begin{array}{ccccccc}
R_0 & & R_1 & \cdots & R_k \\
\searrow{\scriptstyle i'_0} & & \downarrow{\scriptstyle i'_1} & \cdots & \swarrow{\scriptstyle i'_k} \\
 & & R_0 \oplus R_1 \oplus \cdots \oplus R_k & & \\
 & & \downarrow{\scriptstyle (\upharpoonright (A \times M')) \circ ac \circ p'} & & \\
 & & \underline{R}_\omega(R_1,\ldots,R_k) & &
\end{array}$$

where the projection p' is in fact the kernel of o'_ω :

$$p' \;=\; \tilde{p}_\omega \times 1_{M'} \quad .$$

We write as usual for the composite of p' with an injection:

$$p'_j =_{df} p' \circ i'_j$$

and we define:

$$R_j^* =_{df} p_j'^{-1} ; R_j ; p_j'$$

so that we can write:

$$\underline{R}_\omega(R_1, \ldots, R_k) = \left(\bigcup_{i \geq 0} \left(\bigcup_{0 \leq j \leq k} R_j^* \right)^{(i)} \right) \rceil (A \times M') \quad .$$

Similarly we may write:

$$\underline{S}_\omega(S_1, \ldots, S_k) = \left(\bigcup_{i \geq 0} \left(\bigcup_{0 \leq j \leq k} S_j^* \right)^{(i)} \right) \rceil (A \times M) \quad ,$$

with $S_0 = ac(\underline{\mathcal{C}\omega} \otimes \underline{\mathcal{M}}) \rceil (A \times M)$, $S_j^* = p_j^{-1} ; S_j ; p_j$, and p_j the j-th component of $p =_{df} \tilde{p}_\omega \times 1_M$.

Hypothesis 3, in its application to $\underline{\mathcal{C}\omega}$, simply tells us that $\delta : R_0 \mapsto S_0$, and Hypothesis 4 in its application to $\underline{\mathcal{C}\omega}$ simply asserts (**) for R_0 ; hence we see that we have completely assimilated R_0 to the other R_j's , and need not give it any further special treatment.

For conciseness in what follows, we make the abbreviations:

$$U =_{df} \bigcup_{0 \leq j \leq k} R_j^*$$

and

$$W =_{df} \bigcup_{i \geq 0} U^{(i)} \quad ,$$

and we also give names to two more inclusion mappings:

$$\eta : A \subseteq B \quad ,$$

and

$$\nu =_{df} 1_B \times \mu : B \times M \subseteq B \times M' \quad ,$$

so that we have

$$\underline{R}_{\omega}(R_1,\ldots,R_k) = W \rfloor (A \times M) = (\eta \times 1_{M'});W;(\eta^{-1} \times 1_{M'}) \ .$$

The crucial part of the proof consists in showing that an analogue of (**) holds for W, namely:

$$(***) \qquad \nu;W;\nu^{-1};\nu = \nu;W \ .$$

It is easy to show the desired property for the individual R_j^* $(0 \le j \le k)$:

$$\begin{aligned} \nu;R_j^*;\nu^{-1};\nu &= \nu;p_j'^{-1};R_j;p_j';\nu^{-1};\nu \\ &= p_j^{-1};\theta;R_j;\theta^{-1};\theta;p_j' \\ &= p_j^{-1};\theta;R_j;p_j' \qquad \text{(by (**))} \\ &= \nu;R_j^* \ . \end{aligned}$$

The property trivially distributes over union to give:

$$\nu;U;\nu^{-1};\nu = \nu;U \ ,$$

and then we may calculate, for each $i \ge 0$:

$$\nu;U^{(i)};\nu^{-1};\nu = \nu;(U;\nu^{-1};\nu)^{(i)}$$

(We are entitled to insert $(\nu^{-1};\nu)$'s progressively from left to right.)

$$= \nu;U^{(i)} \ ,$$

(We may knock out all the $(\nu^{-1};\nu)$'s , including the last one, from right to left.)

and uniting over all values of i , we have as desired:

$$\nu;W;\nu^{-1};\nu \;=\; \nu;W \quad .$$

Having proved (***), we can immediately pay off our debt to the inductive hypothesis by showing that (**) holds for $\underline{R}_\omega(R_1,\ldots,R_k)$, that is:

$$\begin{aligned}
\Theta;\underline{R}_\omega(R_1,\ldots,R_k);\Theta^{-1};\Theta &= \Theta;(\eta\times 1_{M'});W;(\eta^{-1}\times 1_{M'});\Theta^{-1};\Theta \\
&= (\eta\times 1_M);\nu;W;\nu^{-1};\nu;(\eta^{-1}\times 1_{M'}) \\
&= (\eta\times 1_M);\nu;W;(\eta^{-1}\times 1_{M'}) \qquad \text{(by (***))} \\
&= \Theta;\underline{R}_\omega(R_1,\ldots,R_k) \quad .
\end{aligned}$$

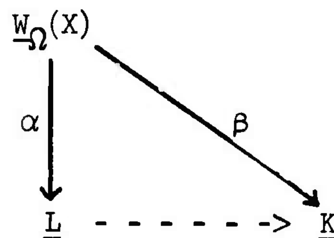
Now at last we can compute:

$$\begin{aligned}
\delta(\underline{R}_\omega(R_1,\ldots,R_k)) &= W \upharpoonright (A\times M') \upharpoonright (A\times M) \\
&= W \upharpoonright (A\times M) \\
&= W \upharpoonright (B\times M) \upharpoonright (A\times M) \\
&= [\nu;\bigcup_{i\geq 0} U^{(i)};\nu^{-1}] \upharpoonright (A\times M) \\
&= \left[\bigcup_{i\geq 0} (\nu;U;\nu^{-1})^{(i)}\right] \upharpoonright (A\times M) \qquad \text{(inserting } (\nu^{-1};\nu)\text{'s from left to right, as before)} \\
&= \left[\bigcup_{i\geq 0}\left(\bigcup_{0\leq j\leq k} (R_j^* \upharpoonright (B\times M))\right)^{(i)}\right] \upharpoonright (A\times M) \\
&= \left[\bigcup_{i\geq 0}\left(\bigcup_{0\leq j\leq k} p_j^{-1};(R_j \upharpoonright (A\times M));p_j\right)^{(i)}\right] \upharpoonright (A\times M) \\
&= \underline{S}_\omega(\delta(R_1),\ldots,\delta(R_k)) \quad .
\end{aligned}$$

We have proved (*), hence the theorem. ∎

A typical application of what has been proved in this chapter will of course be to a situation in which $\underline{L}$ is the target language corresponding to some source language, $\underline{N}$ say, and in which we have already a correct compiler $\lambda: \underline{N} \to \underline{L}$, in addition to κ satisfying Hypotheses 1-4. We will then be able to assert the correctness of $\text{extend}(\kappa,\underline{L}) \circ \lambda: \underline{N} \to \underline{K}$. Hence the name, "compiler composition theorem".

Remark: It is not strictly true that the homomorphism $\text{extend}(\kappa,\underline{L})$ must always exist. What is required is that in the diagram of homomorphisms



where α extends the insertion $: X \subseteq |\underline{L}|$ and β extends the function $\kappa \upharpoonright X$, the dotted arrow should exist; that is, the homomorphism β should factor as

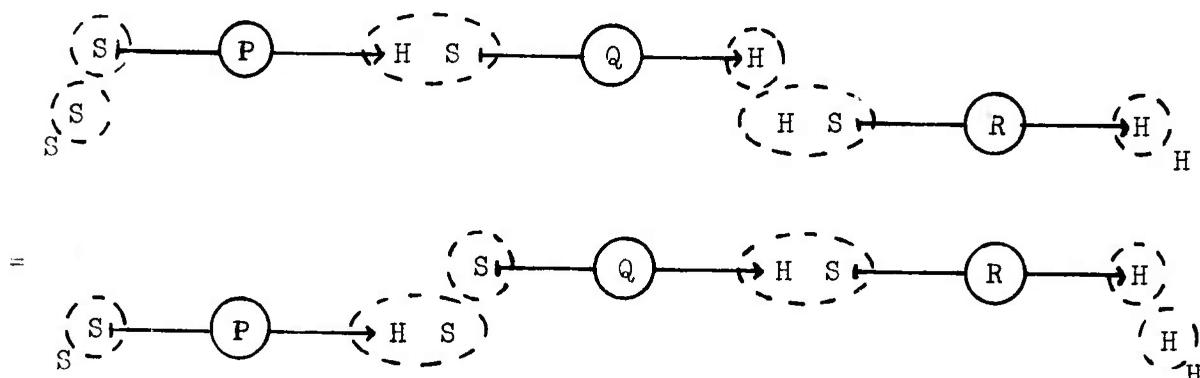
$$\beta = \text{extend}(\kappa,\underline{L}) \circ \alpha \quad .$$

This is to say that for any $w_1, w_2 \in |\underline{W}_\Omega(X)|$ we must have:

$$\alpha(w_1) = \alpha(w_2) \quad \underline{\text{implies}} \quad \beta(w_1) = \beta(w_2) \quad ,$$

or, less formally, that all equations (of terms built up from elements of X with the operators in Ω) which hold in $\underline{L}$ must also hold in $\underline{K}$ (with, for each $\underline{\rho}_i \in X$, the substitution of $\kappa(\underline{\rho}_i)$ for $\underline{\rho}_i$).

This will be the case if the various $\underline{\rho}_i$ and $\underline{\mathcal{C}\omega}$ are all sufficiently distinct so that all the equations which hold in $\underline{L}$ do so by virtue of the structure of the kernels $\tilde{p}_{\omega}$ (which determine the operations in both $\underline{L}$ and $\underline{K}$), a condition which it does not appear difficult to ensure in practice when selecting a generating set for $\underline{L}$. For example, the equations which are instances of the associative law in $\widehat{\underline{SS}}$, and which we may depict as follows:



are of this sort (although it may require some care in the precise specification of how disjoint union is to act on sets to guarantee that they actually do hold).

VIII. The General Plan for Simple Proofs of Compiler Correctness

Chapters IX, X, and XI will consist of example proofs of compiler correctness. These will be simple in the sense that each compiler will be for a language with only a single feature; hence there will be as yet (but see Chapter XII, in which these results will be combined) no compounding of compilers, nor any appeal to the compiler composition theorem. The purpose of the present chapter is to set forth the schema which all these proofs will follow, and to introduce some uniform abbreviations in order to make the formulas less tedious.

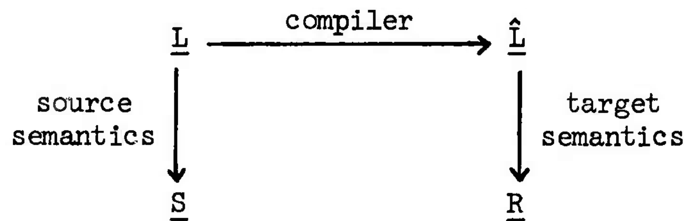
Given a source language (Ω-algebra) $\underline{L}$, a semantic homomorphism φ to an Ω-algebra $\underline{S}$ of meanings (in general of relations, but usually of at least partial functions), and a compiling homomorphism κ to a target language $\hat{\underline{L}}$ which is an Ω-algebra of BRAs with constructions as its operations (we shall always take the algebra $\hat{\underline{L}}$ to be exactly the image of $\underline{L}$ under κ), we may list as follows the steps which must be taken to prove the compiler correct:

(i) Specify the target machine -- i.e., the BRA $\underline{\mathfrak{M}}$ on which compiled programs are to run. This will, by the result of Chapter VI, determine the target semantic homomorphism ψ with the effect

$$\psi: \underline{\rho} \to (\mathrm{ac}(\underline{\rho} \otimes \underline{\mathfrak{M}})) \restriction D \quad ,$$

where the domain D is the Cartesian product of the set of distinguished nodes of $\hat{\underline{L}}$-programs with the carrier of $\underline{\mathfrak{M}}$. The image of $\hat{\underline{L}}$ under ψ is $\underline{R}$, the Ω-algebra of relations computed by $\hat{\underline{L}}$-programs; the elements of $\underline{R}$ are reflexive and transitive relations $: D \nrightarrow D$.

From step (i) we have a three-sided diagram:



What remains is to supply a fourth side for the diagram and prove its commutativity.

(ii) Specify a "decoding" function $\delta: \underline{R} \to \underline{S}$ from relations computed to source meanings.

(iii) Prove δ a homomorphism.

(iv) Prove that the resulting closed diagram commutes for a generating set of the source language.

We may then -- after completing steps (i - iv) -- conclude, by Part II of the unique extension lemma, that the diagram commutes for the whole source language; i.e., that the compiler is correct.

In choosing δ , we shall demand more than just any arbitrary homomorphism $: \underline{R} \to \underline{S}$; we want one which will tell us how to use a compiled program to do the work of the corresponding source program. We may suppose that $\underline{S}$ has for its carrier a set of relations of some type, say a subset of $(E_1 \nrightarrow E_2)$ for some sets E_1 and E_2 . The carrier of $\underline{R}$, the Ω-algebra of relations computed by target language programs, is a subset of $(D \nrightarrow D)$. What we want then is to specify δ by a pair of mappings $\langle e,d \rangle$ where $e: E_1 \to D$ tells us how to "encode", that is to choose an initial state for the compiled program and its

machine, given the argument of the source program, and $d: D \xrightarrow{\sim} E_2$ tells us how to "decode" -- what sense to make of a state in which the compiled program halts (d is partial because halt states are in general a proper subset of D). The effect of δ is then of course $\delta: r \mapsto d \circ r \circ e$, or equivalently if r is a function, $\delta(r)(b) = d(r(e(b)))$. To prove δ a homomorphism will be to prove for all $\omega \in \Omega$ and all $r_1, \ldots, r_k \in |\underline{R}|$ that

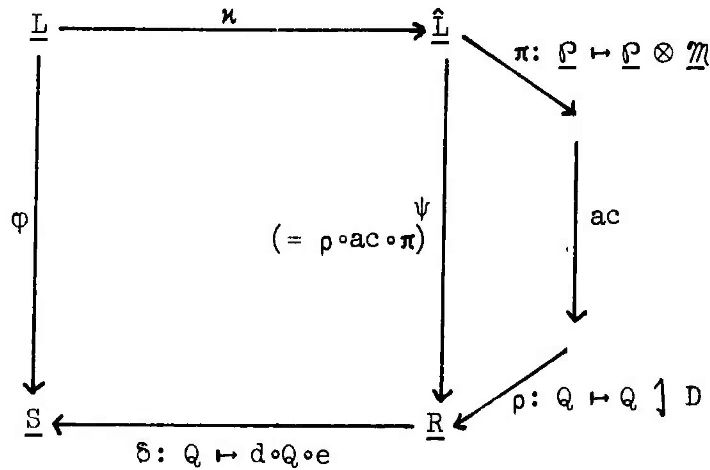
$$\underline{S}_\omega(d \circ r_1 \circ e, \ldots, d \circ r_k \circ e) = d \circ \underline{R}_\omega(r_1, \ldots, r_k) \circ e \quad .$$

(It seems likely that for some proofs of compiler correctness, more advanced than any in the present work, it may be necessary to allow that d and e be general relations. Indeed they might be quite complicated relations, so that it would become a subproblem of the compiler verification to demonstrate the correctness of a method for computing d and e .)

For some source languages it will be the case that the result of a source program does not depend on any argument, so that it would be most natural to take the set of meanings (i.e., the carrier of $\underline{S}$) as some set E not a space of relations. In such a case, computations of target language programs will start always from some fixed element of D . For uniformity's sake it will be most convenient to pick out this initial computation state by means of a function $d: \underline{1} \rightarrow D$. Therefore we shall somewhat artificially enforce that source meanings are relations by choosing for $|\underline{S}|$ either $(\underline{1} \rightarrow E)$ or $(\underline{1} \xrightarrow{\sim} E)$. The former choice is, of course, isomorphic to E ; the latter, which contains additionally the empty partial function, provides a convenient alternative to enlarging E by an artificial "undefined" element in cases where our intuitive idea

of the meaning function is that it is partial, although we are compelled to define a total function to be the semantic homomorphism.

Here in detail is the diagram whose commutativity is the correctness of the compiler $\varkappa$:



The vertices of the diagram are all Ω-algebras; the arrows are (or, in the case of δ , must be proved to be) homomorphisms. Although in the correctness proofs to follow, the source and target languages will have their own specific names, the letters $\varkappa$, φ , ψ , π , $\mathfrak{M}$, ρ , D , $\underline{R}$, $\underline{S}$, δ , d , and e will be used without remark to indicate the entities pictured here.

An additional notational convention: recalling that each operation $\underline{R}_\omega$ is the composite of restriction to D with accessability with a construction, we shall occasionally write $\underline{R}_\omega^-$ for the construction, so that we have

$$\underline{R}_\omega = (\rfloor D) \circ ac \circ \underline{R}_\omega^- \quad .$$

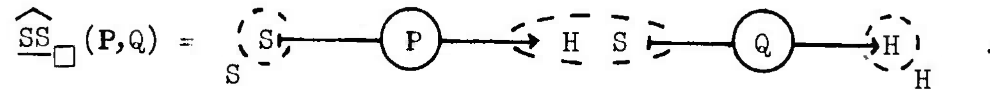
IX. Proofs for Examples SS, BE, AE

Following the plan laid out in Chapter VIII, the present chapter will prove the correctness of the compilers defined in Chapter V for the languages SS, BE, and AE.

Example SS: We recall that the language SS is the free semigroup generated by a set (of "commands") X . We use □ for the operator denoting concatenation in SS . $\widehat{SS}$ is the semigroup (i.e., the associative {□}-algebra) of X-BRAs generated from the image of the set X under the action of the compiler ϰ , namely for each f∈X ,

$$\varkappa : f \mapsto \underset{H}{\overset{S}{\downarrow}} f \quad ,$$

by the operation $\widehat{SS}_{\square}$, for which we have the construction diagram:



(It may be supposed that we always take for result of $\widehat{SS}_{\square}$ a standard representative from the appropriate isomorphism class, e.g. a BRA with carrier {S,1,2,3,...,H} , in order to make $\widehat{SS}_{\square}$ an associative operation.)

We recall further that we assume an interpretation $i: X \to (A \nrightarrow A)$ of the generators of SS as relations on a set A , and that we take φ_i , the semantic homomorphism, to be the unique extension of i to a homomorphism : SS → S , where S is the semigroup of relations on A to itself under the operation of relational composition

$$S_{\square} = ; \quad .$$

We have now to specify the machine, that is the BRA, on which compiled programs are to be run. For this we can take i itself, since i is an X-BRA, with $|i| = A$.

The set of distinguished nodes of $\widehat{\underline{SS}}$-programs is $\{S,H\}$; therefore for ψ_i , the semantic homomorphism $: \widehat{\underline{SS}} \to \underline{R}$, we have:

$$\psi_i : \underline{\rho} \mapsto (ac(\underline{\rho} \otimes i) \rceil \{S,H\} \times A \quad .$$

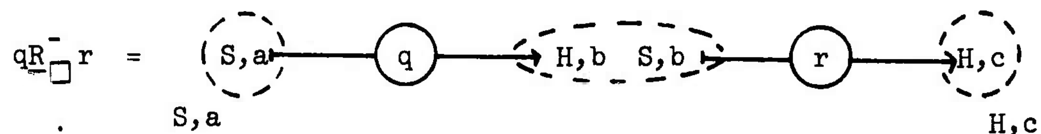
The carrier of $\underline{R}$ is the image of $|\widehat{\underline{SS}}|$ under ψ_i . D is the set $\{S,H\} \times A$; the elements of $\underline{R}$ are reflexive and transitive relations $: D \nrightarrow D$.

No relation in $\underline{R}$ has any instance of the form $H,a \mapsto S,b$, nor, except as required by reflexivity, of the form $X,a \mapsto X,b$. This follows from the fact that the BRAs in $\widehat{\underline{SS}}$ have no edges either arriving at S or departing from H .

We have the decomposition

$$\underline{R}_{\square} = (\rceil D) \circ ac \circ \underline{R}^{-}_{\square}$$

where $\underline{R}^{-}_{\square}$ is a construction on relations which may be represented by the family of construction diagrams, one for every $a,b,c \in A$:



Letting $\bullet$ denote the undistinguished node created by $\widehat{\underline{SS}}_{\square}$, we may express $\underline{R}^{-}_{\square}$ explicitly by:

$$q \, \underline{R}^{-}_{\square} \, r \;=\; 1_{\{S,\bullet,H\} \times A} \;\cup\; \{S, a \mapsto \bullet, b \mid q\colon S,a \mapsto H,b\}$$

$$\cup\; \{\bullet, b \mapsto H, c \mid r\colon S,b \mapsto H,c\} \quad .$$

Then

$$ac(q \, \underline{R}^{-}_{\square} \, r) \;=\; (q \, \underline{R}^{-}_{\square} \, r) \cup \{S,a \mapsto H,c \mid q\colon S,a \mapsto H,b \ \underline{\text{and}}\ r\colon S,b \mapsto H,c\} \quad ,$$

and

$$q \, \underline{R}_{\square} \, r \;=\; (ac(q \, \underline{R}^{-}_{\square} \, r)) \rceil D$$

$$= \; 1_D \cup \{S,a \mapsto H,c \mid q\colon S,a \mapsto H,b \ \underline{\text{and}}\ r\colon S,b \mapsto H,c\}$$

$$= \; 1_D \cup \{S,a \mapsto S,a\} \, ; q \, ; \{H,b \mapsto S,b\} \, ; r \, ; \{H,c \mapsto H,c\} \quad .$$

Define $e\colon A \to D$ and $d\colon D \,\tilde{\to}\, A$ by:

$$e\colon a \mapsto S,a \quad \text{and} \quad d\colon H,a \mapsto a \quad ,$$

and define $\delta\colon \underline{R} \to \underline{S}$ as always by

$$\delta\colon r \mapsto e;r;d \quad .$$

To prove δ a homomorphism, we calculate:

$$\delta(q \, \underline{R}_{\square} \, r) \;=\; e;(q \, \underline{R}_{\square} \, r);d$$

$$= \; \{a \mapsto S,a\};\{S,a \mapsto S,a\};q;\{H,b \mapsto S,b\};r;\{H,c \mapsto H,c\};\{H,c \mapsto c\}$$

$$= \; \{a \mapsto S,a\};q;\{H,b \mapsto b\};\{b \mapsto S,b\};r;\{H,c \mapsto c\}$$

$$= \; e;q;d;e;r;d$$

$$= \; \delta(q) \, \underline{S}_{\square} \, \delta(r) \quad .$$

To check $\varphi_i(f) = \delta \circ \psi_i \circ \varkappa(f)$ for f in the generating set X we compute:

$$f \xmapsto{\varphi_i} i(f)$$

and

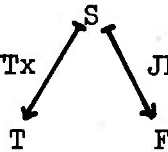
$$f \xmapsto{\varkappa} \begin{matrix} S \\ \downarrow f \\ H \end{matrix}$$

$$\xmapsto{\psi_i} 1_D \cup (\{S \mapsto H\} \times i(f))$$

$$\xmapsto{\delta} \{a \mapsto S,a\};(\{S \mapsto H\} \times i(f));\{H,a \mapsto a\}$$

$$= \quad i(f) \quad .$$

This completes the proof of the correctness of $\varkappa$. ∎

<u>Example BE</u>: For example BE we have $\underline{BE} = W_{\{\neg, \wedge, \vee, \supset\}}(X)$ ($\neg$ is unary, the others binary), and $\widehat{\underline{BE}}$ generated from $\{\varkappa(x) \mid x \in X\}$, where for $x \in X$, $\varkappa: x \to$ 

, with the $\widehat{\underline{BE}}$-operations as given by the construction diagrams in Chapter V (p. 44). $\underline{BE}$, then, is an algebra of $\{JTx, JFx, JTy, JFy, \ldots\}$-BRAs , where $\{x,y,\ldots\} = X$.

For the semantics of $\underline{BE}$ we confine ourselves at present to the three-valued case, and (letting $\underline{2}$ denote the set $\{\underline{true}, \underline{false}\}$), we assume an interpretation

$$i: X \tilde{\rightarrow} \underline{2} \quad .$$

As explained in Chapter VIII, we take for the carrier of $\underline{S}$ the three partial functions $: \underline{1} \tilde{\rightarrow} \underline{2}$, which we name ff ($: 0 \mapsto \underline{false}$) , tt ($: 0 \mapsto \underline{true}$) , and uu (the empty partial function). We take the operations in $\underline{S}$ to be given by McCarthy's truth tables (Chapter IV, p. 33).

If we define the isomorphism $k: X \rightarrow (\underline{1} \rightarrow X)$ by

$$k(x) : 0 \mapsto x \quad ,$$

then we may define the semantic homomorphism φ_i to be the unique homomorphism $: \underline{BE} \rightarrow \underline{S}$ agreeing on X with

$$\varphi_i(x) = i \circ (k(x)) \quad .$$

We shall choose a target machine $\underline{\mathfrak{m}i}$ dependent on the interpretation. Its task will be to allow only those "jumps" to be executed which conform to the facts as represented by i ; since the facts do not change in the course of execution, our machine need only have one state. Thus we take

$$\underline{\mathfrak{m}i} : \{JTx, JFx, \ldots\} \rightarrow (\underline{1} \tilde{\rightarrow} \underline{1}) \quad ,$$

with, for $x \in X$,

$$\underline{\mathfrak{m}i}(\{ {JTx \atop JFx} \}) : 0 \mapsto 0 \ \underline{\text{iff}} \ i : x \mapsto \{ {\underline{true} \atop \underline{false}} \} \quad ,$$

that is,

$$\underline{\text{iff}} \ \varphi_i : x \mapsto \{ {tt \atop ff} \} \quad .$$

For each compiled program $\underline{\rho} \in |\widehat{\underline{BE}}|$ we have $|\underline{\rho} \otimes \underline{\mathcal{M}i}| = |\underline{\rho}| \times \underline{1} \cong |\underline{\rho}|$; for notational convenience we shall pretend that the isomorphism is an equality; that is, we shall write S instead of $\langle S,0 \rangle$ and so on.

For the target semantic homomorphism, $\psi_i : \widehat{\underline{BE}} \to \underline{R}$, we have

$$\psi_i : \underline{\rho} \mapsto ac(\underline{\rho} \otimes \underline{\mathcal{M}i}) \rceil \{S,T,F\} \quad .$$

It is readily verified that for any $\underline{\rho} \in |\widehat{\underline{BE}}|$, $\underline{\rho} \otimes \underline{\mathcal{M}i}$ is monogenic, and that Proposition III.1 applies to give:

$\underline{\rho} \otimes \underline{\mathcal{M}i}$ computes a partial function from $\{S\}$ to $\{T,F\}$.

Since $|\underline{R}|$ is just the image of $|\widehat{\underline{BE}}|$ under ψ_i , this says that for any $Q \in |\underline{R}|$, we have at most one of

$$Q : S \mapsto T$$

and

$$Q : S \mapsto F \quad .$$

Hence defining

$$e : 0 \mapsto S \quad , \quad d : \left\{ \begin{matrix} T \mapsto \underline{true} \\ F \mapsto \underline{false} \end{matrix} \right\} \quad , \quad \delta : Q \mapsto d \circ Q \circ e$$

makes $\delta : |\underline{R}| \to |\underline{S}|$ well defined; we may calculate its effect as:

$\delta : Q \mapsto tt$ <u>iff</u> $Q : S \mapsto T$,

$\delta : Q \mapsto ff$ <u>iff</u> $Q : S \mapsto F$,

$\delta : Q \mapsto uu$ otherwise .

Proving δ a homomorphism is a matter of details. It is immediate from the construction diagram for $\widehat{\underline{BE}}_\neg$ that $\underline{R}_\neg(Q) : S \mapsto T$ <u>iff</u> $Q : S \mapsto F$

and that $\underline{R}_{\neg}(Q): S \to F$ iff $Q: S \to T$; from this it follows that δ is a homomorphism of $\neg$. We consider $\wedge$, $\vee$, $\supset$ in parallel, since they are isomorphic under suitable interchanging of truth values. By considering the construction diagrams we perceive that

$$\underline{R}_{\left\{\begin{matrix}\wedge\\ \vee\\ \supset\end{matrix}\right\}}(P,Q) : S \mapsto \left\{\begin{matrix}T\\ F\\ F\end{matrix}\right\} \ \underline{\text{iff}} \left(P: S \mapsto \left\{\begin{matrix}T\\ F\\ T\end{matrix}\right\} \underline{\text{and}}\ Q: S \mapsto \left\{\begin{matrix}T\\ F\\ F\end{matrix}\right\}\right)$$

and that

$$\underline{R}_{\left\{\begin{matrix}\wedge\\ \vee\\ \supset\end{matrix}\right\}}(P,Q) : S \mapsto \left\{\begin{matrix}F\\ T\\ T\end{matrix}\right\} \ \underline{\text{iff}}\quad P: S \mapsto \left\{\begin{matrix}F\\ T\\ F\end{matrix}\right\} \underline{\text{or}} \left(P: S \mapsto \left\{\begin{matrix}T\\ F\\ T\end{matrix}\right\} \underline{\text{and}}\ Q: S \mapsto \left\{\begin{matrix}F\\ T\\ T\end{matrix}\right\}\right).$$

If neither of the iff conditions is met, the result of the operation must be the identity relation $1_{\{S,T,F\}}$. Applying what we know about δ , we may restate the above results:

$$\delta(P\ \underline{R}_{\left\{\begin{matrix}\wedge\\ \vee\\ \supset\end{matrix}\right\}}\ Q) = \left\{\begin{matrix}tt\\ ff\\ tt\end{matrix}\right\} \underline{\text{iff}}\ \delta(P) = \left\{\begin{matrix}tt\\ ff\\ tt\end{matrix}\right\} \underline{\text{and}}\ \delta(Q) = \left\{\begin{matrix}tt\\ ff\\ ff\end{matrix}\right\} ;$$

$$\delta(P\ \underline{R}_{\left\{\begin{matrix}\wedge\\ \vee\\ \supset\end{matrix}\right\}}\ Q) = \left\{\begin{matrix}ff\\ tt\\ tt\end{matrix}\right\} \underline{\text{iff}}\ \delta(P) = \left\{\begin{matrix}ff\\ tt\\ ff\end{matrix}\right\} \underline{\text{or}} \left(\delta(P) = \left\{\begin{matrix}tt\\ ff\\ tt\end{matrix}\right\} \underline{\text{and}}\ \delta(Q) = \left\{\begin{matrix}ff\\ tt\\ tt\end{matrix}\right\}\right) ;$$

$$\delta(P\ \underline{R}_{\left\{\begin{matrix}\wedge\\ \vee\\ \supset\end{matrix}\right\}}\ Q) = uu \quad \text{otherwise, that is } \underline{\text{iff}}$$

$$\delta(P) = uu\ \underline{\text{or}} \left(\delta(P) = \left\{\begin{matrix}tt\\ ff\\ tt\end{matrix}\right\} \underline{\text{and}}\ \delta(Q) = uu\right) .$$

Comparison with the tables (p. 33) for the S-operations shows that what we have just obtained can be summarized precisely by

$$\delta(P \; \underline{R}_{\left\{\begin{smallmatrix}\wedge\\ \vee\\ \supset\end{smallmatrix}\right\}} Q) = \delta(P) \; \underline{S}_{\left\{\begin{smallmatrix}\wedge\\ \vee\\ \supset\end{smallmatrix}\right\}} \delta(Q) \quad ;$$

that is, we have proven δ a homomorphism.

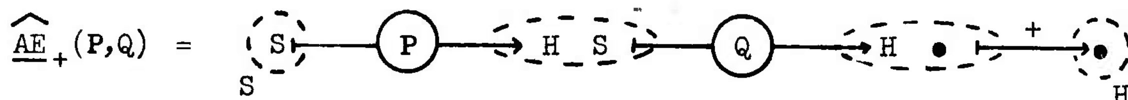
The final step, to check commutativity for an arbitrary $x \in X$, is as usual trivial. We have the three cases $i: x \mapsto$ true, $i: x \mapsto$ false, i not defined at x; these yield respectively $\varphi_i: x \mapsto tt$, $\varphi_i: x \mapsto ff$, $\varphi_i: x \mapsto uu$. In any case we have both $\varkappa(x)_{JTx}: S \mapsto T$ and $\varkappa(x)_{JFx}: S \to F$, but in $\varkappa(x) \otimes \underline{\mathfrak{M}i}$ we have respectively only the first, only the second, and neither of these. It follows on computing the (essentially null) effects of ac, ρ, and δ that we get in the three cases of $i(x)$ respectively $\delta \circ \psi \circ \varkappa: x \mapsto tt$, $\delta \circ \psi \circ \varkappa: x \mapsto ff$, and $\delta \circ \psi \circ \varkappa: x \mapsto uu$; that is, we have commutativity of the diagram, and we are done. ∎

Example BE -- 2-valued semantics: If i is required to be total, the foregoing proof is not affected. It only needs to be checked, as has often been done, that McCarthy's truth functions, restricted to {true, false}, are the classical not, and, or, implies. ∎

Example AE: This we recall is the general case of a word algebra as source language, $\underline{AE} = W_\Omega(X)$, with the compiler producing "Polish postfix code" for a stack machine. For notational simplicity we assume a single binary operator, say $\Omega = \{+\}$, but it will be seen that the

proof is applicable to any number of operators of arbitrary non-negative arity. We have, for $x \in X$, $\varkappa: x \mapsto \underset{H}{\overset{S}{\Big\downarrow}} Lx$, and the operation in $\widehat{\underline{AE}}$ is given by:

$\widehat{\underline{AE}}_+(P,Q)$ =



$\widehat{\underline{AE}}$ is generated by the set $\{\varkappa(x) \mid x \in X\}$.

For the semantics of $\underline{AE}$ we may take the meaning of " + " as given by any $\{+\}$-algebra $\underline{A}$, with carrier some set (of "arithmetic values") A , and we assume an interpretation of the variables $i: X \to A$. We take $\underline{S}$ to be an algebra isomorphic to $\underline{A}$, but with carrier $(\underline{1} \to A)$, and operation

$$\underline{S}_+(s_1,s_2) : 0 \mapsto \underline{A}_+(s_1(0),s_2(0)) \quad .$$

We may explicitly define an isomorphism $h: \underline{S} \to \underline{A}$ by

$$h: s \mapsto s(0) \quad .$$

The semantic homomorphism φ_i may then be defined as the unique homomorphism : $\underline{AE} \to \underline{S}$ such that

$$\varphi \restriction X = h^{-1} \circ i \quad .$$

We now construct our machine $\underline{\mathfrak{M}i}$. Its states will be "stacks", i.e., finite sequences of elements of A . Denoting the set of all such sequences by A^* , we will have:

$$\underline{\mathfrak{M}i} : \{+,Lx,Ly,\dots\} \to (A^* \stackrel{\sim}{\to} A^*) \quad .$$

The effect of any of the various operations $\underline{\mathfrak{m}i}_{Lx}$ will of course be to extend the current stack by one element, and the effect of $\underline{\mathfrak{m}i}_{+}$ will be to replace the top two elements by a result (for a k-ary operation this would be the top k elements); the precise effect of the $\underline{\mathfrak{m}i}$-operations must naturally be chosen to reflect the semantics of $\underline{AE}$ as follows:

For $x \in X$, $\underline{\mathfrak{m}i}_{Lx} : \langle a_1 \ldots a_m \rangle \mapsto \langle i(x), a_1, \ldots, a_m \rangle$

$(m \geq 0$; i.e., $\underline{\mathfrak{m}i}_{Lx}$ is total) ;

$\underline{\mathfrak{m}i}_{+} : \langle a_1, a_2, a_3, \ldots, a_m \rangle \mapsto \langle \underline{A}_{+}(a_2, a_1), a_3, \ldots, a_m \rangle$

$(m \geq 2$; i.e., $\underline{\mathfrak{m}i}$ is not defined on empty or unit stacks).

Our choice of $\underline{\mathfrak{m}i}$ and the set $\{S,H\}$ of distinguished nodes for $\widehat{\underline{AE}}$ gives for the homomorphism $\psi_i : \widehat{\underline{AE}} \to \underline{R}$

$$\psi_i : \underline{\wp} \mapsto (ac(\underline{\wp} \otimes \underline{\mathfrak{m}i})) \rceil (\{S,H\} \times A^*) \quad .$$

$|\underline{R}|$ is the image of $|\widehat{\underline{AE}}|$ under ψ_i .

For any $\underline{\wp} \in |\widehat{\underline{AE}}|$, the product $\underline{\wp} \otimes \underline{\mathfrak{m}i}$ is monogenic, and Proposition III.1 applies to give:

$\underline{\wp} \otimes \underline{\mathfrak{m}i}$ computes a partial function from $\{S\} \times A^*$ to $\{H\} \times A^*$.

This fact and the construction diagram for $\underline{AE}_{+}$ enable us to give an explicit expression for $\underline{R}_{+}$:

$$\underline{R}_{+}(P,Q) = 1_{\{S,H\} \times A^*} \cup P;(\{H \mapsto S\} \times 1_{A^*});Q;(\{H \mapsto H\} \times \underline{\mathfrak{m}i}_{+}) \quad .$$

We may define the decoding function δ by

$$e: 0 \mapsto \langle S, \langle\ \rangle\rangle \qquad (\langle\ \rangle \text{ is the empty stack}),$$

$$d: \langle H, \langle a\rangle\rangle \mapsto a \qquad (\langle a\rangle \text{ is any one-element stack}),$$

$$\delta: Q \mapsto d \circ Q \circ e \quad .$$

We encounter an interesting difficulty, however, when we try to prove δ a homomorphism: that $\delta: P \mapsto s_1$, $\delta: Q \mapsto s_2$ does not by itself suffice to prove $\delta: \underline{R}_+(P,Q) \mapsto \underline{S}_+(s_1,s_2)$; the mapping δ throws away information which is in fact essential to the correctness of the compilation, namely that for $w \in |\underline{AE}|$, $\kappa(w) \otimes \underline{\mathfrak{M}i}$, started with <u>any</u> initial stack (not just the empty one) will halt with $\varphi_i(w)(0)$ adjoined to the top of that stack. Hence we must first prove inductively:

<u>Lemma</u>: For $w \in |\underline{AE}|$, and $Q = \psi_i \circ \kappa(w) \in |\underline{R}|$,

$$Q: \langle S, \langle a_1, a_2, \ldots, a_m\rangle\rangle \to \langle H, \langle b_0, b_1, \ldots, b_n\rangle\rangle$$

$$\underline{\text{iff}}\ \langle a_1, \ldots, a_m\rangle = \langle b_1, \ldots, b_n\rangle\ \underline{\text{and}}\ b_0 = \varphi_i(w)(0) \quad .$$

<u>Proof</u>: For $w \in X$ we have immediately, by $\kappa: w \mapsto \overset{S}{\underset{H}{\downarrow}} Lw$ and the construction of $\underline{\mathfrak{M}i}$,

$$Q: \langle S, \langle a_1, \ldots, a_m\rangle\rangle \mapsto \langle H, \langle i(w), a_1, \ldots, a_m\rangle\rangle$$

$$= \langle H, \langle \varphi_i(w)(0), a_1, \ldots, a_m\rangle\rangle \quad .$$

Now suppose the lemma holds for $u, v \in |\underline{AE}|$, with $P = \psi_i \circ \varkappa(u)$, $Q = \psi_i \circ \varkappa(v)$, and $w = \underline{AE}_+(u,v)$, so that $\psi_i \circ \varkappa(w) = \underline{R}_+(P,Q)$. Then we have:

$$\underline{R}_+(P,Q) : \langle S, \langle a_1, \ldots, a_m \rangle\rangle \overset{P}{\longmapsto} \langle H, \langle \varphi_i(u)(0), a_1, \ldots, a_m \rangle\rangle$$

$$\xrightarrow{\{H \mapsto S\} \times 1_{A^*}} \langle S, \langle \varphi_i(u)(0), a_1, \ldots, a_m \rangle\rangle$$

$$\overset{Q}{\longmapsto} \langle H, \langle \varphi_i(v)(0), \varphi_i(u)(0), a_1, \ldots, a_m \rangle\rangle$$

$$\xrightarrow{\{H \mapsto H\} \times \underline{\mathcal{M}i}_+} \langle H, \langle \underline{A}_+(\varphi_i(u)(0), \varphi_i(v)(0)), a_1, \ldots, a_m \rangle\rangle$$

$$= \langle H, \langle \varphi_i(w)(0), a_1, \ldots, a_m \rangle\rangle \quad ,$$

and our lemma is proved.

The lemma essentially completes our correctness proof; for $w \in |\underline{AE}|$ we have

$$\delta \circ \psi_i \circ \varkappa(w) = e; (\psi_i \circ \varkappa(w)); d$$

$$= \{0 \mapsto \langle S, \langle\ \rangle\rangle\}; (\psi_i \circ \varkappa(w)); \{\langle H, \langle a \rangle\rangle \mapsto a\} \quad .$$

Therefore,

$$\delta \circ \psi_i \circ \varkappa(w) : 0 \overset{e}{\mapsto} \langle S, \langle\ \rangle\rangle \xrightarrow{\psi_i \circ \varkappa(w)} \langle H, \langle \varphi_i(w)(0) \rangle\rangle \overset{d}{\mapsto} \varphi_i(w)(0) \quad ;$$

that is,

$$\delta \circ \psi_i \circ \varkappa = \varphi_i$$

(and it is straightforward to verify from this that δ is indeed a homomorphism). ∎

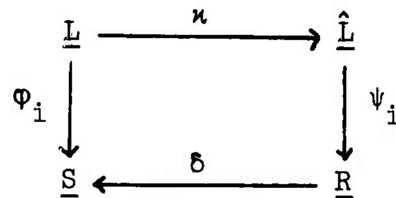
X. Stores and Assignment

In this chapter we will consider a simple form of the problem of languages with assignment. The simplifications we make to the problem are as follows: we suppose that our "variables" (in the programmer's sense) are both simple and static -- i.e., we neither consider arrays or structures, nor do we allow any declaration mechanism; all occurrences of the same identifier will refer to the same store location. We will also simplify the solution by confining ourselves narrowly to a "language" in which single assignment statements are the only programs; the compiler composition theorem will allow us to apply the result of this chapter to languages in which higher-order program structures are built up using assignment statements as constituents. Unfortunately, the compiler composition theorem will not be adequate to give us our assignment statement compiler on the assumption that we have already a compiler for right-hand-side expressions. We will indeed assume that the problem of compiling right-hand-sides has already been solved, but we will need for our proof some specific assumptions about the form of that solution which will become clear as we go on.

The present chapter will be divided into two parts: first, by assuming a target machine with just the operations we need, and by describing a trivial compiler, we will prove essentially the triviality that an assignment statement may be executed by first evaluating the right-hand-side and then storing its value; second, we will make the existence of this target machine more plausible by showing that it can be modified (and that a further compiler can be composed with the first

to obtain equivalent effects on the modified machine) in such a way that it factors into two components which are recognizable as a store and an arithmetic unit. Furthermore, the store component will be seen to factor into individual "location machines".

For the first part of our discussion, then, we assume the existence of an Ω-algebra $\underline{L}$ of "right-hand-side" expressions, generated by a set X of variables and taking values in a set A ; and we suppose that we have obtained a compiler κ which yields a family of commutative diagrams, one for each interpretation $i: X \to A$, as follows:



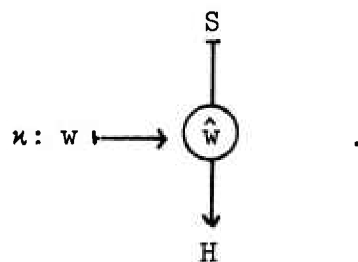
We may suppose that

$$|\underline{S}| = (\underline{1} \to A) \quad ,$$

and that each φ_i is the unique homomorphism satisfying, for all $x \in X$,

$$\varphi_i(x)(0) = i(x) \quad .$$

We further suppose that $\hat{\underline{L}}$ is an Ω-algebra of Γ-BRAs (for some set Γ) with set of distinguished nodes $\{S,H\}$; to indicate this fact we depict the effect of κ schematically by



Each ψ_i is determined by a machine (Γ-BRA) $\underline{\mathcal{M}i}$; we make the assumption that all the $\underline{\mathcal{M}i}$ have the same carrier M , so that we have:

$$\psi_i : \underline{\rho} \mapsto (ac(\underline{\rho} \otimes \underline{\mathcal{M}i})) \rfloor (\{S,H\} \times M) \quad .$$

The homomorphism δ is of course determined by functions $e: \underline{1} \rightarrow \{S,H\} \times M$ and $d: \{S,H\} \times M \tilde{\rightarrow} A$; we assume that

$$e: 0 \mapsto S,m_o$$

for some fixed initial state $m_o \in M$, and that d is defined only for arguments of the form H,m . We may write the effect of δ as

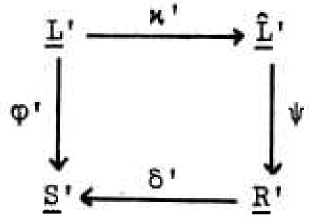
$$\delta: R \mapsto d \circ R(S,m_o) \quad .$$

Our hypothesis of commutativity now appears as:

$$[d \circ (\psi_i \circ \varkappa(w))](S,m_o) = \varphi_i(w)(0) \quad .$$

It appears that these assumptions typically do hold, or can easily be made to hold, for languages of expressions whose values depend on an interpretation of the generating set but not on a choice of initial state; note particularly that they hold for Examples AE and BE.

We now define the language $\underline{L}'$ of assignment statements to be simply the set of pairs $x := w$, with $x \in X$ and $w \in |\underline{L}|$. $\underline{L}'$ is trivially an algebra: it has no operations. (Note that we write " $x := w$ " merely as a suggestive syntactic alternative to " $\langle x,w \rangle$ ".) We have now to construct a commutative diagram:



with φ' an acceptable semantic function for assignment statements; κ' will then be our desired correct compiler for assignments. Note that because $\underline{L}'$ has no operations, to require that the arrows of the primed diagram be homomorphisms is merely to require that they be functions. Likewise we need not bother to distinguish between the trivial algebras in the primed diagram and the sets which are their carriers.

Our definition of the semantics of $\underline{L}'$ will be conventional: we regard the meaning of an assignment statement as being a transformation on states, and we identify "state" with "interpretation of the variables", so that we have $\underline{S}' = (X \to A) \to (X \to A)$. The effect of an assignment $x := w$ should of course be to modify the value of the state at x so that it assumes the previous value of w ; therefore we naturally define φ' by (for $i: X \to A$ and $y \in X$):

$$\varphi'(x := w)(i)(y) = \underline{\text{if}}\ y = x\ \underline{\text{then}}\ \varphi_i(w)(0)\ \underline{\text{else}}\ i(y) \quad .$$

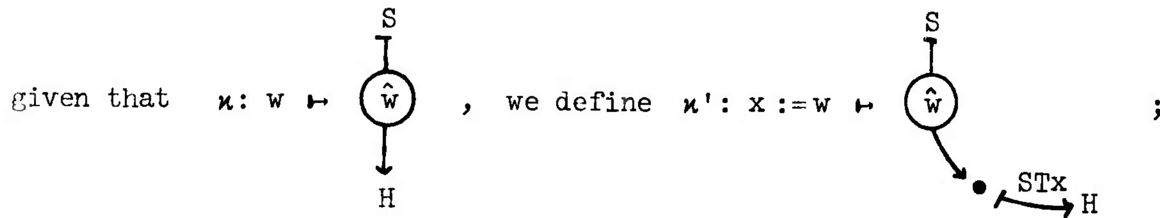
As an abbreviation we may define a function, "assign", by:

$$\text{assign}(x,a,i)(y) = \underline{\text{if}}\ y = x\ \underline{\text{then}}\ a\ \underline{\text{else}}\ i(y) \quad ,$$

and then write:

$$\varphi'(x := w):\ i \mapsto \text{assign}(x, \varphi_i(w)(0), i) \quad .$$

The compiler we will need, κ', is trivial, as was promised:



we see that the operator set of our new target programs (and hence of the new target machine, $\underline{\mathfrak{M}'}$) must be $\Gamma \cup \{STx \mid x \in X\}$. We shall define ψ' in the standard way by $\psi' = \rho' \circ ac \circ \pi'$, with $\pi' : \underline{P} \mapsto \underline{P} \otimes \underline{\mathfrak{M}'}$ and $\rho' : R \mapsto R \rceil (\{S,H\} \times |\underline{\mathfrak{M}'}|)$. Even before the action of $\underline{\mathfrak{M}'}$ has been defined, we can see the essential property of κ' (the argument was gone through in detail for Example SS and need not be repeated): we will have for every statement $x := w \in \underline{L}'$,

$$(*) \quad \psi' \circ \kappa'(x := w) = 1_{\{S,H\} \times |\underline{\mathfrak{M}'}|} \cup \{S,m \mapsto S,m\};(\psi' \circ \kappa(w));(\{H \mapsto H\} \times \underline{\mathfrak{M}'}_{STx}) .$$

The way to our desired result is now clear: we need to combine all the $\underline{\mathfrak{M}i}$ into $\underline{\mathfrak{M}'}$, by taking $|\underline{\mathfrak{M}'}| = M \times (X \to A)$, and defining the action of $\underline{\mathfrak{M}'}$ for $\gamma \in \Gamma$ by

$$\underline{\mathfrak{M}}'_\gamma : m_1, i \mapsto m_2, i \ \underline{\text{iff}} \ \underline{\mathfrak{M}i}_\gamma : m_1 \mapsto m_2 \quad ;$$

this will evidently give us, for the evaluation of a right-hand-side on $\underline{\mathfrak{M}'}$,

$$(**) \quad \psi' \circ \kappa(w) : \langle S,m,i \rangle \mapsto \langle H, [\{H,m \mapsto m\} \circ (\psi_i \circ \kappa(w))](S,m), i \rangle \quad .$$

We then define the operations $\underline{\mathfrak{M}}'_{STx}$ for $x \in X$ in such a way as to store in the appropriate "location" of the $(X \to A)$-component that element of A represented by the state of the M-component we have arrived at; that is, we define:

(***) $\quad \underline{\mathfrak{M}}'_{STx}: m,i \mapsto m_o,assign(x,d(H,m),i)$.

(It will be convenient when, for example, we come to compounding assignment statements, that the execution of a store operation returns the M-component to its initial state. If we had been more specific about the structure of M, we might have been able to specify the action of $\underline{\mathfrak{M}}'_{STx}$ on M more conservatively, for example to remove only the top element of a stack.)

The correct definition for δ' is now evident: $\delta': R \mapsto d' \circ R \circ e'$, where $e': i \mapsto \langle S,m_o,i\rangle$ and $d': \langle H,m_o,i\rangle \mapsto i$.

The correctness of $\varkappa'$, that is the commutativity condition $\delta' \circ \psi' \circ \varkappa' = \varphi'$, is now easily checked. We have by definition:

$$\varphi'(x := w): i \mapsto assign(x,\varphi_i(w)(0),i) \quad .$$

But,

$$\delta' \circ \psi' \circ \varkappa'(x := w): i \mapsto [d' \circ (\psi' \circ \varkappa'(x := w))](S,m_o,i)$$

$$= [d' \circ ((\psi' \circ \varkappa(w));(\{H \to H\} \times \underline{\mathfrak{M}}'_{STx}))](S,m_o,i) \qquad \text{(by (*))}$$

$$= [d' \circ (\{H \mapsto H\} \times \underline{\mathfrak{M}}'_{STx})](H,[\{H,m \mapsto m\} \circ (\psi_i \circ \varkappa(w))](S,m_o),i) \qquad \text{(by (**))}$$

$$= d'(H,m_o,assign(x,[d \circ (\psi_i \circ \varkappa(w))](S,m_o),i)) \qquad \text{(by (***))}$$

$$= d'(H,m_o,assign(x,\varphi_i(w)(0),i)) \qquad \text{(by assumed correctness of } \varkappa)$$

$$= assign(x,\varphi_i(w)(0),i) \quad . \quad \blacksquare$$

With an eye to the application in Chapter XII, in which we shall want assignment to both arithmetic and Boolean variables, in the latter case coercing arithmetic values to truth values, we note the following

evident generalization (by no means as general as possible) of the result just proved: Suppose that we have an additional set of variables Y , disjoint from X , which we wish to take values in a set B , in general different from A , and that we have a function rep: $A \tilde{\rightarrow} B$ which allows at least some of the values in A to stand as representatives of the values in B. Then we may define an augmented language $\underline{L}^+ = \underline{L}' \cup \{y := w \mid y \in Y \ \underline{\text{and}}\ w \in |\underline{L}|\}$, an augmented set of state transformations $\underline{S}^+ = (X \rightarrow A) \times (Y \rightarrow B) \tilde{\rightarrow} (X \rightarrow A) \times (Y \rightarrow B)$, and if we define a modified assign function, assign^+ , by:

$$\begin{cases} \text{if } x \in X \text{ then } \text{assign}^+(x,a,i)(z) = \underline{\text{if}}\ z = x\ \underline{\text{then}}\ a\ \underline{\text{else}}\ i(z) \\ \text{if } y \in Y \text{ and rep: } a \mapsto b \text{ then} \\ \qquad \text{assign}^+(y,a,i)(z) = \underline{\text{if}}\ z = y\ \underline{\text{then}}\ \text{rep}(a)\ \underline{\text{else}}\ i(z) \\ (\text{assign}^+ \text{ undefined otherwise}) \end{cases}$$

then we may for our semantic homomorphism φ^+ write as before:

$$\varphi^+(x := w): i \mapsto \text{assign}^+(x, \varphi_{(i \upharpoonright X)}, i) \quad .$$

It is evident that by taking a suitably augmented target machine $\underline{\mathcal{M}}^+$ we can define κ^+ , ψ^+ , and δ^+ as before, and again get a commutative diagram. The necessary change to the target machine is simply to take $|\underline{\mathcal{M}}^+| = M \times (X \rightarrow A) \times (Y \rightarrow B)$, and to provide $\underline{\mathcal{M}}^+$ with additional operations $\underline{\mathcal{M}}^+_{STy}$ for $y \in Y$, defined by:

$$\underline{\mathcal{M}}^+_{STy}: m,i \mapsto m_o, \text{assign}^+(y, d(H,m), i)$$

whenever the latter is defined.

The store operations of the machine $\underline{\mathcal{M}}'$ are similar in effect to instructions of many real digital computers; in the synthetic example

of Chapter XII they will be assumed to be executable, together with load operations like those introduced in Example AE, by the final target machine. The remainder of this chapter, which will carry the compilation of assignment statements one step farther, forms an example of modelling machines by BRAs, but is perhaps not directly relevant to practical compiler-correctness proofs.

The machine $\underline{\mathfrak{M}}'$ which we have just developed, although its operator set is analogous to the instruction set of a typical digital computer, is theoretically unsatisfactory because its structure -- of a store combined with an "arithmetic unit" -- is not apparent. Moreover, the assign function used in the definition of $\underline{\mathfrak{M}}'$ is mathematically rather complicated (although familiar to programmers); we would like to not only isolate a store component but analyze it as an assemblage of "locations".

We proceed to meet these two criticisms by introducing a modified machine $\underline{\mathfrak{M}}''$, and a compiler $\varkappa''$ which carries programs for $\underline{\mathfrak{M}}'$ into programs for $\underline{\mathfrak{M}}''$; once we have shown $\varkappa''$ correct, $\varkappa'' \circ \varkappa'$ will be a correct compiler for $\underline{L}'$ with $\underline{\mathfrak{M}}''$ as target machine, and $\underline{\mathfrak{M}}''$ will be defined as a product of meaningful factors -- "arithmetic unit" and "locations". The proof about $\varkappa''$ will be simpler than a general compiler proof, because we will have $|\underline{\mathfrak{M}}''| = |\underline{\mathfrak{M}}'|$ and also for all $\underline{L}'$-programs $\underline{\rho}$, $|\varkappa''(\underline{\rho})| = |\underline{\rho}|$, and we will be able to show that $\Sigma(\varkappa''(\underline{\rho}) \otimes \underline{\mathfrak{M}}'') = \Sigma(\underline{\rho} \otimes \underline{\mathfrak{M}}')$, so that no argument will need to be made about $\mathcal{A}nc$ (recall that $ac = \mathcal{A}nc \circ \Sigma$), restriction, or decoding.

Before we start we must make some further assumptions about the family of machines $\underline{\mathcal{M}i}$: in essence that they really do only differ according to the values their respective interpretations give to the variables in X , that they treat all variables alike, and that at most one variable is "read" by a single operation (these restrictions are met by, for example, the machines for Examples AE and BE; in more complicated situations in which not all the restrictions were met, one would expect to need a more complicated construction than the one we shall give here). Formally, we assume (recalling $\underline{\mathcal{M}i} : \Gamma \to (M \to M)$)

(i) $\Gamma = \Gamma_c \cup \{Lx \mid x \in X\}$ (L for "load") ;

(ii) $\underline{\mathcal{M}i}_\gamma = \underline{\mathcal{M}j}_\gamma$ for $\gamma \in \Gamma_o$ and for $i,j: X \to A$;

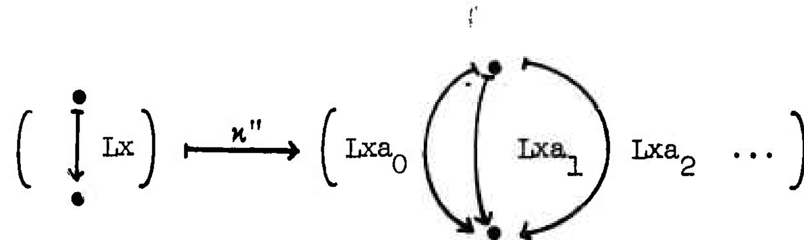
(iii) there is a function $\ell: A \to (M \to M)$ such that for any $i: X \to A$,

$\underline{\mathcal{M}i}_{Lx} = \ell(i(x))$.

(It will be evident how the construction which follows could be extended trivially in case there were many load instructions for each variable rather than one. On the other hand, it is also clear that one kind of load instruction is enough: we could adjoin a "memory buffer register" -- i.e., make the carrier of the $\underline{\mathcal{M}i}$ be $A \times M$ rather than M -- and then split each variety of load operation into "load memory buffer register" followed by a suitable new Γ_o-operation.)

The idea behind the construction of $\underline{\mathcal{M}''}$ is very simple: it is to split each operation Lx into a family of operations Lxa , one for each $a \in A$, each Lxa to be capable of loading only the specific value a from location x ; and similarly to split each operation STx into a family of operations STxa . Even this intuitive description makes it

clear how we want $\varkappa''$ to behave: it has simply to split each operation-instance of Lx (respectively STx) occurring in a program into a bundle of operation-instances for the various elements of A . We may picture the process of compiling with $\varkappa''$ as follows:



(and a similar diagram for STx). More rigorously, if not very transparently, we can define $\varkappa''(\underline{\rho})$ for any $\underline{L}'$-program $\underline{\rho}$ by $|\varkappa''(\underline{\rho})| = |\underline{\rho}|$,

$$\varkappa''(\underline{\rho})_\gamma = \underline{\rho}_\gamma \quad \text{for } \gamma\epsilon\Gamma_o \ ,$$

$$\varkappa''(\underline{\rho})_{Lxa} = \underline{\rho}_{Lx} \quad \text{for } x\epsilon X \ , \ a\epsilon A \ , \text{ and}$$

$$\varkappa''(\underline{\rho})_{STxa} = \underline{\rho}_{STx} \quad \text{for } x\epsilon X \ , \ a\epsilon A \ .$$

Now we see that if we can define $\underline{\mathfrak{M}}''$ so that

(+) $$\underline{\mathfrak{M}}''_\gamma = \underline{\mathfrak{M}}'_\gamma \quad \text{for } \gamma\epsilon\Gamma_o \ ,$$

$$\bigcup_{a\epsilon A} \underline{\mathfrak{M}}''_{Lxa} = \underline{\mathfrak{M}}'_{Lx} \ , \quad \text{and}$$

$$\bigcup_{a\epsilon A} \underline{\mathfrak{M}}''_{STxa} = \underline{\mathfrak{M}}'_{STx} \ ,$$

then we will have at once our desired result:

$$\Sigma(\kappa''(\underline{P}) \otimes \underline{\mathfrak{M}}'') = \bigcup_{\gamma\in\Gamma_o \cup \{STxa, Lxa\}} \kappa''(\underline{P})_\gamma \times \underline{\mathfrak{M}}''_\gamma$$

$$= \bigcup_{\gamma\in\Gamma \cup \{STx\}} \underline{P}_\gamma \times \underline{\mathfrak{M}}'_\gamma$$

$$= \Sigma(\underline{P} \otimes \underline{\mathfrak{M}}') \quad ;$$

all we need observe is that (+) and the definition of κ'' imply:

$\underline{P} \otimes \underline{\mathfrak{M}}'_{Lx}: p,m \mapsto p',m'$ <u>iff</u> <u>for</u> <u>some</u> $a\epsilon A$, $\kappa''(\underline{P}) \otimes \underline{\mathfrak{M}}''_{Lxa}: p,m \mapsto p',m'$

and the analogous biconditional for STx and STxa . But it turns out that we will get exactly the $\underline{\mathfrak{M}}''$ we need by defining it as the following product:

$\underline{\mathfrak{M}}'' = \bar{\underline{\mathfrak{M}}} \otimes \underline{\mathcal{J}}$, where $|\bar{\underline{\mathfrak{M}}}| = M$, $|\underline{\mathcal{J}}| = (X \rightarrow A)$;

$\bar{\underline{\mathfrak{M}}}_\gamma = \underline{\mathfrak{M}i}_\gamma$ for $\gamma\epsilon\Gamma_o$, $\bar{\underline{\mathfrak{M}}}_{Lxa}: m \mapsto \ell(a)(m)$,

$\bar{\underline{\mathfrak{M}}}_{STxa}: m \mapsto m_o$ <u>iff</u> $d: H,m \mapsto a$;

$\underline{\mathcal{J}}_\gamma = 1_{X \rightarrow A}$ for $\gamma\epsilon\Gamma_o$, $\underline{\mathcal{J}}_{Lxa}: i \mapsto i$ <u>iff</u> $i: x \mapsto a$, and

$\underline{\mathcal{J}}_{STxa}: i \mapsto assign(x,a,i)$.

To verify (+), we simply check:

$$(\bar{\underline{\mathfrak{M}}} \otimes \underline{\mathcal{J}})_\gamma = \bar{\underline{\mathfrak{M}}}_\gamma \times 1_{X \rightarrow A} = \underline{\mathfrak{M}}'_\gamma \quad \text{for } \gamma\epsilon\Gamma_o \quad ;$$

and

$$\bigcup_{a\epsilon A} \bar{\underline{\mathfrak{M}}} \otimes \underline{\mathcal{J}}_{Lxa}: m,i \mapsto \ell(a)(m),i \text{ \underline{iff} } i: x \mapsto a \quad ,$$

that is,

$$\bigcup_{a\epsilon A} \bar{\underline{\mathfrak{M}}} \otimes \underline{\mathcal{J}}_{Lxa}: m,i \mapsto \ell(i(x))(m),i \quad ;$$

and

$$\bigcup_{a\in A} \underline{\bar{\mathcal{M}}} \otimes \underline{\mathcal{S}}_{STxa} : m,i \mapsto m_o, \text{assign}(x,a,i) \ \underline{\text{iff}} \ d: H,m \mapsto a \quad ,$$

that is,

$$\bigcup_{a\in A} \underline{\bar{\mathcal{M}}} \otimes \underline{\mathcal{S}}_{STxa} : m,i \mapsto m_o, \text{assign}(x,d(H,m),i) \quad . \quad \blacksquare$$

We now have our two factors of $\underline{\mathcal{M}}''$: $\underline{\mathcal{S}}$ the "store" and $\underline{\bar{\mathcal{M}}}$ the "arithmetic unit". We claim, moreover, that $\underline{\mathcal{S}}$ is the product of one factor for each element of X , which it may be appropriate to call a "location machine", namely:

$$\underline{\mathcal{S}} = \bigotimes_{x\in X} \underline{\mathcal{S}x} \quad ,$$

where

$$|\underline{\mathcal{S}x}| = A \ ,$$

$$\underline{\mathcal{S}x}_\gamma = 1_A \quad \text{for} \quad \gamma\in\Gamma_o \quad ,$$

$$\underline{\mathcal{S}x}_{Lya} = \underline{\text{if}} \ y = x \ \underline{\text{then}} \ \{a \mapsto a\} \ \underline{\text{else}} \ 1_A \quad ,$$

and

$$\underline{\mathcal{S}x}_{STya} = \underline{\text{if}} \ y = x \ \underline{\text{then}} \ \{b \mapsto a\} \ \underline{\text{else}} \ 1_A \quad .$$

The verification of this decomposition is a trivial exercise (making, of course, the necessary identification of the Cartesian product $\prod_{x\in X} A$ with $(X \to A)$).

It is evident that we can carry through this second part of our construction in essentially the same way in the case where we have an additional set Y of B-valued variables, to get an augmented version

of $\underline{\mathcal{M}}''$ which factors as $\bar{\underline{\mathcal{M}}}^+ \otimes \underline{\mathcal{S}}^+$. Here $\bar{\underline{\mathcal{M}}}^+$ is defined the same way as $\bar{\underline{\mathcal{M}}}$, but with the operations

$$\bar{\underline{\mathcal{M}}}^+_{STxa}: m \mapsto m_o \ \underline{\text{iff}} \ d: H,m \mapsto a$$

now existing for all $x \in X \cup Y$; and $\underline{\mathcal{S}}^+$ is defined the same way as $\underline{\mathcal{S}}$, except that $|\underline{\mathcal{S}}^+| = (X \to A) \times (Y \to B)$, and now

$$\underline{\mathcal{S}}^+_{STxa}: i \mapsto \text{assign}^+(x,a,i)$$

for any $x \in X \cup Y$.

Furthermore, $\underline{\mathcal{S}}^+$ decomposes into all the factors of $\underline{\mathcal{S}}$, one for each $x \in X$ (with the trivial modification that the additional operators STya , $y \in Y$, each denote the identity operation 1_A) and into additional factors $\underline{\mathcal{S}y}^+$, one for each $y \in Y$, each of which has $|\underline{\mathcal{S}y}^+| = B$, all operations the identity 1_B except for:

$$\underline{\mathcal{S}y}^+_{STya}: b \mapsto \text{rep}(a)$$

(note that this is the empty relation if rep is undefined at a).

XI. While Statements

The purpose of this section is to show by example that the style of compilers we allow -- BRA-producing homomorphisms -- easily and naturally handles (as we should expect) the sort of programming language construct which is customarily defined by a rule for replacing each instance of it by a system of tests and branches. A typical instance of this sort of construct, and probably the simplest, is the <u>while</u> statement (familiar to students of Algol-like languages, even though not exhibited in its pure form in Algol 60). Note that whereas the ordinary theoretical treatment of <u>while</u>s (see, for example, [Hoare 69]) takes the equivalence of the <u>while</u> statement to a loop as given and proceeds to derive the consequence that the function denoted by a <u>while</u> statement satisfies a recursive inequality, we shall be proving the same thing for an ostensibly different reason -- i.e., we shall take the recursive semantics of <u>while</u>s as given and proceed to show that the compilation of <u>while</u>s as loops is correct.

The present context may also serve to exemplify two other points of possible significance. The first is that there need be no incompatability between an algebraic semantics and an axiomatic one. That is, we may give axioms for a semantic algebra without determining it completely, and prove from them properties which must be true of any algebra satisfying the axioms. (This is, of course, standard mathematical practice; we lay stress on it here only because most of our algebras have been explicitly defined.) The second is that even without use of the compiler composition theorem we may in certain

circumstances claim that correct compilers for two languages automatically yield a correct compiler for the omnibus language which combines their "features" -- namely in the case that the two compilers are sufficiently compatible, in a sense which will become apparent.

We take then $\underline{L} = \underline{W}_{\{\underline{while}\ p\ \underline{do},\ \underline{while}\ q\ \underline{do},\ \ldots\}}(X)$, where $X = \{x,y,\ldots\}$ is a set of elementary statements, and $P = \{p,q,\ldots\}$ is a set of predicate expressions.

We want $\underline{S}$, the semantic algebra, to be one of relations on a set A to itself. We assume an interpretation $i: X \to (A \twoheadrightarrow A)$ of the elementary statements as relations on A , and an interpretation $j: P \to (A \to \underline{2})$ of the predicate expressions as predicates on A . We specify the operations of $\underline{S}$ incompletely by laying down for each $p \epsilon P$ an axiom, namely the following recursive inequality:

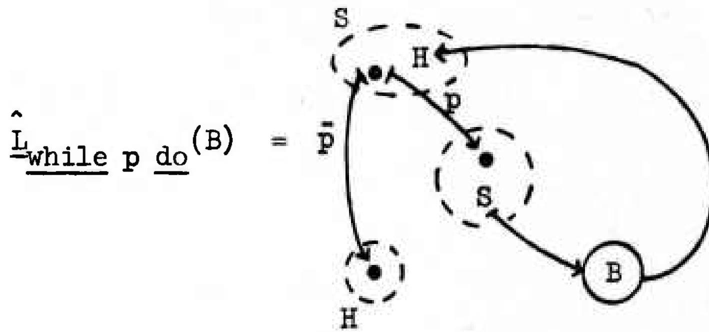
$$(*) \quad \underline{S}_{\underline{while}\ p\ \underline{do}}(f) \supseteq \{a \mapsto a \mid \underline{not}\ j(p)(a)\} \cup (\{a \mapsto a \mid j(p)(a)\};f;\underline{S}_{\underline{while}\ p\ \underline{do}}(f))$$

and we require φ , the source semantic homomorphism, to be the extension of i to a homomorphism : $\underline{L} \to \underline{S}$.

Forseeing that we shall expect the target machine to be able to evaluate the elementary statements and predicates directly, and knowing the shape of while loops, it is easy to specify the action of the compiler on X :

$$\kappa: x \to \begin{matrix} S \\ \big\downarrow x \\ H \end{matrix} ,$$

and to give the operations in $\hat{\underline{L}}$ by the following diagram for each $p \epsilon P$:



Evidently the target BRAs (hence also the target machine) are to have operator set $X \cup P \cup \{\bar{p} \mid p \in P\}$.

We may now describe the target machine $\underline{\mathcal{M}}$. Naturally we take $|\underline{\mathcal{M}}| = A$, and for $x \in X$, $\underline{\mathcal{M}}_x = i(x)$. We use the device described in Example I.2 for modelling predicates by partial identity functions ("Karping"), and require, for $p \in P$,

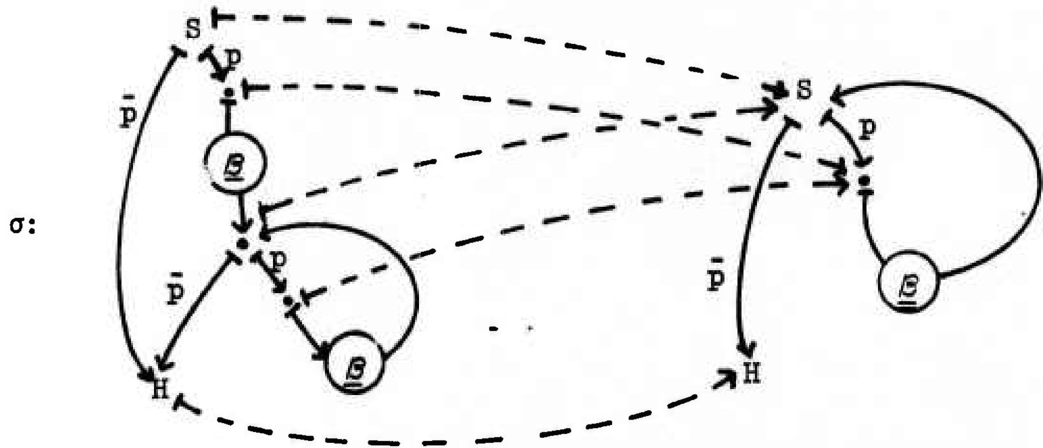
$$\underline{\mathcal{M}}_p = \{a \mapsto a \mid j(p)(a)\} \quad , \quad \underline{\mathcal{M}}_{\bar{p}} = \{a \mapsto a \mid \underline{\text{not}}\ j(p)(a)\} \quad .$$

For D , the domain of the relations computed, we have $\{S,H\} \times A$, and this suggests the already-familiar choice of $\delta: r \mapsto d \circ r \circ e$ given by $d: \langle H,a\rangle \mapsto a$, $e: a \mapsto \langle S,a\rangle$. δ is evidently an isomorphism from $\underline{R}$ to an algebra $\underline{S}'$ whose operations are given by

$$\underline{S}'_{\underline{while}\ p\ \underline{do}}(f) = d \circ (\underline{R}_{\underline{while}\ p\ \underline{do}}(d^{-1} \circ f \circ e^{-1})) \circ e \quad .$$

$\underline{S}$ being only specified as satisfying the axioms (*), all we need prove for the penultimate step of our correctness proof (closure of the diagram) is that $\underline{S}'$ satisfies (*), and hence is a suitable $\underline{S}$. But this turns out surprisingly easy to prove, once we make the observation that for any BRA $\underline{\mathcal{B}}$ we have the simulation $\sigma: \underline{\mathcal{B}}'' \to \underline{\mathcal{B}}'$, where $\underline{\mathcal{B}}'$ is $\underline{L}_{\underline{while}\ p\ \underline{do}}(\underline{\mathcal{B}})$ and $\underline{\mathcal{B}}''$ is the same loop "unrolled" once, i.e.,

σ:



where σ acts as indicated by the dotted arrows, and in addition, of course, coalesces the two copies of $\underline{\mathcal{B}}$ into one.

It is easily seen that if we follow the execution and decoding morphisms around to $\underline{S}'$, we will obtain for the relation computed by $\underline{\mathcal{B}}''$,

$$\delta\circ\psi(\underline{\mathcal{B}}'') = \{a \mapsto a \mid \underline{\text{not}}\ j(p)(a)\} \cup (\{a \mapsto a \mid j(p)(a)\};\delta\circ\psi(\underline{\mathcal{B}});\delta\circ\psi(\underline{\mathcal{B}}')) \quad .$$

However, decomposing the target semantic homomorphism as $\psi = \rho\circ ac\circ\pi$, and recalling that π and ac are functors, and hence preserve simulations, we see that from the simulation of flow diagrams noted above follows

$$\sigma\times 1_A\colon ac(\underline{\mathcal{B}}''\otimes\underline{\mathfrak{M}}) \to ac(\underline{\mathcal{B}}'\otimes\underline{\mathfrak{M}}) \quad ;$$

i.e., (by the definition of simulation)

$$ac(\underline{\mathcal{B}}''\otimes\underline{\mathfrak{M}});(\sigma\times 1_A) \subseteq (\sigma\times 1_A);ac(\underline{\mathcal{B}}'\otimes\underline{\mathfrak{M}}) \quad .$$

But, as can be seen from our definition of σ,

$$(\sigma \times 1_A) \rceil \{S,H\} \times A = 1_{\{S,H\} \times A} \quad ;$$

hence we obtain in fact

$$\psi(\underline{\mathcal{B}}'') = ac(\underline{\mathcal{B}}'' \otimes \underline{\mathcal{M}}) \rceil \{S,H\} \times A$$

$$\subseteq ac(\underline{\mathcal{B}}' \otimes \underline{\mathcal{M}}) \rceil \{S,H\} \times A = \psi(\underline{\mathcal{B}}') \quad ,$$

and following δ from $\underline{R}$ to $\underline{S}'$ we at last have $\delta \circ \psi(\underline{\mathcal{B}}'') \subseteq \delta \circ \psi(\underline{\mathcal{B}}')$ which is just what we wanted to prove; we are justified in taking $\underline{S}'$ to be $\underline{S}$, and we have our closed diagram of homomorphisms.

The last step of proving correctness of our <u>while</u> statement compiler has been done already in example SS; it consists simply in observing that $\varphi(x) = i(x) = \delta \circ \psi \circ \kappa(x)$ for $x \in X$. ∎

In fact the similarity between the current example and example SS has more far-reaching consequences than merely to save us redoing a step of a proof. Every pair of corresponding homomorphisms in the two examples are extensions to homomorphisms of the same function on the generating set; moreover the two target machines agree on their common operators; in short, there is nothing to stop us claiming that we have given a (disconnected) proof of the correctness of a single compiler for a language which has as its operations the formation of both compound statements and <u>while</u> statements. It is this combined language which will be meant by references below to "the language of Chapter XI".

What we may conclude in general is that when there is no conflict in the compilation of two algebras (action of the compilers the same on generating sets, compatible machines, same restriction, same decoding in both cases) then simply from the fact that a proof about homomorphisms for multi-operation algebras is just a proof for each

operation separately, we have for free a compiler for the combination of the algebras. In more programming-language-oriented terms, under these conditions we obtain automatically a compiler for a language in which mutual recursion between two constructs is allowed, although we have apparently only proved compiler correctness assuming that either of the constructs could be used in isolation.

XII. An Exemplary Synthesis

The aim of the present chapter is to illustrate the utility of what has gone before by using the compiler composition theorem to tie together most of the previous examples of "single-feature" languages into a demonstration of the correctness of a compiler for a somewhat "realistic" language -- very loosely, a language "with the features" of whiles, sequencing, gotos, assignment to simple variables, Boolean and arithmetic expressions. It cannot be over-emphasized that the achievement of this one proof is not to be considered as the total accomplishment of the present work; rather the synthesis to be performed in this chapter should be understood as an advertisement for our algebraic approach; it is meant to exemplify a class of possible syntheses which could be made easily and naturally with the tools we have developed. (Admittedly, we have in our examples treated only a very small set of language fragments, and will here assemble essentially all of them; it is not evident that a synthesis interestingly different from the one we shall show could be performed without first inventing some new "single-feature" languages as raw materials.)

We proceed forthwith to an informal description of the language $\underline{L}_{synth}$ with which we shall deal. The following parameters of the language will be left unspecified: the choice of a domain A of "arithmetic" operands; the choice of a family of "arithmetic" operations on A and of a set Ω of operators to denote them, the choice of a (partial or total) function rep: $A \rightharpoonup \{\underline{true}, \underline{false}\}$ by which certain arithmetic values may be used to represent truth values, and the choice of a set V of arithmetic variables and of a set U (disjoint from V)

of Boolean variables. (For brevity we shall henceforth throughout this chapter denote the set $\{\underline{\text{true}}, \underline{\text{false}}\}$ by $\underline{2}$.)

Programs of $\underline{L}_{synth}$ will be finite, multi-entrance, multi-exit (Karped) flowcharts. The tests will be of the form $p \swarrow\!\searrow \bar{p}$, where p is any Boolean expression (as in example BE) built from variables in U . The commands of $\underline{L}_{synth}$ will be arbitrary nests of <u>while</u> and compound statements built up from assignment statements of the form $x := e$, where x is a variable in $V \cup U$, and e is an expression built up, as in example AE, by "arithmetic" operators from the variables in V . The <u>while</u> statements are to admit the same set of Boolean expressions as may appear in the top-level tests of $\underline{L}_{synth}$.

To restate the foregoing somewhat more formally, and in the bottom-up direction, we define the following languages:

$\underline{L}_{arith} = \underline{W}_{\Omega}(V)$, that instance of the language of example AE got by taking the particular sets Ω of operators and V of arithmetic variables;

$\underline{L}_{assig} = \{x := e \mid x \in U \cup V \text{ and } e \in |\underline{L}_{arith}|\}$, that instance of the "augmented" assignment language of Chapter X got by taking $\underline{L}_{arith}$ as the language of right-hand-side expressions, and U as the set of "extra" assignable variables;

$\underline{L}_{bool} = \underline{W}_{\{\neg, \wedge, \vee, \supset\}}(U)$, that instance of example BE got by taking U for the generating set;

$\underline{L}_{wc} = \underline{W}_{\{\Box\} \cup \{\underline{\text{while}}\ p\ \underline{\text{do}} \mid p \in |\underline{L}_{bool}|\}}(\underline{L}_{assig})$, an instance of the language of Chapter XI;

$\underline{L}_{synth}$ = an algebra of $|\underline{L}_{wc}| \cup |\underline{L}_{bool}| \cup \{\bar{p} \mid p \in |\underline{L}_{bool}|\}$ - BRAs , with suitable construction operations for building up flowcharts -- the choice of these constructions turns out to be a rather special problem, whose discussion we defer.

The semantic homomorphisms for the various intermediate languages are the appropriate instances of the ones we developed earlier:

For $\underline{L}_{arith}$ we have for each function $i: (V \to A) \times (U \to \underline{2})$ a homomorphism $\varphi_{i\restriction V}: \underline{L}_{arith} \to \underline{A}$, agreeing with i on V , where $\underline{A}$ is the Ω-algebra with carrier A whose operations are whatever "arithmetic" operations we may have chosen.

For $\underline{L}_{bool}$ we have, as defined in example BE, a homomorphism $\varphi_{i\restriction U}: \underline{L}_{bool} \to \underline{B2}$ for each $i: (V \to A) \times (U \to \underline{2})$, agreeing with i on U , where $\underline{B2}$ is the $\{\neg, \wedge, \vee, \supset\}$-algebra of truth values with the classical operations <u>not</u>, <u>and</u>, <u>or</u>, <u>implies</u>.

For $\underline{L}_{assig}$ we have a semantic function

$\varphi_{assig}: \underline{L}_{assig} \to ((V \to A) \times (U \to \underline{2}) \rightleftarrows (V \to A) \times (U \to \underline{2}))$ given by

$\varphi_{assig}(x := e): i \mapsto assign^{+}(x, \varphi_{i\restriction V}(e), i)$, where

$$\begin{cases} \text{for } x \in V, \quad assign^{+}(x,a,i): y \mapsto \underline{\text{if}}\ y = x\ \underline{\text{then}}\ a\ \underline{\text{else}}\ i(y), \\ \text{for } x \in U, \quad \text{if } rep: a \mapsto b \text{ then} \\ \quad assign^{+}(x,a,i): y \to \underline{\text{if}}\ y = x\ \underline{\text{then}}\ b\ \underline{\text{else}}\ i(y), \\ (assign^{+}(x,a,i) \text{ undefined otherwise}), \end{cases}$$

and where rep is a (partial) function which interprets certain arithmetic values as representations for truth values -- e.g. we might have $rep: \begin{Bmatrix} 0 \to \underline{\text{false}} \\ 1 \to \underline{\text{true}} \end{Bmatrix}$.

(The idea is that an operation such as $\leq$, which we would naturally think of as operating on two arithmetic values to yield a truth value, will here be thought of as yielding another arithmetic value; the latter will be appropriately interpreted by rep on assignment to a Boolean variable. This device allows us to assimilate predicates to the ordinary operations of our arithmetic algebra $\underline{A}$, and in the implementation whose correctness we shall prove, it will model faithfully the commonly existing situation in which representations of truth values, as held on the stack, are indistinguishable from representations of numbers. All the same, the necessity to introduce rep is displeasing and suggests that our algebraic notions are too rigid; this problem and the possibility of its solution will be discussed in the conclusion.)

For $\underline{L}_{wc}$ we require a semantic homomorphism φ_{wc} to an algebra of partial functions on $(V \to A) \times (U \to \underline{2})$ to itself. This will simply be an instance of the development in Chapter XI, where the semantic homomorphism for a general language of <u>while</u> and compound statements is defined (or rather partly defined and partly axiomatized) in terms of interpretations for elementary statements and for Boolean expressions. We have already an interpretation of the correct type for our elementary statements, namely:

$$\varphi_{assig}: \underline{L}_{assig} \to ((V \to A) \times (U \to \underline{2}) \rightharpoonup (V \to A) \times (U \to \underline{2})) \ .$$

We need also an interpretation:

$$j: \underline{L}_{bool} \to ((V \to A) \times (U \to \underline{2}) \to \underline{2}) \ ;$$

we may obtain the function we want by interchanging the arguments of the semantic homomorphism for $\underline{L}_{bool}$; that is, we define:

$j(p): i \mapsto \varphi_{(i \upharpoonright U)}(p)$ for each $p \in |\underline{L}_{bool}|$.

Finally (since $\underline{L}_{synth}$ is to be an algebra of BRAs) we may describe the semantic homomorphism for $\underline{L}_{synth}$, $\varphi_{synth}: \underline{L}_{synth} \to \underline{S}$, as a composition $\varphi_{synth} = \rho \circ ac \circ \pi$, even though the constructions of $\underline{L}_{synth}$ (and hence as well the operations of $\underline{S}$) are as yet undecided. We define the source machine, $\underline{\mathscr{A}}$, for $\underline{L}_{synth}$ by taking

$|\underline{\mathscr{A}}| = (V \to A) \times (U \to \underline{2})$,

$$\underline{\mathscr{A}}_s: i \mapsto \varphi_{wc}(s)(i) \qquad \text{for } s \in |\underline{L}_{wc}|$$

and

$$\underline{\mathscr{A}}_p = \{i \mapsto i \mid \varphi_{bool}(p)(i)\} \quad , \quad \underline{\mathscr{A}}_{\bar{p}} = \{i \mapsto i \mid \underline{\text{not}}\ \varphi_{bool}(p)(i)\}$$

$$\text{for } p \in |\underline{L}_{bool}| \ .$$

We then have π defined as usual, for $\underline{\rho} \in |\underline{L}_{synth}|$, by $\pi: \underline{\rho} \mapsto \underline{\rho} \otimes \underline{\mathscr{A}}$.

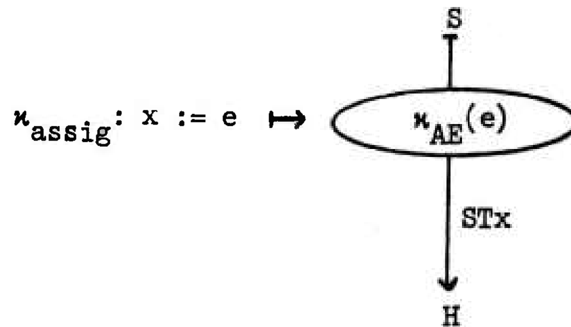
Since we wish to allow our flowcharts to have an unbounded number of entries and exits, we define a set $SH = \{S_0, S_1, \ldots\} \cup \{H_0, H_1, \ldots\}$ containing infinitely many distinguished start and halt nodes, and we shall insist that for $\underline{\rho} \in |\underline{L}_{synth}|$, we have $SH \subseteq |\underline{\rho}|$. It is then natural to take for our restriction homomorphism:

$$\rho: R \mapsto R \rfloor (SH \times |\underline{\mathscr{A}}|) \ .$$

Evidently the semantic algebra $\underline{S}$ will be one of relations on $SH \times (V \to A) \times (U \to \underline{2})$ to itself.

We have now to show that we can construct a correct compiler, κ_{synth} , for $\underline{L}_{synth}$ by compounding our previous fragmentary results.

We have immediately that $\varkappa_{AE}$, the compiler for arithmetic expressions developed in example AE, is a suitable compiler for right-hand-side expressions which may be plugged in to the ("augmented") construction of Chapter X. This will give us a correct compiler $\varkappa_{assig}$ defined by

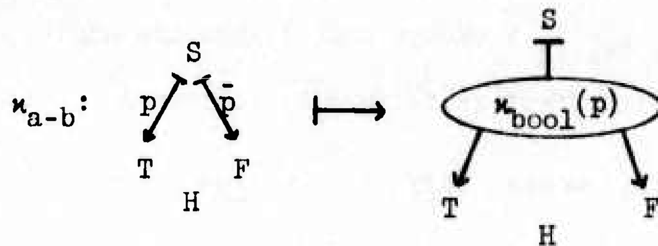


The target machine for $\underline{L}_{assig}$, which we may call $\underline{\mathfrak{M}a}$, will have carrier $|\underline{\mathfrak{M}a}| = A^* \times (V \to A) \times (U \to \underline{2})$, operator set $\Omega \cup \{Lx \mid x \in V\} \cup \{STx \mid x \in V \cup U\}$, and operations as defined in Chapter X and in example AE. We must, of course, suppose that the operations $\underline{\mathfrak{M}a}_\omega$ for $\omega \in \Omega$ do indeed apply $\underline{A}_\omega$ (recall $\underline{A}$ is the source semantic algebra for $\underline{L}_{arith}$) to the top k elements of the stack (k being the arity of ω) and replace them with the result.

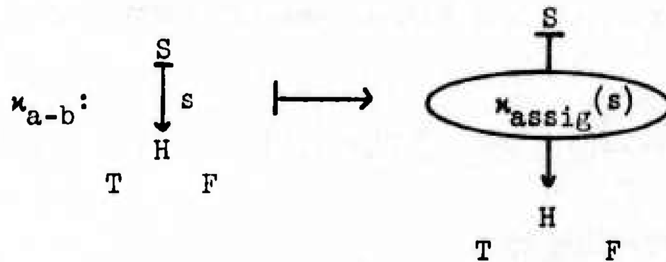
We have also, from example BE, (and with no further specialization than taking the generating set to be U) a compiler $\varkappa_{bool}$ taking the expressions of $\underline{L}_{bool}$ into $\{JTx, JFx \mid x \in U\}$ - BRAs .

We now define a compiler $\varkappa_{a-b}$ (to be a candidate for forming extend($\varkappa_{a-b}, \hat{\underline{L}}_{wc}$)) which acts on, and produces, certain BRAs with set of distinguished nodes {S,H,T,F} as follows:

For $p \in |\underline{L}_{bool}|$,



and for $s \in |\underline{L}_{assig}|$,



The set of BRAs to which κ_{a-b} is applicable are suitable to be run on the machine $\underline{d}\restriction AB$, where $AB = |\underline{L}_{bool}| \cup \{\bar{p} \mid p \in |\underline{L}_{bool}|\} \cup |\underline{L}_{assig}|$, and we may take φ_{a-b} , the semantic function for the set, to be given by

$$\varphi_{a-b}: \underline{\rho} \mapsto ac(\underline{\rho} \otimes \underline{d}\restriction AB) \rfloor (\{S,H,T,F\} \times (V \to A) \times (U \to \underline{2})) \ .$$

We can work this out more explicitly, using the definition of $\underline{d}$, as follows:

$$\varphi_{a-b}\left(\begin{array}{c} S \\ \downarrow \; x := e \\ H \\ T \quad\quad F \end{array} \right): X,i \mapsto Y,i' \quad \underline{\text{iff}} \quad (X = S \ \underline{\text{and}} \ Y = H \ \underline{\text{and}} \ i' = assign^{+}(x, \varphi_{i\restriction V}(e), i) \ \underline{\text{or}} \ X = Y \ \underline{\text{and}} \ i = i') \ ,$$

$$\varphi_{a-b}\left(\begin{array}{c} S \\ p \swarrow \quad \searrow \bar{p} \\ T \quad\quad F \\ H \end{array} \right): X,i \mapsto Y,i' \quad \underline{\text{iff}} \quad (X = S \ \underline{\text{and}} \ i = i' \ \underline{\text{and}} \ (Y = T \ \underline{\text{and}} \ \varphi_{i\restriction U}(p) \ \underline{\text{or}} \ Y = F \ \underline{\text{and}} \ \underline{\text{not}} \ \varphi_{i\restriction U}(p)) \ \underline{\text{or}} \ X = Y \ \underline{\text{and}} \ i = i') \ .$$

The target machine, $\underline{\mathfrak{M}ab}$, for $\varkappa_{a-b}$ is the same as $\underline{\mathfrak{M}a}$ (the target machine for $\varkappa_{assig}$) except that it has the additional operators $\{JTx, JFx \mid x \in U\}$ whose effect is defined by (for $m \in A^*$, $i: (V \to A) \times (U \to \underline{2})$):

$$\underline{\mathfrak{M}ab}_{JTx}: m,i \mapsto m,i \quad \underline{\text{iff}} \quad i: x \mapsto \underline{\text{true}} ,$$

$$\underline{\mathfrak{M}ab}_{JFx}: m,i \mapsto m,i \quad \underline{\text{iff}} \quad i: x \mapsto \underline{\text{false}} .$$

As might be expected, the target semantic function, ψ_{a-b} , is given by

$$\psi_{a-b}: \rho \mapsto ac(\rho \otimes \underline{\mathfrak{M}ab}) \rfloor (\{S,H,T,F\} \times A^* \times (V \to A) \times (U \to \underline{2}))$$

and the decoding function by:

$$\delta_{a-b}: R \mapsto d \circ R \circ e ,$$

where

$$e: X,i \to X,\langle\rangle,i \quad \text{and} \quad d: X,\langle\rangle,i \to X,i$$

(for $X \in \{S,H,T,F\}$, $i: (V \to A) \times (U \to \underline{2})$, and $\langle\rangle$ denoting the empty stack).

We proceed to show that $\varkappa_{a-b}$ (with φ_{a-b} , ψ_{a-b} , and δ_{a-b}) satisfies hypotheses 1-4 for the compiler composition theorem:

1. We have $\{S,T,F,H\} \subseteq \left|\varkappa_{a-b}\left(\begin{array}{ccc} & S & \\ & \downarrow s & \\ & H & \\ T & & F \end{array}\right)\right|$ and

   $\{S,T,F,H\} \subseteq \left|\varkappa_{a-b}\left(\begin{array}{ccc} & S & \\ p \swarrow & & \searrow \bar{p} \\ T & & F \\ & H & \end{array}\right)\right|$, so Hypothesis 1 is satisfied.

2. By identifying X,i with $X,\langle\rangle,i$ ($\langle\rangle$ the empty stack), we have $|\mathscr{L}| = (V \to A) \times (U \to \underline{2}) \subseteq |\underline{\mathfrak{M}ab}| = A^* \times (V \to A) \times (U \to \underline{2})$, so Hypothesis 2 is satisfied.

3. For $\underline{\rho} = \begin{array}{c} S \\ \downarrow x := e \\ H \\ T \quad F \end{array}$, it is clear that

$\delta_{a-b} \circ \psi_{a-b} \circ \kappa_{a-b}(\underline{\rho}) = \varphi_{a-b}(\underline{\rho})$; this is only a slightly disguised form of the statement of correctness for κ_{assig} . It may be noted that the operators JTx , JFx of $\underline{\mathcal{M}ab}$ are never obeyed, as they do not occur in $\kappa_{a-b}(\underline{\rho})$; we have in fact that:

$$\delta_{a-b} \circ \psi_{a-b} \circ \kappa_{a-b}(\underline{\rho}) : X,i \mapsto Y,i' \quad \underline{\text{iff}} \quad (X,i = Y,i' \ \underline{\text{or}}$$
$$X = S \ \underline{\text{and}} \ Y = H \ \underline{\text{and}} \ i' = \text{assign}^{+}(x,\varphi_{i \upharpoonright V}(e),i) \ ),$$

which is exactly the behaviour of $\varphi_{a-b}(\underline{\rho})$.

For $\underline{\rho} = \begin{array}{c} S \\ p \swarrow \searrow \bar{p} \\ T \quad\quad F \\ H \end{array}$, we have somewhat more of an argument to make, because in example BE we took as target machine only a one-state machine, modelling a single interpretation of the Boolean variables. However, we may note that in $\underline{\mathcal{M}ab}$ all the operators JTx,JFx denote partial identity functions (even total identities except on the $(U \rightarrow \underline{2})$ state component) so that, as far as the evaluation of Boolean expressions is concerned, $\underline{\mathcal{M}ab}$ merely unites a number of non-interacting machines of the sort defined in example BE; it follows from the proof of that example that we will have:

$\psi_{a-b} \circ \kappa_{a-b}(\underline{\wp}): X,m,i \to Y,m',i'$ iff $m = m'$ and $i = i'$ and

$(X = Y$ or

$X = S$ and $Y = T$ and $\varphi_{i\uparrow U}(p)$ or

$X = S$ and $Y = F$ and not $\varphi_{i\uparrow U}(p)$).

Similarly, $\delta_{a-b} \circ \psi_{a-b} \circ \kappa_{a-b}: X,i \mapsto Y,i'$ under the same conditions (omitting mention of m), and this is exactly the behavior of $\varphi_{a-b}(\underline{\wp})$. Hence we have satisfied Hypothesis 3.

4. For Hypothesis 4 we have to show that, if $\psi_{a-b} \circ \kappa_{a-b}(\underline{\wp}): X,m,i \mapsto Y,m',i'$ with X and Y both elements of $\{S,H,T,F\}$ and $m,i \in |\mathcal{S}|$, then also $m',i' \in |\mathcal{S}|$; that is, if m is the empty stack, so is m' .

For $\underline{\wp} =$ (S; $p \swarrow$ T, $\bar{p} \searrow$ F; H) this is immediate, since we must have

$m',i' = m,i$. For $\underline{\wp} =$ (S $\downarrow$ $x := e$ H; T F) we have only to recall from

example AE that it was shown that the evaluation of an arithmetic expression starting with an empty stack yielded a one-element stack, and from Chapter X that we defined the store operations so as to remove the top stack element. Hence Hypothesis 4 is satisfied as well.

We may now assert by the compiler composition theorem that our diagram of $\varphi_{a-b}, \kappa_{a-b}, \psi_{a-b}, \delta_{a-b}$ extends to a commutative diagram of homomorphisms, yielding a correct compiler extend$(\kappa_{a-b}, \hat{\underline{L}}_{wc})$ for the

algebra $\hat{\underline{L}}_{wc}$ of AB-BRAs, with generating set

$$\left\{ \begin{array}{c} S \\ \downarrow x := e \\ H \\ T \quad F \end{array} \middle| \, x \in V \cup U \, , \, e \in |\underline{L}_{arith}| \right\} ,$$

and operations

$\hat{\underline{L}}_{wc}(\square)$: P,Q ↦ [diagram: S → P → H S → Q → H; TT with exit T; FF with exit F]

and, for $p \in |\underline{L}_{bool}|$,

$\hat{\underline{L}}_{wc}(\underline{\text{while}}\ p\ \underline{\text{do}})$: Q ↦ [diagram: S H node branching on $\bar{p}$ to F H (exit H) and on p to T S, through Q back to S H; T with exit T; F with exit F] .

(Images under the just-given constructions of isolated T , F , and H nodes were chosen rather arbitrarily. These nodes plainly serve no purpose other than to render the compiler composition theorem slightly less cumbersome to state.)

We now need a compiler $\kappa_{wh}: \underline{L}_{wc} \rightarrow \hat{\underline{L}}_{wc}$ with which we can compose the extend$(\kappa_{a-b}, \hat{\underline{L}}_{wc})$ we have just developed. In essence, the required κ_{wh} is just an instance of the general compiler for whiles and compounds developed in Chapter XI, with $|\underline{L}_{assig}|$ taken for the generating set, and $\{\underline{\text{while}}\ p\ \underline{\text{do}} \mid p \in |\underline{L}_{bool}|\}$ taken as the set of while-statement building

operators. To make the co-domain of κ_{wh} come out to be exactly $\hat{\underline{L}}_{wc}$ we must adjoin the isolated nodes T and F to the BRAs

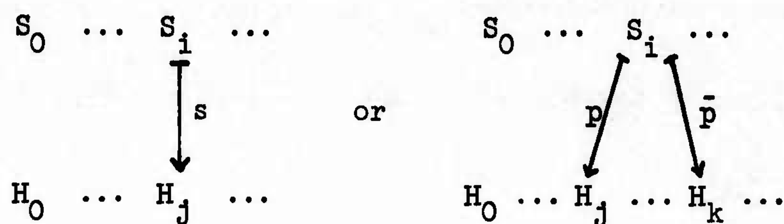
$\kappa_{wh}(x := e) = \overset{S}{\underset{H}{\downarrow}} x := e$, and the isolated node H to the BRAs

$\mathcal{C}_{\underline{\text{while}}\ p\ \underline{\text{do}}} = \overset{S}{\underset{T\qquad F}{p\swarrow\ \searrow\bar{p}}}$, but doing so plainly will not have the

slightest effect on the proof of correctness given in Chapter XI.

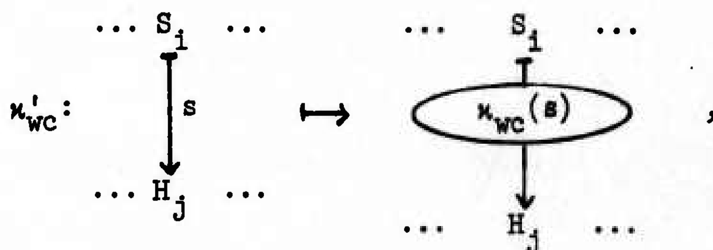
Composing, we may now assert that we have a correct compiler $\kappa_{wc} = \text{extend}(\kappa_{a-b}, \hat{\underline{L}}_{wc}) \circ \kappa_{wh}$ for $\underline{L}_{wc}$, producing programs for our final target machine $\underline{\mathcal{M}ab}$.

We now have to obtain from κ_{wc} a compiler for $\underline{L}_{synth}$. This will be done by an atypical application of the compiler composition theorem, in that no composition will occur: $\underline{L}_{synth}$ is already an algebra of BRAs , and our final compiler κ_{synth} will be simply $\text{extend}(\kappa'_{wc}, \underline{L}_{synth})$ where κ'_{wc} is essentially κ_{wc} , but modified to act on suitable BRAs . (One could easily imagine removing this anomaly by making $\underline{L}_{synth}$ the target algebra of a compiler for a more conventional programming language in which programs were linear strings containing labels and goto statements.) Still postponing a decision on just what algebra $\underline{L}_{synth}$ is to be, we will proceed to show that the compiler composition theorem must be applicable, assuming only that all the BRAs in the generating set of $\underline{L}_{synth}$, together with any constant BRAs $\mathcal{C}_\omega$ it may employ in its constructions, are of one of the forms:

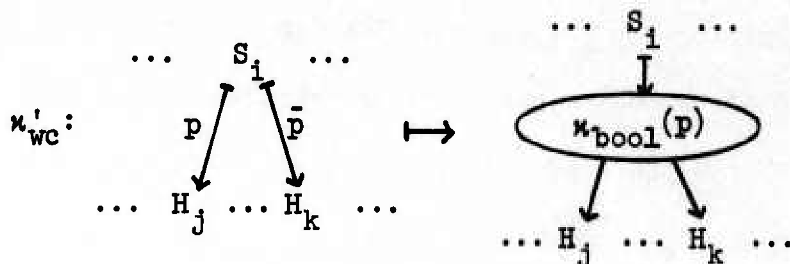


with $s \in |\underline{L}_{wc}|$, $p \in |\underline{L}_{bool}|$, and without any implication that $k > j$. We denote this set of BRAs by X_{synth} .

We specify κ'_{wc} in the obvious way by:



and



Our target semantic function will of course be given by:

$$\psi_{synth}: \underline{P} \mapsto ac(\underline{P} \otimes \underline{\mathcal{M}ab}) \rceil (SH \times A^* \times (V \to A) \times (U \to \underline{2})) \quad ,$$

and our decoding function by:

$$\delta_{synth}: R \mapsto R \rceil (SH \times (V \to A) \times (U \to \underline{2}))$$

(keeping in mind our convention that $\{i \mapsto \langle\rangle, i\}: |\mathscr{A}| \subseteq |\underline{\mathcal{M}ab}|$).

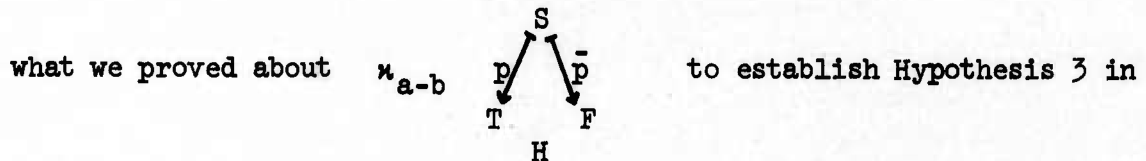
We recall that $\varphi_{synth}: \underline{P} \mapsto ac(\underline{P} \otimes \mathscr{A}) \rceil (SH \times (V \to A) \times (U \to \underline{2}))$.

We may now verify Hypotheses 1-4:

Hypothesis 1, that $SH \subseteq |\kappa'_{wc}(\underline{P})|$ for $\underline{P} \in X_{synth}$, is immediate by definition of κ'_{wc}.

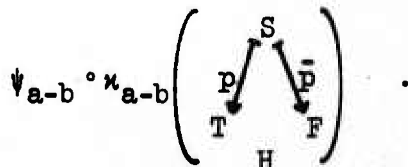
Hypothesis 2, that $|\mathcal{A}| \subseteq |\mathfrak{M}ab|$, holds as before by identification of i with $\langle\rangle,i$.

Hypothesis 3, that $\varphi_{synth}(\underline{P}) = \delta_{synth} \circ \psi_{synth} \circ \kappa'_{wc}(\underline{P})$ for $\underline{P} \in X_{synth}$, is easily seen to follow from the correctness of κ_{wc}, together with



what we proved about κ_{a-b} (diagram: S, p, $\bar{p}$, T, F, H) to establish Hypothesis 3 in the foregoing proof for κ_{a-b}; all that has changed is that we are allowing the role of S to be played by an arbitrary S_i, and those of T, F, and H by arbitrary H_j and H_k.

Finally, Hypothesis 4, that relations $\psi_{synth} \circ \kappa'_{wc}(\underline{P})$, if applied to states with empty stack, yield only states with empty stack, comes as a by-product of the application of the compiler composition theorem to κ_{a-b} (recall that the inductive part of the proof of that theorem extends not only Hypothesis 3 to the whole algebra, but Hypothesis 4 as well), together with the already proved satisfaction of Hypothesis 4 by



$\psi_{a-b} \circ \kappa_{a-b}$ (diagram: S, p, $\bar{p}$, T, F, H) .

We may conclude, by the compiler composition theorem, that once we settle on $\underline{L}_{synth}$, $\kappa_{synth} = \text{extend}(\kappa'_{wc}, \underline{L}_{synth})$ will exist and be correct.

It remains to specify a set of constructions which will, from our generating set for $\underline{L}_{synth}$, yield all and only the multi-entrance, multi-exit deterministic flowcharts, where by "deterministic" we mean that at most either one arc labelled with a statement from $\underline{L}_{wc}$, or a pair of arcs labelled with p and $\bar{p}$, where $p \in |\underline{L}_{bool}|$, is allowed to depart from any one node. (The restriction to determinism is quite unnecessary to the success of our correctness proof, and is made in the interests of realism: that is, we imagine that the "real" machine modelled by $\underline{\mathfrak{M}ab}$ is only able to execute deterministic programs -- it may easily be checked that it will only be called upon to do so, provided we keept the $\underline{L}_{synth}$-programs deterministic.) It appears that the following set of constructions will do as well as any: we take for the operations of $\underline{L}_{synth}$ __all__ binary constructions derived from kernels of the form:

$$\tilde{q}: SH \dot{\cup} SH \rightarrow SH \cup \{I_0, I_1, \ldots\}$$

such that

(i) every S_i has at most one inverse image under $\tilde{q}$ in (one component of) $SH \dot{\cup} SH$, which is of the form S_k ;

(ii) every I_i has either empty inverse image, or inverse image consisting of exactly one S_k and one or more $H_{j_1}, H_{j_2}, \ldots$;

(iii) every H_i has inverse image under $\tilde{q}$ consisting of zero or more $H_{j_1}, H_{j_2}, \ldots$.

The idea is that we only allow arcs to depart from S-nodes and I-nodes and we prevent these from coalescing.

The generating set of $\underline{L}_{synth}$ we naturally take to be X_{synth} .

Having finally fixed on $\underline{L}_{synth}$, we may say that extend(κ'_{wc},$\underline{L}_{synth}$) is a correct compiler for $\underline{L}_{synth}$, yielding programs for $\underline{\mathfrak{M}ab}$; our synthesis is finished. ∎

XIII. Conclusion

We may ask how well this thesis has conformed to the aim, announced in the introduction, of bringing mathematical order as well as mathematical rigor to a part of the theory of computation. A partial affirmative answer is given by the fact that a very short list of well-defined ideas provides the basis for the examples of correctness proofs which we have seen; the fertility of these ideas appears far from exhausted:

1) The diagram of operator-algebraic homomorphisms as a model for the compiler correctness problem, taken from [Burs 69].

2) The category of BRAs, obtained by combining the concept of the $\otimes$ operation from [Land 70] with the idea that interpretation of a program scheme is a functor from [deB 69].

3) The result of Chapter VI, that the semantics of BRAs acts as a homomorphism on a construction-algebra of BRAs.

4) The compiler composition theorem.

Miscellaneous Observations

The algebra $\underline{L}_{synth}$ which we chose, for want of a better, in Chapter XII did not impose any natural structure on flowcharts; indeed the idea of multi- (rather than single-) entrance flowcharts, which was forced on us because there appears to be no way to break up a single-

entrance flowchart into single-entrance pieces, is itself somewhat unnatural. However, in [Cooper 71] it is shown that every flowchart simulates (although the notion of simulation does not appear there explicitly) a flowchart in what is there called "block form" -- i.e., a tree form, except that an arc may lead back from any leaf to one of its ancestors -- and "reasonable", i.e., tree-structure-reflecting, algebraic operations are given for generating the block-form flowcharts. It seems probable, therefore, that a more perspicuous approach to the algebraic treatment of the compilation of flowcharts would be to define first a compiler (homomorphism) for block-form flowcharts, and then to show that it can be extended to a functor between two categories of BRAs, i.e., that it can be extended to arbitrary flowcharts in such a way as to preserve simulations. The potential of this approach remains to be investigated.

It is a question of some importance whether optimizing compilers, which, particularly when they use global information about the source program, are superficially very unlike homomorphisms, can be rendered amenable to algebraic description. The author speculates that many cases of optimization will allow description as an underlying non-deterministic compiler (i.e., one computing a relation between source and target language programs rather than a function) under the control of a "black box" which selects one of the many possible compiling functions. If we could prove the underlying compiler correct for all its possible outputs, then we could claim correctness for the optimizing compiler without ever concerning ourselves with the contents of the black box.

As we have defined it, a homomorphism is of course required to be a function. It seems very probable, however, that the following property, if required of compiling relations, would make them sufficiently like homomorphisms to allow analogous results to those of this thesis to be obtained -- (a property of a relation ρ, for particular Ω-algebras $\underline{A}$ and $\underline{B}$ with $\rho: |\underline{A}| \nrightarrow |\underline{B}|$ and for all $\omega \in \Omega$ of any arity k):

$$\rho: \underline{A}_\omega(a_1,\ldots,a_k) \mapsto b \ \underline{\text{iff}}\ \underline{\text{for}}\ \underline{\text{some}}\ b_1,\ldots,b_k[\rho: a_j \mapsto b_j\ (1 \le j \le k)\ \underline{\text{and}}\ b = \underline{B}_\omega(b_1,\ldots,b_k)] \ .$$

This notion is a special case of that of "pseudohom" defined by Lloyd in [Llo 72], for which he is able to prove a unique extension lemma.

The following remarks develop informally the claim made in the introduction that Dijkstra's co-operating sequential processes [Dijk 68] can be naturally modelled by BRAs. We may define an operation $\check{\times}$ on BRAs by:

$$|\mathcal{A} \check{\times} \mathcal{B}| = |\mathcal{A}| \times |\mathcal{B}| \ ,$$

$$(\mathcal{A} \check{\times} \mathcal{B})_\gamma: a,b \to a',b' \ \underline{\text{iff}}\ \mathcal{A}_\gamma: a \to a' \ \underline{\text{or}}\ \mathcal{B}_\gamma: b \to b' \ .$$

It is readily verified that $\check{\times}$ is a bifunctor; it appears to play a natural role in the assembly of machines from components. (Had we troubled to define $\check{\times}$ earlier, we might have been spared some of the tediousness of assigning identity relations to "extraneous" operators in the analysis of the machine in Chapter X into stack and location components.)

It appears that if we have two programs modelled by BRAs $\mathcal{A}$ and $\mathcal{B}$, then the BRA which models their concurrent operation is simply $\mathcal{A} \check{\times} \mathcal{B}$

Moreover, a semaphore $\mathscr{s}$ is just a component (under $\check{\times}$) of a machine, with $|\mathscr{s}| = \mathbb{N}$, and operators Vs , Ps having the effects:

$$\mathscr{s}_{Vs}: n \mapsto n+1 \quad ,$$

$$\mathscr{s}_{Ps}: n+1 \mapsto n \quad .$$

The preceding explication follows Dijkstra's concept in an ugly but perhaps essential characteristic, that what are the possible computations of a set of programs running concurrently is crucially dependent on just what is taken to be an atomic act of computation. (For example, if one supposed that accessing the value of a variable decomposed into destructive readout followed by restore, then programs which Dijkstra considers to have determinate outcomes would cease to do so.) This property is reflected in the BRA-model by the fact that the operation $\check{\times}$ does not commute with compilation.

Prognosis

An attempt will here be made to evaluate the practical applicability of the algebraic methods which have been developed in the foregoing chapters.

First of all it is plain that the "stratified" kind of semantics to which we have been limited, and according to which all the constructs (at the top level) of a language-algebra must be of the same type, is a serious obstacle to the treatment of "realistic" languages. However, the work of Birkhoff [Birk 70] and [Birk 71], which the author saw too late for it to be reflected in the development above, seems to hold out hope

for a great amelioration of this difficulty. Birkhoff introduces the notion of a "heterogeneous algebra", essentially an algebra with several carrier sets, which is to say several types of element; each operation has not only a numerical arity, but as well a characteristic type for each argument and for its result. E.g., a language containing both expressions and statements, and allowing each to be embedded in the other, could be modelled as a single heterogeneous algebra, as could its set of meanings which correspondingly would contain functions of diverse types.

It appears that the notions of generating set and homomorphism are extended to heterogeneous algebras in such a way that the elementary theorems of universal algebra, and in particular the unique extension lemma, are preserved. It seems reasonable to expect, therefore, that the methods for proving compiler correctness which we have developed will remain valid in the heterogeneous context. Birkhoff also has some general insights about derived operations, of which the constructions we have defined are a special case (at least if we regard all BRAs as originally forming an operator algebra, say with $\otimes$ and $\oplus$ as operations). Homomorphisms such as our compilers for which the target algebra operations are not given in advance but created to "go with" the homomorphism seem to be what Birkhoff calls "cryptomorphisms".

There is also the question whether some particular kind of sophisticated programming language feature will cause algebraic methods of description to break down, or at best become terribly unwieldy. The author's expectation is that the modelling of arrays and other data structures will not present any great difficulty; he further conjectures

that at least a limited form of closed subroutine facility can, with some ingenuity, be modelled directly in BRAs. Languages in which bound variables play an essential role (e.g. those having dynamic declarations, or procedures with formal parameters) may present graver difficulties. It appears that the best prospect of coping with these is to take the meanings of program phrases to be appropriate functions of environments, as outlined in the remarks on λ-calculus semantics given above in Chapter IV.

It is clear that the attempt to produce an algebraic proof for a typical existing compiler will get nowhere; even if a homomorphism is what is "really" being computed, that fact is usually well hidden. The author expects that practical application of the methods developed here will come, if at all, within the framework of a "verifying compiler-compiler" -- i.e., a compiler-writing system which accepts algebraic descriptions of source and target languages and a definition of the compiling function as a homomorphism (or rather, like our κ_{synth}, as a complex edifice built by extension and composition from homomorphisms) and which produces a compiling program. The system envisioned here would further accept algebraic specifications of source and target language semantics, and be able to verify assertions accompanying the compiling specification which would constitute a proof of the produced compiler's correctness.

This thesis plainly contains only a part of the groundwork which must be done before a verifying compiler-compiler can be produced. A significant part of the effort entailed in creating it would be the devising of heuristic techniques for generating an efficient structure

of passes and phases in the produced compiler from the numerous fragments of the compiling function specified by the user. Incorporated in this structure there would of course have to be an automatically generated parser, similar to those produced by existing compiler-compilers, which would produce from a concrete (string-of-characters) source program, elements of the source language algebra corresponding to each of its phrases.

Birkhoff's generalized notion of (heterogeneous) algebra should almost certainly be the one incorporated in the verifying compiler-compiler. Furthermore, the restriction that the target of any compiling homomorphism must be an algebra of BRAs should certainly not be made; even if this were generally true of the final target algebra, it would probably be appropriate for most higher-level languages that the first several steps of compilation should be into intermediate languages in which the sequential nature of the ultimate computation was still partially hidden. Indeed, for a language some of whose features were definitional extensions to a kernel sub-language, the first steps of compilation might well be endomorphisms.

A problem which has been totally ignored in this thesis, but whose solution is essential to the verification of a real-world compiler, is that of making the transition at the output end of compilation from BRAs, however machine-language-like they may appear, to programs for some real-world machine. Problems of memory allocation arise here, e.g. of assigning parts of a homogeneous store to program, variable values, and stack. Also, some device must be found by which we may accept as correct a target program which does its best within the limits of

available memory, but comes to an error stop when space is exhausted. The author expects that this part of the compiling problem will not prove intractable, but on no other than intuitive grounds.

Finally, of course, a verifying compiler-compiler would have to incorporate a proof checker capable of appreciating the reasoning about algebras, relations, and functions on which the correctness of compilers might depend. The most promising work in this area with which the author is acquainted is the ongoing development by Milner [Miln 72] of the LCF system, an implementation of a logic for computable functions due to Dana Scott.

References

[Birk 70] G. Birkhoff and J. D. Lipsom, "Heterogeneous Algebras," Journal of Combinatorial Theory 8, pp. 115-133, 1970.

[Birk 71] G. Birkhoff, "The Role of Algebra in Computing," in Computers in Algebra and Number Theory, vol. IV, SIAM-AMS Proceedings, American Mathematical Society, 1971.

[Burs 69] R. M. Burstall and P. J. Landin, "Programs and their Proofs: an Algebraic Approach," in Machine Intelligence 4, (B. Meltzer and D. Michie, eds.), Edinburgh University Press, 1969.

[Burs 72] R. M. Burstall, "An Algebraic Description of Programs with Assertions, Verification, and Simulation," in Proceedings of an ACM Conference on Proving Assertions About Programs, SIGPLAN Notices 7, 1, Association for Computing Machinery, 1972.

[Cooper 71] D. C. Cooper, "Programs for Mechanical Program Verification," in Machine Intelligence 6, (B. Meltzer and D. Michie, eds.), Edinburgh University Press, 1971.

[deB 69] J. W. deBakker and D. Scott, "A Theory of Programs," (mimeographed notes), IBM Seminar, Vienna, August 1969.

[Dijk 68] E. W. Dijkstra, "Co-operating Sequential Processes," in Programming Languages, (F. Genuys, ed.), NATO Advanced Study Institute, Villard-de-Lans, 1966, Academic Press, London and New York, 1968.

[Hoare 69] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," Communications of the ACM 12, 10, October 1969.

[Karp 59] Richard Karp, "Some Applications of Logical Syntax to Digital Computer Programming," Ph.D. Thesis, Harvard University, 1959.

[Land 70] P. J. Landin, "A Program Machine Symmetric Automata Theory," in Machine Intelligence 5, (B. Meltzer and D. Michie, eds.), Edinburgh University Press, 1970.

[Llo 72] C. Lloyd, "Some Concepts of Universal Algebra and their Application to Computing Science," Computing Science Working Paper: CSWP-1, University of Essex, February 1972.

[MacL 67] S. MacLane and G. Birkhoff, Algebra, Macmillan, New York, 1967.

[McC 62] J. McCarthy, "Towards a Mathematical Science of Computation," Proceedings of the ICIP, 1962.

[McC 63] J. McCarthy, "A Basis for a Mathematical Theory of Computation," in Computer Programming and Formal Systems, (P. Braffort and D. Hirschberg, eds.), North-Holland, Amsterdam, 1963.

[Miln 72] R. Milner, "Logic for Computable Functions: Description of a Machine Implementation," Stanford University Artificial Intelligence Project Memo AIM-169, Stanford University, 1972.

[Paint 67] J. A. Painter, "Semantic Correctness of a Compiler for an Algol-like Language," Ph.D. Thesis, Stanford University, 1967.

[Scott 69] D. Scott, "A Construction of a Model for the λ-Calculus," (mimeographed notes), Oxford Seminar, November 1969.