Paul M. Jones

IBM Research

REDUCTION LANGUAGES AND VARIABLE–FREE PROGRAMMING

John Backus

April 7, 1972

RJ 1010

Yorktown Heights, New York

San Jose, California

Zurich, Switzerland

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication elsewhere and has been issued as a Research Report for early dissemination of its contents. As a courtesy to the intended publisher, it should not be widely distributed until after the date of outside publication.

Copies may be requested from:
IBM Thomas J. Watson Research Center
Post Office Box 218
Yorktown Heights, New York 10598

REDUCTION LANGUAGES AND VARIABLE-FREE PROGRAMMING

by

John Backus

IBM Research Laboratory
San Jose, California

Dept K08 Research
Monterey & Cottle Roads
San Jose, CA 95114

ABSTRACT: Classification of programming languages: Programming languages are classified according to their semantics: Complete languages associate a meaning or value with a formula. Primary languages have a transition function on formulas; repeated application of it produces the meaning of a formula, if it exists. Secondary languages employ an auxilliary "state" language, which is primary, to give meaning to their formulas. Reduction languages are primary and have properties which assure that it is easy to find the meaning of a formula.

Red languages: Red languages are a particular series of reduction languages. Their syntax and especially their semantics are believed to be simpler than those of existing languages: They employ no variables, no GO TO statements, no lambda-expressions, and they have no built-in prescription for evaluating self-invoking functions. They can nevertheless express any computable function. They employ two kinds of composition for functions, regular composition and meta composition. These permit the definition of powerful modifiers which alter the behavior of other operators and make possible the introduction of an unprecedented variety of facilities (continued next page)

RJ 1010 (#17246)
April 7, 1972
Computer Sciences

P.73

ABSTRACT, CONTINUED:

without altering the syntax or basic semantic rules of Red languages and without applying the function-valued functions required in lambda-notation-based languages.

REDUCTION LANGUAGES AND VARIABLE-FREE PROGRAMMING

TABLE OF CONTENTS

INTRODUCTION 1
General 1

Classification of programming languages 1
Complete languages 1; Incomplete languages 2; Primary languages 2; Secondary languages 3; Programming languages 4; Reduction languages 4.

Red languages 5
General 5; Structure and denotation 5; Regular composition of operators 6; Meta composition of operators 7; Modifiers of operators 8; Extensibility and definability within a fixed language 10; Variable-free languages 11; Efficiency 13.

CLASSIFICATION OF PROGRAMMING LANGUAGES 14

General 14

Primary languages 14
Definition of primary languages 14; The semantic function, ρ, of a primary language 15; Example 1: The primary language for a Turing machine 15; Example 2: The primary language for a computer 16; Remarks 17.

Secondary languages 18
Definition of secondary languages 18; Example 3: The secondary language for the programs of a computer 18; Remarks 19.

Complete secondary languages 21
Definition of complete secondary languages 21; The semantic function, ω, of a complete secondary language 21; Example 4: The complete secondary language for applicative expressions 22; Remarks 23.

Reduction languages 24
General 24; preliminary definitions 28; Definition of reduction languages 30; Example 5: A reduction language for simple expressions 30; Theorems about reduction languages 31; Remarks 32.

Parenthetic languages 34
Preliminary definition: factors of a formula 34; Definition of parenthetic languages 34; Theorems

about parenthetic languages 35; Remarks 43.

RED LANGUAGES 45

Introduction 45

Syntax of Red languages 45
General 45; Syntax of L: formulas 45; Syntax of L: constants 46; Syntax of L: notation 46; Syntax of L: examples of constant items 47; Syntax of L: examples of non-constant items 47.

Semantics of Red languages 47
General 47; The restricted transition function for Red languages 48; Notation 49; Description of functions 50; Definition of the restricted transition function, Δ 50; Examples 51.

THE RED/1 LANGUAGE 55

Introduction 55

Syntax 55
Notation 55; Basic syntax 56; Syntax of atoms 56; constants 57.

Semantics 57
General 57; Variables and their ranges 58; Prefixed sets 58; Description of functions 59; Description of ρ 59; Description of Δ 60; Description of ψ 60; Description of ϵ 60.

Red/1 Summary 65
Basic syntax 65; Syntax of atoms 65; Constants 65; Non-constants 65; Description of ℝρ 65; Description of Δ 66; Description of ψ 66; Description of ϵ 66; Remarks 68.

Examples of defined operators 69
Notation 69; Definitions of operators 71; Analysis of examples 71; Remarks 76; Defined operators 76.

RED/2 LANUGAGES 80

Introduction 80
Language extensions 80.

Syntax 81
Basic syntax 81; Syntax of atoms 82; Constants.

Semantics 82
General 82; Conversion to Red/1 synonyms 82; The defining set ∇ of a Red/2 language 83; Example 84;

Description of ρ and Δ 84; Description of ψ 85; Description of ϵ 85; Remarks 86; Conversion of Red/1 definitions to Red/2 definitions 87.

Variable-free programming techniques, universal operators, parsing, and controlling order of execution 89

Universal operators 89; Variable-free programming techniques 90; Preliminary definitions 91; The semantic operator for Red/2 and auxilliary operators 94; Parsing 96; Controlling order of execution 96.

RED/3 LANGUAGES 97

Introduction 97; Syntax and semantics 98; Sampler of Red/3 operators 98; Sampler of Red/3 modifiers 99; Examples of Red/3 defined operators 101.

OTHER RED-LIKE LANGUAGES; COMPARISONS WITH OTHER SYSTEMS 105

A red-like complete secondary language 105; comparison of combinators and Red operators 107; Comparison of AE's and Red languages 108; Relation to other work 109; Acknowledgements 110; References 111.

REDUCTION LANGUAGES AND VARIABLE-FREE PROGRAMMING

ERRATA (5/2/72)

Page 6, line 7 from below:

...analagous... := ...analogous...

Page 31, 4th line after "Theorem Th/RE1"

proof := proof By reductio ad absurdum

Page 31, Theorem Th/RE1, line #3

case 1 := case 1: Assume the theorem false

Page 31, Theorem Th/RE1, line #6

...//2, 3, RE3 := ...//1, 2, 3, RE3

Page 45, last line

...a symbol or a string. := ...a symbol [a sequence of characters] or a string [a quoted sequence of characters and system-characters].

Page 56, lines 3, 4 and 5 from below

S-ELEMENT := *STRING-ELEMENT*

S-LIST := *STRING-LIST*

Page 59, last line

The last line should be replaced by the following three lines:

$\rho(A\ \langle\langle DBL\ C\rangle\ \langle D\ E\rangle\rangle\ \langle F\ G\rangle) = (A\ \langle\Delta\langle DBL\ C\rangle\ \langle D\ E\rangle\rangle\ \langle F\ G\rangle)$

$= (A\ \langle(C\ C)\ \langle D\ E\rangle\rangle\ \langle F\ G\rangle)$

Where $(C\ C)$ is the value of $\Delta\langle DBL\ C\rangle$.

Page 75, line 2

→ *CONDARG*.[(... := → ψ*CONDARG*.[(...

Page 80, line 5

...defining set. := ...defining set whose effect is similar to that of a library of definitions.

Page 83, rule 3), line 3

...not enclosed in string-quotes by... := ...not enclosed in string-quotes, by...

Page 84, Example

The defining set is incomplete since the definition of *FST* employs *ROT* but the defining set does not define *ROT*.

Page 87, line 12

...meaningless. := ...useless.

Page 89, line 6

...interprenation... := ...interpretation...

Page 90, line 5

a-prefixed pairs... := *A*-prefixed pairs...

Page 94, line 8 from below

PSAP.*X* → *DELTA*.*X* should be replaced by the following lines:

PSAP.*X* →

AND.(*PSCON*.∘1.*X* *PSCON*.∘2.*X*) → *DELTA*.*X*

PSCON.∘1.*X* → *A*(∘1.*X* *RHO*.∘2.*X*)

T → *A*(*RHO*.∘1.*X* ∘2.*X*)

Page 94, line 2 from below

PSAP → *DELTA*) should be replaced by the following lines:

PSAP → [*CONDARG*

AND:(2*F* *PSCON*:∘1 *PSCON*:∘2) → *DELTA*

PSCON:∘1 → *SEC*:*RHO*

C:*T* → *FST*:*RHO*])

Page 103, lines 8 and 9 from below

The two instances of the set (0 9 11 12) should be replaced by the set (0 6 10 12 13).

REDUCTION LANGUAGES AND VARIABLE-FREE PROGRAMMING

ERRATA (5/8/73)

Does not include errata of 5/2/72 (p. iv, v of RL&VFP)

Page 7, line 6 from below:

...composition *F*,... → ...composition, *F*,...

Page 22, line 13

...form:t(X:(),... → ...form: (X:(),...

Page 28, line 10 from below:

All variables will... → All variables [e.g., *F*,*G*,*H*, etc.] will...

Page 53, line 12:

...another whose... → ...another operator whose...

Page 65, line 10 from below:

...: Applications. → ...: Applications and sets with non-constant members.

Page 66, line 3 from below:

($\underline{X}$) → $\underline{S}(\underline{X})$

Page 67, line 8 from below:

...$\alpha\underline{C}2...\underline{C}n$) → ...$\alpha\underline{C}2...\alpha\underline{C}n$)

Page 68, line 3 from below:

...none are needed... → ...none is needed...

Page 103, line 5 from below:

...each member... → ...each non-solitary member...

Page 107, line 1:

...ambiguity... → ...allowable variation...

965-IBM-04

INTRODUCTION

General

The principal object of this paper is to present an interesting new class of programming languages called reduction languages and a particular series of these: Red languages. In order to make clear the nature of these new languages, the first section of the paper discusses the methods by which programming language semantics are specified and classifies languages accordingly. The subsequent sections describe various Red languages and give examples of the variable-free programming which they employ.

This introductory section is a summary of the main sections of the paper except the last section which compares Red languages and other languages and logical systems.

Classification of programming languages

Complete languages

In one dimension, some programming languages are "complete"--their semantics specify for many "programs" a "result" which is the "meaning" of the original program and is itself a "program". Complete languages with the power of computability are rare. One example is Landin's language of "applicative expressions" or AE's [2]. Each "program" or AE is either "meaningless" [roughly speaking, non-terminating] or has a value which is an AE and the meaning of the original.

Incomplete languages

Most programming languages are "incomplete". For example, most Algol programs do not have a specific result. The only meaning one can give to an Algol program is to specify the sequence of actions which will take place when it is executed. The knowledge about which variables are the input of a program and which are output is known to the user perhaps, but is not part of the formal semantics of Algol. Further, the output is not in general an Algol program.

Primary languages

In another dimension, a programming language is "primary" or "secondary". A primary language is self-contained; it has a transition function defined for its programs or formulas which, when repeatedly applied to a formula, will ultimately produce a formula which is the meaning of the original--if a meaning exists. Thus the meaning of a formula is found by transitions entirely within the space of formulas. Primary languages are necessarily complete.

Two systems of logic, Curry's system of combinators [3] and Church's calculus of lambda-conversion [9], among others, are capable of being described, in their essentials, as primary languages. However, the requisite transition functions are quite complicated to describe completely or to program. Although several programming languages, e.g., McCarthy's LISP [8] and Landin's AE's [2], make use of lambda notation, all have avoided the complicated transition

they avoided this for efficiency, not because it was too hard.

function needed for their primary language formulation. Instead, they have chosen the simpler task of formulating them as "secondary" languages [see below]. Thus it is perhaps fair to say that up to now there have been no "actual" primary programming languages.

Secondary languages

A secondary language provides no transition function which maps formulas into formulas and which ultimately produces the meaning or value of a given formula. Instead, for example, the process of finding the meaning of an AE, is given by leaving the space of AE's and describing transitions in a more complex space--the space of "states". These states are the quadruples (S,E,C,D) which record not only various augmented AE's and fragments thereof, but also the values of variables. The process of finding the meaning of an AE, X, takes place by state transitions of the SECD machine, after first embedding X in the initial state of the machine. When a terminal state is reached after repeated transitions, the meaning of X can be extracted from it.

The states of the SECD machine and its transition function on states [or formulas] form a primary language: the meaning of an initial state, if any, is the terminal state obtained from it. However, this "state language" is not a programming language in the following viewpoint-determined sense: the objects--states--which receive meaning are not the objects we want to have meaning. The state language exists solely to give meaning to AE's;

the notion of the meaning of a state is of no other interest.

Thus all current programming languages are secondary languages, "secondary" since their semantics employ a state language, itself primary, whose formulas are not those of the programming language.

Programming languages

In this paper we shall use the term "programming language" loosely, but in general we shall mean either (a) an incomplete language such as a machine language or Algol or (b) a complete language whose semantic function gives meaning to the formulas of interest to the user [as distinct from a state language]. In both cases we expect the language to be capable of expressing any computation.

Reduction languages

A reduction language is a primary language, and therefore complete, with additional properties which make it especially easy to find the meaning of a formula. These properties are typified by those of completely bracketed arithmetic expressions which involve only constants and operators: one can replace any subexpression of an expression by its value and: (a) still have an expression, (b) not change its value and (c) be closer to the result [unless the subexpression is its own value]. While there are many trivial examples of such reduction languages, they all lack the power to describe an arbitrary computable

function and are not adequate as full programming languages.

The axioms for reduction languages cover much richer systems. And they assume a rough equivalent of the Church-Rosser theorem for lambda-conversion [10], i.e., that all choices of the order of performing reductions lead to the same result. Another class of languages, called parenthetic languages, is defined whose axioms are easily satisfied, and it is then proved that this latter class is a subclass of reduction languages.

It has not been recognized, I believe, that reduction languages can be constructed which are unusually simple in both their syntax and their semantics and yet have the power to express any computable function. Several particular reduction languages, called Red languages, are exhibited in this paper in the hope of convincing the reader of this thesis.

Red languages

General

In addition to the properties of reduction languages indicated above, Red languages are characterized by the following properties.

Structure and denotation

There are no variables, no GO TO statements, no lambda-expressions, and no built-in prescription for evaluating self-invoking functions in the basic semantics of

Red languages. The last fact contrasts with the need for a LABEL operator in LISP [8] and the Y function in AE's [2].

The structure of a Red formula or "item" is simple. It is either an atom or an ordered set or an "application"--an ordered pair denoting the result of applying its operator to its operand. Any item may appear as a member of a set or as the operator or operand of an application. The only kind of item which does not denote itself is an application, which denotes its result. All operators have a single operand or argument [which of course may be a set of one, two, or more members].

Regular composition of operators

If applications are written as

<*X Y*>

and sets as

(*X*1 *X*2...*X*n)

then the Red formula

<(*F G H*) *X*>

is analogous to the mathematical expression for the composition of three functions:

fgh(x)

and has the same meaning as

<*F* <*G* <*H X*>>>

just as the mathematical expression means

f(g(h(x)))

Meta composition of operators

Red languages have two rules of composition, regular composition as above, and meta composition. It is the latter which enables Red languages to escape the expressive limitations of simple expressions.

Some items can be made "meta" items by affixing the prefix * . Thus if *X* is a regular item, then

$$*X$$

is the corresponding meta item. The composition rule used to "reduce" an application with a composite operator *(F G H)* is determined by whether the first of the composed operators is meta or regular [i.e., not meta]. Thus if *F*, *G*, and *H* are regular items, then regular composition is used:

$$<(F\ G\ H)\ X>$$

$$\rightarrow <F\ <(G\ H)\ X>>$$

$$\rightarrow <F\ <G\ <(H)\ X>>>$$

$$\rightarrow <F\ <G\ <H\ X>>>$$

[we use "→" to indicate a transition, here the one for regular composition]. Whereas, if the first member is meta, the transition for meta composition is used:

$$<(*F\ G\ H)\ X> \rightarrow <F\ ((*F\ G\ H)\ X)>$$

After a transition for meta composition, *F*, the regular part of the meta operator **F*, has as its operand the pair which was the original application. The options for *F* are then many. Typically, it can rearrange the original operator and/or operand and then cause the altered operator to be applied to the altered operand. One option, of course, is

to cause **F* to be applied to some operand formed from *G*, *H*, and *X*.

Modifiers of operators

Meta composition allows one to form meta operators which modify the behavior of other operators. One simple but useful modifier is the meta operator "first", *FST*. When paired with any other operator, *FST* causes the other to act, not on the operand as a whole, but on its first member. Thus if *DBL* does the following:

$$\langle DBL\ X\rangle \rightarrow (X\ X)$$

then

$$\langle (FST\ DBL)\ (X\ Y)\rangle$$
$$\rightarrow (\langle DBL\ X\rangle\ Y)$$
$$\rightarrow ((X\ X)\ Y)$$

In more detail, suppose we have the operators *APPLY* and *REGRP*, where:

$$\langle APPLY\ ((X\ Y)\ Z)\rangle \rightarrow (\langle X\ Y\rangle\ Z)$$
$$\langle REGRP\ ((W\ X)\ (Y\ Z))\rangle \rightarrow ((X\ Y)\ Z)$$

Then, let *FST* = *(*APPLY REGRP*) and we have:

$$\langle (FST\ DBL)\ (X\ Y)\rangle =$$
$$\langle (*(APPLY\ REGRP)\ DBL)\ (X\ Y)\rangle$$
$$\rightarrow \langle (APPLY\ REGRP)\ ((*(APPLY\ REGRP)\ DBL)\ (X\ Y))\rangle$$
$$\rightarrow \langle APPLY\ \langle (REGRP)\ ((*(APPLY\ REGRP)\ DBL\ )\ (X\ Y))\rangle\rangle$$
$$\rightarrow \langle APPLY\ \langle REGRP\ ((*(APPLY\ REGRP)\ DBL)\ (X\ Y))\rangle\rangle$$
$$\rightarrow \langle APPLY\ ((DBL\ X)\ Y)\rangle$$
$$\rightarrow (\langle DBL\ X\rangle\ Y)$$
$$\rightarrow ((X\ X)\ Y)$$

The first transition above is that for meta composition, the

next two are for regular composition, and the last three are for "primitive application", the application of primitive operators. These reductions illustrate the way to find the meaning of any formula: find an innermost application and reduce it using a composition rule or applying a primitive; repeat until all applications are eliminated.

More complex and more useful modifiers than *FST* may be put together. One may have the modifier *REPEAT* such that

<(*REPEAT OP*) (*N X*)> → <(*OP OP...OP*) *X*>

Thus *REPEAT* causes *OP* to be applied *N* times to *X*. Or there is *CONDARG* such that, for example

<(*CONDARG F1 G1 F2 G2*) *X*>

→ <*G1 X*> if <*F1 X*> is "true"

<(*CONDARG F2 G2*) *X*> if <*F1 X*> is "false"

Thus *CONDARG* provides one Red equivalent of the conditional expression of LISP [8].

The ability to define an unlimited number of powerful modifiers of various types is one of the most interesting features of Red languages. They offer the chance to provide almost any conceivable semantic ability without the slightest change in the simple syntax of Red languages. Contrast this ability with that of Algol, for example. If Algol lacked its <u>for</u> statement, it would be impossible to introduce a procedure within Algol which would provide the facilities of <u>for</u> without first introducing new syntactic structures.

Extensibility and definability within a fixed language

Red languages have the ability to introduce modifiers and other operators for desired new semantic facilities at many levels without adding to their spare, simple syntax or changing their basic rules of composition and denotation. This ability appears to be unique among programming languages in a number of respects. Even the ability of AE-like languages to define function-valued functions does not seem exactly equivalent to the concept of modifiers. Thus for example, it is possible to define a Red operator similar to *FST* [first], let us call it *FST*1, which is function-valued, such that

$$\langle\langle FST1\ OP\rangle\ (X\ Y)\rangle \rightarrow (\langle OP\ X\rangle\ Y)$$

Thus *FST*1 when applied to an operator *OP*, produces another operator which has the same effect as the operator (*FST OP*). In the absence of modifiers <*FST*1 *OP*> denotes an abstact object, a function, with no operational representation. This creates difficulties of comprehension in more complex situations. The modifier *FST* is such that simply pairing it with *OP* yields the desired operator, (*FST OP*), whose behavior is explained by the behavior of *FST* = *(*APPLY REGRP*).

It appears that the only extensions needed for a language with a really good power of definability are provided by the ability to adjoin synonomous forms for existing expressions.

The power of definability of Red languages make them

interesting candidates, I believe, for more study and possibly for reference languages with which to define others. It should also prove interesting to study the possibility of introducing the meta composition notion into other languages.

Variable-free languages

The price for being a primary language is the absence of all but place-holder variables. Primary languages do not utilize states in which the values of variables can be recorded. And, as noted earlier, primary languages with place-holder variables seem to be impractical. Thus reduction languages and Red languages are variable-free. [Although operators which deal with variables may be defined.]

There is nothing perhaps more familiar and natural in programming than the use of names and variables--indeed in all forms of discourse. Therefore nothing might appear less natural or desirable than a programming language which uses no variables. There may be, in fact, situations in which variables are virtually necessary for the clear and concise expression of a process. But there are negative and cloudy aspects to the design of languages with variables which are not emphasized, since it has been felt that there is no alternative.

There are four important elements in ordinary programming:

(1) Obtaining the value of a variable.

(2) Altering the value of a variable.

(3) Selecting a variable from a set of variables.

Thus A[I] selects one variable from A, given I.

(4) Applying a function to one or more arguments.

Most programming languages cannot dispense with any one of these four. And yet there is an agonizingly close similarity between (3) and (4). One can indeed regard A as a function. Yet the two features cannot be merged into one nor one eliminated. I believe that this near-identity with not-well-understood differences is the source of much unnecessary complexity in programming. And (1) and (2) introduce further problems: e.g., scope and side-effects.

In the case of LISP-like or AE-like languages, variables serve a different role: that of place-holders in a function-defining expression. Difficulties arise in these languages with the treatment both of variables which name self-invoking functions and of uses of one variable in different roles. The solution of these difficulties involves building into the semantics of the language an elaborate machinery for evaluating such functions--machinery whose implications are difficult to understand fully. [I understand that a subtle problem in this area went undetected in the LISP interpreter for a long period.] Red languages do not require such built-in machinery.

In view of the many questions and difficulties posed by languages with variables, the question arises: what are the

benefits of variable-free languages? First, many of the complexities and unresolved questions in designing a language seem to go away. Second, in the brief experience I have had writing Red programs, a number of techniques have already arisen which make it a fairly easy and orderly task to write a variable-free program. Of course, variable-free programming presents an initial difficulty due to its novelty. But I believe that further experience may well devise programming techniques for variable-free Red-like languages which are simpler and more elegant than those which are now used in present languages with variables.

Efficiency

The question of the possibility of efficiently executing Red programs is left entirely unexplored in the present paper.

CLASSIFICATION OF PROGRAMMING LANGUAGES

General

This section proposes the definitions of several types of programming languages. Each is accompanied by an example, usually an existing language. The definitions characterize languages by the way in which their semantics are specified. Some minor technical details in the description of the example languages must be changed to conform to these definitions. However, the need for clear definitions of major classes of languages outweighs the need for these minor adjustments.

The purpose of the definitions below is to make some significant distinctions between different kinds of semantics for languages and to provide a background for the main subject of this paper: reduction languages. These notions will perhaps also prove useful in other studies of programming languages.

Primary languages

Definition of primary languages

A primary language L is a triple, (F,C,ρ), where:

F is the set of formulas of L,

C is the set of constants of L,

ρ is the transition function of L.

These elements have the following properties:

PR1) ρ maps F into F.

PR2) ρ is everywhere defined on F.

PR3) Every constant is a formula.

PR4) A formula F is a constant if and only if

$$\rho F = F$$

The semantic function, $\ddot{\rho}$, of a primary language

Associated with the transition function ρ of a primary language $L=(F,C,\rho)$ is a partial function $\ddot{\rho}$ called the semantic function of L, defined as follows:

> If F is a formula and there is an integer n such that $\rho^n F$ is a constant, then $\ddot{\rho}F = \rho^n F$; otherwise $\ddot{\rho}F$ is not defined.

We say that $\ddot{\rho}F$ is the meaning of F, if it is defined, and that F is then a meaningful formula. If ρ is not single-valued, $\ddot{\rho}$ may or may not be single-valued. If $\ddot{\rho}$ is not single-valued at F, we say that L is ambiguous for F.

Example 1: The primary language for a Turing machine

Let Z be a Turing machine as given in Davis [1], pages 3 through 7, with the following change: whenever the successor of an instantaneous description α is undefined in Davis [because there is no applicable quadruple of Z], we shall define its successor to be α and write $\alpha \to \alpha(Z)$. And we shall say that α is a terminal instantaneous description whenever $\alpha \to \alpha(Z)$ [instead of when $\alpha \to \beta(Z)$ is false for all β.

In Davis, $\alpha \rightarrow \alpha(Z)$ can also result from a quadruple $q_i S_j S_j q_i$, but we shall call α terminal in this case, even though in Davis α would yield a non-terminating computation].

Let FZ be the set of all instantaneous descriptions of Z. Let CZ be the set of all terminal instantaneous descriptions of Z. Let ρZ be the transition function from one instantaneous description α to its successor β as given by $\alpha \rightarrow \beta(Z)$ as modified in the above paragraph.

Then LZ = (FZ,CZ,ρZ) is a primary language, since:

1) $\rho Z : FZ \rightarrow FZ$
2) ρZF is defined for every F in FZ.
3) $CZ \subset FZ$
4) For all F in F: $F \in CZ \leftrightarrow \rho ZF = F$.

Example 2: The primary language for a computer

In this example and in subsequent ones we shall be less precise than in the first.

Let M be a computer. Let a state of M comprise the contents of all the registers and storage cells of M. Let a terminal state be one in which the program counter points at a cell containing a STOP instruction. Let ρM be the transition function which describes the change in state resulting from the execution of one machine instruction [with $\rho M S = S$ for a terminal state S]. Let SM be the set of states of M and SMt be the set of terminal states of M. Then LM = (SM,SMt,ρM) is a primary language.

Remarks

In the above example, if M has two program counters which can proceed asynchronously, then ρM can be multiple-valued in that the next state will depend on which of two instructions is executed next. However, if the program counters control separate programs A and B and if these share no cells or registers, then the semantic function $\ddot{\rho}M$ can be defined and single-valued since the terminal state when both programs end will not depend on the order in which instructions were taken from A and B.

We are not accustomed to regarding states of a machine as formulas of a language. They are much too complex. Machine programs perhaps come closer to our notion of a language. Yet to fully describe the meaning of a program the notion of states and their transitions must be invoked. The following definition of secondary languages deals with this dependence of meaning on "state languages".

Although, as noted in the introduction, the lambda notation of Church [9] and Curry's system of combinators [3] could be formulated as primary languages, they have not been so formulated sufficiently clearly to do more than note the possibility. Thus from here on we shall only consider their available formulations as secondary languages [see below].

Secondary languages

Definition of secondary languages

A secondary language L is a quadruple,

$$(P,(S,St,\rho),\alpha_i,\alpha_t)$$

where:

P is the set of programs of L.

(S,St,ρ) is the state language of L.

S is the set of states of L.

St is the set of terminal states of L.

ρ is the state transition function of L.

α_i is the initial-state-to-program function of L.

α_t is the terminal-state-to-program function of L.

These entities have the following properties:

1) (S,St,ρ) is a primary language.

2) α_i maps S into P. α_i may be partial or total on S.

3) α_t maps St into P. α_t may be partial or total on St.

When $\alpha_i S = P$ we say that S is an initial state for P. And when $\alpha_t S = P$ we say that S is a terminal state for P. If there is no initial state for P, we say that P is indeterminate. If P has an initial state S and $\ddot{\rho}S$ is undefined, we say that P is non-terminating.

Example 3: The secondary language for the programs of a computer

Let M be the computer of example (2). Let LM be the

primary language of that example. Thus LM = (SM,SMt,ρM). Let P be the set of relocatable programs for M. Let $\alpha_i S = P$ whenever S is a state containing P relocated and stored beginning at cell 1 and the program counter pointing at 1 [the other cells may contain anything]. If S is not such an initial state for any program, $\alpha_i S$ is undefined. Let α_t be everywhere undefined. Then L = (P,LM,α_i,α_t) is a secondary language.

Remarks

Notice what L describes and what it does not. It describes the set of relocatable programs of M: if P belongs to P, then P is a program, otherwise not. It describes what happens when a program is put in a certain place in the store and executed. It has nothing to say about what data the program will refer to in the store or where to find the "result" if the program terminates.

In fact, L cannot indicate the result unless the state containing it can be meaningfully mapped into a "program" by α_t. [The alternative to this requirement is more complicated: a universe of programs and a universe of data plus a number of other notions are then needed to construct a "language" and its semantics.]

Observe that the definition of secondary languages permits a much more complex relationship between a program and a corresponding initial state than that of the above

example. One might, for example, have a secondary language with P the set of Algol 60 programs and with a state language of some computer. The correspondence between a program and an initial state would then involve the translation of the Algol program into some machine program. Thus we would have $\alpha_i S=P$ when S contains a translation of P.

Some languages, e.g., Landin's Applicative Expressions, or AE's [2], have a unified notion of "program" which includes that of "data". In such languages the natural meaning of a "program"--the application of a function to some data--is the result of that application [which is also a "program"]. However, in all existing examples of this kind of language, none express the relation between a program and its result as the semantic function of a primary language. Thus, for example, no transition function is given which maps AE's into AE's and which when applied often enough produces the meaning of the original AE. "States" are employed which record not only various extended AE's [extended by the inclusion of the non-AE "ap"] but the values of variables as well. Nevertheless, these languages have a natural semantic function: the value of an expression.

Complete secondary languages describe the situation in which a semantic function exists for languages which are not primary.

Complete secondary languages

Definition of complete secondary languages

Let $L = (P,(S,St,\rho),\alpha_i,\alpha_t)$ be a secondary language. Let P be any program of L. Then L is complete if and only if:

If:

a) There is an initial state S for P, i.e., $\alpha_i S = P$,

and b) $\ddot{\rho}S$ is defined,

Then:

c) $\alpha_t\ddot{\rho}S$ is defined,

and d) For every initial state S' for P, both $\ddot{\rho}S'$ and $\alpha_t\ddot{\rho}S'$ are defined,

and e) $\alpha_t\ddot{\rho}S = \alpha_t\ddot{\rho}S'$.

The semantic function, ω, of a complete secondary language

Associated with every complete secondary language $L = (P,(S,St,\rho),\alpha_i,\alpha_t)$ is a semantic function, ω, defined as follows:

If P is a program of L for which there is at least one initial state S, then ωP is defined whenever $\ddot{\rho}S$ is defined:

$$\omega P = \alpha_t\ddot{\rho}S$$

Otherwise ωP is undefined.

The definitions of complete secondary language and of ω assure that ω is a single-valued partial function from P into P. Thus the meaning of a program P is found--if it has

one--by first finding an associated initial state S for P, then applying the state transition function repeatedly until a terminal state is reached, and finally extracting the result from this by α_t.

Example 4: The complete secondary language for applicative expressions

The terms and notation used in this example are from The mechanical evaluation of expressions by P. J. Landin [2].

Let P be the set of applicative expressions without free variables. Let SS be the set of states: (S,E,C,D) except that we admit the null list as a possible value of D. Let St be the set of states of the form: t(X:(),(),(),()) or ((),(),(),()). Let ρ be the function transform as defined by Landin with the following extension for the case D = ():

Definition of ρ:

When transform(S,E,C,D) is defined:

ρ(S,E,C,D) = transform(S,E,C,D)

Otherwise, if S is not null:

ρ(S,E,(),()) = (hS:(),(),(),())

and for all other arguments for which transform is not defined:

ρ(S,E,C,D) = ((),(),(),())

Thus ρ is defined for every state and is constant on exactly St. Hence (SS,St,ρ) is a primary language.

Let α_i and α_t be defined:

$$\alpha_i((),(),X:(),()) = X$$

and

$$\alpha_t(X:(),(),(),()) = X$$

α_i and α_t are undefined for all other arguments.

Then $L = (P,(SS,St,\rho),\alpha_i,\alpha_t)$ is a complete secondary language and its semantic function $\omega = \alpha_t\ddot{\rho}(\alpha_i)^{-1}$ maps an AE X into val()X when the latter is defined.

Remarks

If we confine our attention to "complete" languages [i.e., those which associate with some formulas or programs certain other formulas or programs which are the "meaning" of the former], we can crudely distinguish between "state languages" and "programming languages". The distinction is one of viewpoint only. Thus a state language is not a "programming language" as long as the objects to which we wish to attach meaning are not the states themselves. We are usually concerned with the meaning of programs imbedded in initial states. On the other hand, a complete secondary language is a "programming language" because its semantic function gives meaning to those objects--programs--which we want to have meaning. Finally, a state language becomes a "programming language" as soon as we become interested in assigning the meaning to states which $\ddot{\rho}$ gives them.

Reduction languages

General

In preceding paragraphs we have attempted to structurally characterize the semantics of existing programming languages by our description of secondary and complete secondary languages. They are fundamentally complex in that they comprise a set of programs, a state language, and two mappings of states into programs. The meaning of a program--if it has one--is obtained by first finding an associated initial state, then applying the semantic function of the state language to obtain a terminal state, and finally extracting from the latter the program which is the meaning of the original.

In the present section we begin the discussion of programming languages with an essentially simpler structure.

Reduction languages are first of all primary languages whose formulas are strings. Their essential additional properties are:

For any subformula G of a formula F,

1) If F is meaningful, then G is.

2) If G is replaced in F by its meaning, the meaning of F is unchanged.

3) If G is replaced by ρG, the meaning of F will then be reached in one less application of ρ to F, provided G is not a constant.

These properties make the procedure for finding the meaning of a formula a very simple one. By repeatedly applying ρ, one procedes from one formula to another until the meaning of the original, if any, is reached. To apply ρ to a formula, that is, to "reduce" it, one need only know how to reduce any one of its simplest non-constant subformulas. Thus reduction languages assume an equivalent of the Church-Rosser theorem for lambda-conversion: the result does not depend on the choice of the simplest non-constant subformula to be reduced at each step.

Examples of simple reduction languages are common. Thus if F is some set of arithmetic expressions involving only constants, and if C is the set of their possible values, and if ρ is some rule for simplifying them, then $L=(F,C,\rho)$ is likely to be a reduction language. However, these languages have very little expressive power. What is not generally recognized is that there are reduction languages which are complete programming languages capable of describing any computable function. It is the main goal of this paper to exhibit several reduction languages whose syntax and semantics are unusually simple but which nevertheless possess considerable expressive power.

One striking feature of reduction languages is the absence of variables. There has been only one well-known effort to study systems without variables, that of Curry and his associates [3]. Curry shows that certain variable-free systems are equivalent to that of lambda-conversion.

Although his variable-free system of combinators is very pleasing in many respects, the difficulty of interpreting its statements has apparently discouraged further efforts to develop other variable-free systems.

It is interesting to note in this connection that a programming system has been defined which reduces "objects" of Curry's combinatory calculus to their normal form. One need only study G. W. Petznick Jr.'s thesis, Combinatory Programming [4] to become convinced (a) that the meaning of an "object" is indeed difficult to comprehend and (b) that the complexity of the process for finding the normal form of an object greatly exceeds that of processes for reducing expressions with variables [e.g., AE's]. The latter comparison is probably very unfair to Curry's system. Although it appears that Petznick has treated it as a secondary language [auxilliary data appears to be carried over more than one transition], one suspects that a treatment similar to Landin's treatment of AE's could comprise a semantics for objects at least as simple as that for AE's.

A variable-free programming system has much simpler requirements than a system of logic. [It is possible, however, that some of the devices used in Red languages may be useful in constructing simpler logical systems.] It is hoped that the subsequent examples of reduction programming languages will convince the reader that interpreting their formulas is a simple and straightforward matter.

Variables and names in a programming language are both a blessing and a curse. Their conveniences are well-known but their peculiarities are accepted because they are thought to be essential. One of the crucial sources of inflexibility of a programming language is the naming system which comes with it. This given system hinders the introduction of new naming conventions. In order to be complete the naming system of a language carries with it a host of rules and conventions which are a major part of the complexity of the language. Furthermore, one of the major categories of work done by a program is that of fetching the referents of names. This work is not properly reflected by the use of names in conventional languages.

Languages with variables have two primary functions involved in their semantics. One is the function var which maps variable/subscript pairs (v,i) into the value v(i). The other, app, maps function/object pairs (f,x) into the value f(x). The context-dependency of var and the subtle differences between these two nearly identical functions are important sources of complexity and difficulty in using languages with variables.

The irony of a conventional programming language with elaborate naming conventions is that its semantics must ultimately be described using a language without names--its state language. [If its state language is not primary, then it must rely, in turn, on yet another language for its definition, etc.] Just as a computer deals with addresses in

a program by making transformations of its state which involve putting an address into an address register, so a state language deals with names of a programming language as ordinary objects at the state level.

One great advantage of a variable-free programming language is its ability to define various rules for replacing names by their referents and for assigning referents, without conflicting with rules built into the language itself. Such languages might be useful as reference languages in which to define and study languages with a variety of naming systems.

Preliminary definitions

In the following we shall be implicitly referring to a primary language L = (F,C,ρ) whose formulas are strings of characters from some alphabet. All variables [e.g., F, G, H etc.] will refer to formulas of L unless otherwise stated.

Substrings

We say that a string *S* is a substring of a string *T* if there are strings (possibly empty) *S*1 and *S*2 such that *T* is the concatenation of *S*1, *S*, and *S*2 [written: *T* = *S*1 *S* *S*2].

Atoms

L may have certain formulas which are designated atoms by its syntax.

Subformulas

We shall say that G is a subformula of F, and denote this by: $G \leq F$, whenever:

1) G is a formula; and,

2) G is a substring of F; and,

3) If $F = S\ G\ T$ for strings S and T and if A is an atom and a substring of F, then A is a substring of S or of G or of T.

We shall write $G<F$ to indicate that G is a proper subformula of F.

When we say that G is a subformula of F, the identity of G involves S and T as well as G, where $F = S\ G\ T$. Thus two subformulas G and H of F are distinct even when $G=H$ provided that $F = S\ G\ T$ and $F = S'\ H\ T'$ and $S \neq S'$.

Notation for replacement: $F[G/H]$

By the notation $F[G/H]$ we shall mean the string or formula obtained by replacing the subformula G by H in F. Thus if $F = S\ G\ T$, then $F[G/H] = S\ H\ T$. Again, it is important to note that $F[G/H]$ denotes the replacement of one particular instance of G within F, as noted above.

Definition of reduction languages

A reduction language is a primary language, $L=(F,C,\rho)$, such that, for every formula F and every subformula G of F:

RE1) F is a string.

RE2) $F[G/\rho G]$ is a formula.

RE3) If G is not a constant and if either $\rho^n F$ or $\rho^{n-1}F[G/\rho G]$ is a constant for some $n \geq 1$, then:

$$\rho^n F = \rho^{n-1}[G/\rho G]$$

Example 5: A reduction language for simple expressions

Let F be the set of expressions formed as follows:

1) An integer [a string of digits] is an atom and an expression.

2) If e and f are expressions, then the following are expressions:

(e+f)

and

(e×f)

Let C be the set of integers. Let ρ be defined for expressions as follows:

Let e be a non-integer expression and f be an expression. Let i and j be integers; let $\circ$ be either + or ×. Then:

1) $\rho i = i$

2) $\rho(i+j)$ = sum of i and j

$\rho(i\times j)$ = product of i and j

3) $\rho(i \circ e) = (i \circ \rho e)$

4) $\rho(e \circ f) = (\rho e \circ f)$

Then $L=(F,C,\rho)$ is a reduction language.

Theorems about reduction languages

The following theorems refer to a reduction language $L=(F,C,\rho)$.

Theorem Th/RE1

If: 1 $F \in C$

2 $G \leq F$

Then: $G \in C$

proof

assertion // justification [=numbers of previous assertions, definitions, cases, etc.]

3 $\sim(G \in C)$ // case 1: Assume the theorem false

4 $\rho G \neq G$ // 3, PR4 [the 4th property of primary languages]

5 $F \neq F[G/\rho G]$ // 4, def of $F[G/H]$

6 $\rho F = F[G/\rho G]$ //1,2, 3, RE3

7 $\rho F \neq F$ // 6, 5

8 case 1 cannot hold // 7 contradicts 1

9 $G \in C$ // 8, 3

Theorem Th/RE2

If: 1 $\rho^n F \in C$ for some $n \geq 0$

2 $G \leq F$

Then: $\rho^n G \in C$

proof By induction on n.

assertion // justification

3 n=0 // first part of induction

4 $F \in C$ // 1, 3

5 $G \in C$ // 4, 2, Th/RE1

6 theorem holds for n=0 // 5, 3

7 theorem holds for n<k // induction hypothesis

8 $\rho^k F \in C$ // 1 with n=k

9 $G \in C$ // case 1

10 $\rho^k G \in C$ // 9, PR4

11 $\sim(G \in C)$ // case 2

12 $\rho^k F = \rho^{k-1} F[G/\rho G]$ // 2, 11, 8, RE3

13 $\rho^{k-1} F[G/\rho G] \in C$ // 12, 8

14 $\rho G \leq F[G/\rho G]$ // def of $F[G/H]$

15 $\rho^{k-1} \rho G \in C$ // 7, 13, 14

16 theorem holds for n=k // 15, 10

17 Q.E.D. // 6, 16

Theorem Th/RE3

If: 1 $G \leq F$

2 $\ddot{\rho}F$ is defined

Then: $\ddot{\rho}F = \ddot{\rho}F[G/\ddot{\rho}G]$

proof

By use of Th/RE2, and repeated use of property RE3.

Remarks

Theorem Th/RE2 confirms our earlier statement about reduction languages that if F is meaningful and if G is a subformula of F, then G is meaningful. Theorem Th/RE3 confirms the fact that replacing G by its meaning does not change the meaning of a formula containing it.

The principal consequence of the properties of reduction languages is the following: To reduce any formula F to its meaning $\ddot{\rho}F$, one can choose any <u>minimal non-constant subformula</u> G of F [G has no proper non-constant subformula] and replace it by ρG. Repeating this process will produce $\ddot{\rho}F$ if it exists. Although there may be many paths from F to $\ddot{\rho}F$, depending on the choice of a minimal non-constant at each step, all arrive at the same constant: $\ddot{\rho}F$. For example, consider the following two reductions of an expression from example 5:

$$\rho((1\times2)+(3\times4)) = (\rho(1\times2)+(3\times4))$$
$$= (2+(3\times4))$$
$$\rho(2+(3\times4)) = (2+\rho(3\times4)) = (2+12)$$
$$\rho(2+12) = 14$$

But instead of using the definition of ρ as above to reduce the expression, we could use the property RE3 and obtain the following:

$$\rho^3((1\times2)+(3\times4)) = \rho^2((1\times2)+\rho(3\times4))$$
$$= \rho^2((1\times2)+12) = \rho(\rho(1\times2)+12) = \rho(2+12)$$
$$= 14$$

We have not shown that L of example 5 is actually a reduction language. It is intuitively clear that replacing any subexpression E by ρE in an expression F will not change the value of F. But to prove that L or any other language satisfies the definition of a reduction language can be a lengthy chore. Therefore we introduce a class of languages called parenthetic languages whose axioms are easier to satisfy and show that it is a subclass of reduction

languages. It will then suffice to show that L of example 5 is parenthetic.

Parenthetic languages

Preliminary definition: factors of a formula

If L=(F,C,ρ) is a primary language with string formulas and F is a formula of L, then G is a factor of F whenever:

1) $G<F$

2) $\sim(G\in C)$

3) For all G':

$G\leq G'<F \rightarrow G=G'$

Thus a factor of F is a non-constant maximal proper subformula of F.

Definition of parenthetic languages

A parenthetic language is a primary language L=(F,C,ρ) such that, for every formula F:

PA1) F is a string.

PA2) If $G\leq F$, then $F[G/\rho G]$ is a formula.

PA3) If $G\leq F$ and $H\leq F$, then one of the following holds:

a) $G<H$

b) $H\leq G$

c) For all E, if $F[G/E]$ is a formula, then $H\leq F[G/E]$

PA4) If F has a factor, then there is a factor H of F such that:

$\rho F = F[H/\rho H]$

Theorems about parenthetic languages

The following theorems refer to a parenthetic language L=(F,C,ρ).

Theorem Th/PA1

If 1 $F \in C$

2 $G \leq F$

Then: $G \in C$

proof

Assertion // Justification

3 $\sim(G \in C)$ // case 1

4 $G \neq F$ // 1, 3

5 $G < F$ // 2, 4

6 F has a factor // 3, 5, def of factor

7 There is a factor H of F such that: $\rho F = F[H/\rho H]$ // 6, PA4

8 $\sim(H \in C)$ // 7, def of factor

9 $\rho H \neq H$ // 8, PR4

10 $F[H/\rho H] \neq F$ // 9, def of $F[G/H]$

11 $\rho F \neq F$ // 10, 7

12 case 1 cannot hold // 11 contradicts 1

13 $G \in C$ // 3, 12

Theorem Th/PA2

If: 1 $F \in F$

2 $G \leq H \leq F$

3 $F[G/E] \in F$

4 $H[G/E] \in F$

Then: $H[G/E] \leq F[G/E]$

proof

Assertion // Justification

5 $F = S\ H\ S'$ for some strings S and S' // 2, def of subformula

6 $H = T\ G\ T'$ for some strings T and T' // 2, def of subformula

7 $F = S\ T\ G\ T'\ S'$ // 5, 6

8 $F[G/E] = S\ T\ E\ T'\ S'$ // 7, def of $F[G/H]$

9 $H[G/E] = T\ E\ T'$ // 6, def of $F[G/H]$

10 $H[G/E] \leq F[G/E]$ // 8, 9, 3, 4, def of subformula

Theorem Th/PA3

If: 1 $F \in$ F

2 $G \leq H \leq F$

3 $F[G/\rho G] \in$ C

Then: $H[G/\rho G] \in$ C

proof

Assertion // Justification

4 $H \in$ F // 1, 2, def of subformula

5 $H[G/\rho G] \in$ F // 2, 4, PA2

6 $H[G/\rho G] \leq F[G/\rho G]$ // 2, 5, Th/PA2

7 $H[G/\rho G] \in$ C // 3, 6, Th/PA1

Theorem Th/PA4

If: 1 $G \leq H \leq F$

Then: $F[H/H[G/E]] = F[G/E]$

proof

Assertion // Justification

2 $F = S\ H\ S'$ for strings S and S' // 1

3 $H = T\ G\ T'$ for strings T and T' // 1

4 $H[G/E] = T\ E\ T'$ // 3

5 $F = S\ T\ G\ T'\ S'$ // 2, 3

6 $F[H/H[G/E]] = S\ T\ E\ T'\ S'$ // 2, 4

7 $F[G/E] = S\ T\ E\ T'\ S'$ // 5

8 $F[H/H[G/E]] = F[G/E]$ // 6, 7

Theorem Th/PA5

If: 1 $G_i = F_i[F_{i+1}/G_{i+1}]$ for $1 \le i \le n$

2 $F_{i+1} \le F_i$ for $1 \le i \le n$

Then: $G_1 = F_1[F_{n+1}/G_{n+1}]$

proof By repeated application of Th/PA4.

Theorem Th/PA6

If: 1 $F \in F$

2 $G \le F$

3 $\sim(G \in C)$

4 $\rho F \in C$ or $F[G/\rho G] \in C$

Then: $\rho F = F[G/\rho G]$

proof

Assertion // Justification

Case 1

5 F has no factor // case 1

6 $F \in C$ or F is a minimal non-constant // 5, def of factor

7 F is a minimal non-constant // 6,2,3, Th/PA1

8 $G = F$ // 7, 2, 3,

9 $\rho F = F[G/\rho G]$ // 8

Case 2

10 F has a factor // case 2

11 There is an H such that:

a) $\rho F = F[H/\rho H]$; and,

b) H is a factor of F

// 10, PA4

12 a) $H < G$

or b) $G \leq H$

or c1) $G \leq F[H/\rho H]$ and c2) $H \leq F[G/\rho G]$

// 2, 11b, PA3

13 12a does not hold // 2, 3, 11b, def of factor

Case 2.1

14 12c holds // case 2.1

15 $\sim(\rho F \in C)$ // Th/PA1, 14, 12c1, 11a, 3

16 $\sim(F[G/\rho G] \in C)$ // Th/PA1, 14, 12c2, 3, def of factor

17 12c does not hold // 15 and 16 contradict 4

18 $G \leq H$ // 12, 13, 17

Case 2.2

19 $\rho F \in C$ // case 2.2

20 $\rho H \leq \rho F$ // 11a

21 $\rho H \in C$ // Th/PA1, 19, 20

Case 2.3

22 $F[G/\rho G] \in C$ // case 2.3

23 $G \leq H < F$ // 18, 11b

24 $H[G/\rho G] \in C$ // Th/PA3, 22, 23

25 $\rho H \in C$ or $H[G/\rho G] \in C$ // 4, 19, 21, 22, 24

26 G and H satisfy 1, 2, 3, and 4 with H replacing F;
and $H<F$ and $\rho F=F[H/\rho H]$ // 11b, 18, 3, 25, 11

27 There is a sequence $F_0,F_1,\ldots,F_{n+1}$ such that:

a) $F=F_0$,

and b) $F_{i+1}<F_i$, $0\leq i<n$,

and c) $G\leq F_i$, $0\leq i\leq n$,

and d) $\rho F_i=F_i[F_{i+1}/\rho F_{i+1}]$, $0\leq i<n$,

and e) $\rho F_i\in C$ or $F_i[G/\rho G]\in C$, $0\leq i\leq n$

// 26 and repeated application of the argument of case 2 beginning at assertion 10, with $F_i=F$ and $F_{i+1}=H$.

28 For some n, either case 1 applies to G and F_n or $G=F_n$. // 27

29 $\rho F_n = F_n[G/\rho G]$ // 28, 8, 9

30 $\rho F = F[G/\rho G]$ // 29, 27, Th/PA5

31 Q.E.D. // 9, 30

Theorem Th/PA7

If: 1 $F \in F$

2 $G \leq F$

3 $\sim(G \in C)$

4 $\rho^n F \in C$ or $\rho^{n-1}F[G/\rho G] \in C$ for some $n\geq 1$

Then: 5 $\rho^n F = \rho^{n-1}F[G/\rho G]$

proof By induction on n.

n=1 This case is precisely Th/PA6.

We assume that the theorem is true for $n<k$, and prove it for $n=k$, $k>1$.

Assertion // Justification

IH Conditions 1 through 4 imply 5 for $n<k$.

// induction hypothesis

case 1

6 F has no factor // case 1

7 $G = F$ // 6, 2, 3, def of factor

8 $\rho^k F = \rho^{k-1} F[G/\rho G]$ // 7, def of $F[G/H]$

case 2

9 F has a factor // case 2

10 There is an H such that:

a) $\rho F = F[H/\rho H]$ and,

b) $H < F$ and,

c) $\sim(H \in C)$

// 9, PA4

11 One of the following holds:

a) $H < G$

b) $G \leq H$

c1) $G \leq F[H/\rho H]$ and c2) $H \leq F[G/\rho G]$

// PA3, 2, 10b, def of factor

12 $\sim(H < G)$ // 2, 10b, 10c, 3, def of factor

case 2.1

13 11c holds:

a) $G \leq F[H/\rho H]$ and,

b) $H \leq F[G/\rho G]$

// case 2.1

case 2.1.1

14 The first part of 4 holds: $\rho^k F \in C$ // case 2.1.1

15 $\rho^{k-1} F[H/\rho H] = \rho^k F$ // 10a

16 $\rho^{k-1}F[H/\rho H] = \rho^{k-2}F[H/\rho H][G/\rho G]$

// IH: 1→PA2, 1, 10b; 2→13a; 3→3; 4→14, 15

17 $F[H/\rho H][G/\rho G] = F[G/\rho G][H/\rho H]$ // 13, def of $F[G/H]$

18 $\rho^{k-2}F[G/\rho G][H/\rho H] \in C$ // 14, 15, 16, 17

19 $\rho^{k-1}F[G/\rho G] = \rho^{k-2}F[G/\rho G][H/\rho H]$

// IH: 1→PA2, 1, 2; 2→13b; 3→10c; 4→18.

20 $\rho^{k}F = \rho^{k-1}F[G/\rho G]$ // 15, 16, 17, 19

Q.E.D., case 2.1.1

case 2.1.2

21 $\rho^{k-1}F[G/\rho G] \in C$ // case 2.1.2

22 $\rho^{k-1}F[G/\rho G] = \rho^{k-2}F[G/\rho G][H/\rho H]$

// IH: 1→PA2, etc.; 2→13b; 3→10c; 4→21

23 $\rho^{k-1}F[H/\rho H] = \rho^{k-2}F[H/\rho H][G/\rho G]$

// IH: 1→PA2, etc.; 2→13a; 3→3; 4→21, 22

24 $\rho^{k}F = \rho^{k-1}F[H/\rho H]$ // 10a

25 $\rho^{k}F = \rho^{k-1}F[G/\rho G]$ // 22, 23, 17, 24

Q.E.D. case 2.1.2 and 2.1

case 2.2

26 $G \leq H$ // case 2.2

case 2.2.1

27 $G = H$ // case 2.2.1

case 2.2.2

28 $G < H$ // case 2.2.2

29 H has a factor // 28, 3, def of factor

30 There is an L such that:

a) $\rho H = H[L/\rho L]$

b) L is a factor of H

// 29, PA3

31 Either

a) $G \leq L$ or,

b) $G \leq H[L/\rho L]$ and $L \leq H[G/\rho G]$

// see 6 through 12

32 There is a sequence $F_0, F_1, \ldots, F_{m+1}$ such that:

a) $F = F_0$

b) $F_{i+1} < F_i$

c) $\rho F_i = F_i[F_{i+1}/\rho F_{i+1}]$

d) $G < F_i$

e) F_{i+1} is a factor of F_i

hold for $0 \leq i \leq m$. // repeated argument of case 2.2 with $F_i = H$ and $F_{i+1} = L$, $1 \leq i \leq m$.

33 The above sequence ends with either:

a) $G = F_m$ or,

b) $G \leq F_m[F_{m+1}/\rho F_{m+1}]$ and $F_{m+1} \leq F_m[G/\rho G]$

// 31, 26, 27

<u>case 2.2.2.1</u>

34 $G = F_m$ // case 2.2.2.1 [33a]

35 $\rho F_0 = F_0[F_m/\rho F_m]$ // Th/PA5, 32c, 32b

36 $\rho F = F[G/\rho G]$ // 32a, 35, 34

37 $\rho^k F = \rho^{k-1} F[G/\rho G]$ // 36

Q.E.D. case 2.2.2.1

<u>case 2.2.2.2</u>

38 a) $G \leq F_m[F_{m+1}/\rho F_{m+1}]$ and,

b) $F_{m+1} \leq F_m[G/\rho G]$

// case 2.2.2.2 [33b]

39 $\rho F = F[F_{m+1}/\rho F_{m+1}]$ // Th/PA5, 32c, 32b

40 $F_m[F_{m+1}/\rho F_{m+1}] \leq F[F_{m+1}/\rho F_{m+1}]$ // Th/PA2, 32b

41 $G \leq F[F_{m+1}/\rho F_{m+1}]$ // 38a, 40

42 $F_m[G/\rho G] \leq F[G/\rho G]$ // Th/PA2, 32b, 32d

43 $F_{m+1} \leq F[G/\rho G]$ // 38b, 42

44 F, G, and F_{m+1} replacing H satisfy all the conditions of case 2.1; namely, they satisfy assertions 1, 2, 3, 4, 9, 10, and 13.

// 1→1; 2→2; 3→3; 4→4; 9→9; 10a→39; 10b→32a, 32b; 10c→32e, def of factor; 13a→41; 13b→43.

45 $\rho^k F = \rho^{k-1} F[G/\rho G]$ // 44, case 2.1

Q.E.D. case 2.2.2.2

Q.E.D.

Remarks

Theorem PA7 proves that every parenthetic language is a reduction language. Therefore, to show that L of example 5 is a reduction language, it is only necessary to show that it is parenthetic. It is not difficult to show that L satisfies PA2. PA3 requires of two subformulas of F that one contain the other or that they not overlap. If one observes that integers are atoms of L, it is clear that PA3 holds for L. That L satisfies PA4 follows immediately from the definition of ρ for L. Thus L is parenthetic.

Since virtually all reduction languages one would wish to define have non-overlapping subformulas, and since recursive definition of ρ usually conforms to PA4, most reduction languages will be parenthetic. We

have introduced the two definitions because the properties of reduction languages are the semantically important ones whereas those of parenthetic languages conform to a convenient scheme for defining a language. Theorem PA7 provides the bridge, therefore, between this convenient scheme of definition and the semantic properties of reduction languages.

RED LANGUAGES

Introduction

In this section we shall describe the common features of several particular reduction languages, called Red languages. Each of these languages is an elaboration or variation of a predecessor. The first, Red/1, is of course the most primitive but is intended to be able to deal with any kind of parenthesized or string-like objects, including those with characters from its own syntax.

The syntax of each Red language is a slight elaboration of that of Red/1. The semantics of all are the same except for the set of primitive operators and the way in which the two classes of operators are distinguished.

Syntax of Red languages

General

In this section we describe the common features of the syntax of an arbitrary Red language $L=(F,C,\rho)$.

Syntax of L: formulas

1 A formula of L is an item.

2 An item is either an atom or a set or an application or □, the "undefined" item, quad.

3 A set is an ordered set of one or more items. A set may have a symbol as a prefix.

4 An application is an ordered pair of items.

5 An atom is a symbol or a string.

[a sequence of characters] [a quoted sequence of characters and system-characters]

Syntax of L: constants

The constants of L are determined as follows:

1 Quad is a constant.

2 Every atom is a constant.

3 A set is a constant whenever all its members are constants.

4 No application is a constant.

Syntax of L: notation

Without committing ourselves to any particular syntax for L, we shall use that of Red/1 for examples. Thus we shall denote a set by:

$$(\underline{X}1...\underline{X}n)$$

where $\underline{X}1...\underline{X}n$ denote the items which comprise the members of the set.

We use underlining to distinguish variables denoting items from non-underlined symbols which are atoms. Of course the expressions we use which contain such variables are not themselves items but merely denote a class of items with a given form. Thus the expression

$$(\underline{X})$$

denotes the class of all solitary sets. If *A* is an atom, then

$$(A)$$

is a particular solitary set.

We shall write an application as

$$<\underline{X}\ \underline{Y}>$$

where $\underline{X}$ and $\underline{Y}$ denote items.

Syntax of L: examples of constant items

Let *A*, *B*, *C* be atoms. Then the following are constants:

Atoms

A

B

Sets

((A))

(A B C)

(A ((B) C))

Thus all constants are atoms, or sets whose ultimate members are atoms.

Syntax of L: examples of non-constant items

Let *A*, *B*, *C*, *D*, *E* be atoms. Then the following are non-constants:

Sets

(A <B C> D)

(A <(B C) (C D)>)

Applications

<A B>

<<A B> (C D E)>

<(A B C) <D E>>

Semantics of Red languages

General

In the following paragraphs we shall describe precisely how to find the meaning of an item. Basically this involves replacing each application by the item it denotes, working

from the inside out. [Every constant item denotes itself.]

The first member of an application is its operator, the second its operand, and it denotes the result of applying the operator to the operand. If the operator is a set, this indicates the composition of its member operators which, if all are regular, is the familiar composition of functions used in mathematics. If a member belongs to the other class of operators, called meta operators, meta composition is invoked. This form of composition is the key to the expressive power of Red languages.

The restricted transition function for Red languages

Our object is to describe the transition function for Red languages. As noted in the section on reduction languages, the total transition function is effectively determined by one which is restricted to minimal non-constant formulas. For Red languages the latter are applications with constant members. Thus it is sufficient that we define for these applications a restricted transition function which we shall call Δ.

To describe Δ we shall employ two auxilliary functions whose definitions may vary from one Red language, L, to another. One of these is ϵ, the primitive execution function, which gives the result of applying any primitive operator of L to an operand. The other is ψ, the regular part function, which distinguishes between regular and meta operators [=items] of L. For any item $\underline{X}$, $\psi\underline{X}$ is always a regular item, called the regular part of $\underline{X}$. If $\underline{X}$ is regular, $\psi\underline{X} = \underline{X}$. If $\underline{X}$ is meta, $\psi\underline{X}$ is an associated regular item

[which of course differs from $\underline{X}$]. In Red/1 for example, a meta item is a set with the prefix * ; its regular part is the set without the prefix. E.g., $\psi*(A\ B\ C) = (A\ B\ C)$.

Notation

In describing functions we need to distinguish between different kinds of items. We shall use certain variables and expressions for this purpose as follows:

Variable	Range of variable
$\underline{X}\ \underline{Y}\ \underline{Z}\ \underline{X}1\ \underline{X}n$ etc.	Items
$\underline{R}$	Regular items: $\psi\underline{R}=\underline{R}$
$\underline{M}$	Meta items: $\psi\underline{M}\neq\underline{M}$
$\underline{A}\ \underline{B}$	Atoms

Expression	Denotation
$(\underline{X}1...\underline{X}n)$	An unprefixed, hence regular set with one or more members, each member [e.g., $\underline{X}1$] is an item. By regular we mean that $\psi(\underline{X}1...\underline{X}n) = (\underline{X}1...\underline{X}n)$.
$*(\underline{X}1...\underline{X}n)$	A meta set whose regular part is $(\underline{X}1...\underline{X}n)$.
$(\underline{X})$	An unprefixed set with one member, $\underline{X}$.
$(\underline{A})$	An unprefixed set with one member which is an atom.
$<\underline{X}\ \underline{Y}>$	The application of item $\underline{X}$ to item $\underline{Y}$.

Description of functions

Functions on items are described by cases with one case per line. Thus

$\alpha(\underline{X}) = \underline{X}$

$\alpha\underline{A} = (\underline{A}\ B)$

$\alpha\underline{X} = \underline{X}$

is a three-case description of a function α whose value $\alpha\underline{Y}$ is $\underline{X}$ when $\underline{Y}$ is a solitary unprefixed set $(\underline{X})$ and is the set $(\underline{Y}\ B)$ when $\underline{Y}$ is an atom [since $\underline{A}$ ranges over atoms only]. For all other arguments [e.g., prefixed solitary sets; sets of two or more members], α is the identity function.

Note that α and its arguments are juxtaposed and that the parentheses in the first case are part of the argument. Note also that in a description by cases, each line or case takes precedence over its successors: the first case having an argument of a given form determines the value of the function for arguments of that form. Thus $\alpha(A) = A$ is obtained from the first case of the description of α, even though the argument (A) conforms to that of the third case as well.

Definition of the restricted transition function, Δ

In the following description of Δ by cases, we have inserted titles for later reference without altering the nature of the description. The symbol $\emptyset$ is an atom used to represent the empty set.

Regular composition RC [$\psi\underline{R} = \underline{R}$]

$\Delta\langle(\underline{R})\ \underline{Y}\rangle = \langle\underline{R}\ \langle\phi\ \underline{Y}\rangle\rangle$

$\Delta\langle(\underline{R}\ \underline{X}2\ldots\underline{X}n)\ \underline{Y}\rangle = \langle\underline{R}\ \langle(\underline{X}2\ldots\underline{X}n)\ \underline{Y}\rangle\rangle$

Meta composition MC [$\psi\underline{M} \neq \underline{M}$]

$\Delta\langle(\underline{M})\ \underline{Y}\rangle = \langle\psi\underline{M}\ ((\underline{M})\ \underline{Y})\rangle$

$\Delta\langle(\underline{M}\ \underline{X}2\ldots\underline{X}n)\ \underline{Y}\rangle = \langle\psi\underline{M}\ ((\underline{M}\ \underline{X}2\ldots\underline{X}n)\ \underline{Y})\rangle$

Meta application MA [$\psi\underline{M} \neq \underline{M}$]

$\Delta\langle\underline{M}\ \underline{Y}\rangle = \langle\psi\underline{M}\ (\underline{M}\ \underline{Y})\rangle$

Primitive application PA

$\Delta\langle\underline{X}\ \underline{Y}\rangle = \epsilon\langle\underline{X}\ \underline{Y}\rangle$

Since ϕ is the primitive identity operator of all Red languages [i.e., $\epsilon\langle\phi\ \underline{Y}\rangle = \underline{Y}$], the first case of regular composition can be abbreviated to:

$\Delta\langle(\underline{R})\ \underline{Y}) = \langle\underline{R}\ \underline{Y}\rangle$

And we shall do so in the following.

Examples

In the following examples we shall write

$$\underline{X} \text{ -RC}\rightarrow \underline{Y}$$

to indicate that $\underline{Y}$ is obtained from $\underline{X}$ by a single application of regular composition, RC, to some innermost application within $\underline{X}$. We shall use similar notation for the other rules of Δ.

Example 1

Here we illustrate the use of regular composition. Let integers be atoms. Let + be a primitive operator such that, e.g.,

$$\epsilon\langle+\ (2\ 3)\rangle = 5$$

Let $\surd$ be a primitive operator such that, e.g.,

$$\epsilon\langle\surd\ 9\rangle = 3$$

Then:

$\langle(\surd\ +)\ (13\ 3)\rangle$

$\quad -RC\rightarrow\ \langle\surd\ \langle(+)\ (13\ 3)\rangle\rangle$

$\quad -RC\rightarrow\ \langle\surd\ \langle+\ (13\ 3)\rangle\rangle$ [abbreviated rule]

$\quad -PA\rightarrow\ \langle\surd\ 16\rangle$

$\quad -\grave{P}A\rightarrow\ 4$

Example 2

Here we exhibit a meta operator *ADJOIN* such that

$$\rho\langle(ADJOIN\ \underline{X})\ \underline{Y}\rangle = (\underline{X}\ \underline{Y})$$

Notice that although *ADJOIN* is a meta operator, the set $(ADJOIN\ \underline{X})$ is a composite regular operator. Let *ROT* and *CAR* be primitive operators such that

$$\epsilon\langle ROT\ ((\underline{X}\ \underline{Y})\ \underline{Z})\rangle = ((\underline{Y}\ \underline{Z})\ \underline{X})$$

and

$$\epsilon\langle CAR\ (\underline{X}\ \underline{Y})\rangle = \underline{X}$$

Let $*(\underline{X}1\ldots\underline{X}n)$ denote a meta set whose regular part is $(\underline{X}1\ldots\underline{X}n)$. I.e., $\psi*(\underline{X}1\ldots\underline{X}n) = (\underline{X}1\ldots\underline{X}n)$.

Then $ADJOIN = *(CAR\ ROT)$

For, if *A* and *B* are atoms, we have the following sequence of reductions:

<(*ADJOIN A*) *B*> =

<(*(CAR ROT) A) B*>

-MC→ <(*CAR ROT*) ((*(CAR ROT) A) B*)>

-RC→ <*CAR* <(*ROT*) ((*(CAR ROT) A) B*)>>

-RC→ <*CAR* <*ROT* ((*(CAR ROT) A) B*)>>

-PA→ <*CAR* ((*A B*) *(CAR ROT)*)>

-PA→ (*A B*)

ADJOIN typifies the class of meta operators called modifiers. In general, a set whose first member is a modifier forms a composite regular operator which depends on the other members in some manner other than normal composition. Often a modifier is paired with another operator whose behavior it alters.

Example 3

We exhibit a meta operator which removes leading zeroes from a set whose members are digits. Since our aim is to illustrate the use of meta application, we shall assume a regular primitive operator Z which does most of the work.

Let Z be a primitive operator such that:

ϵ<Z (X (Y1...Yn))>

= (Y1...Yn) when Y1 ≠ 0 or n=1

= <X (Y2...Yn)> when Y1 = 0

Let *Z denote a meta operator such that ψ*Z = Z.

Then we have the following reductions, for example:

```
<*Z (0 0 1 2)>
      -MA→  <Z (*Z (0 0 1 2))>
      -PA→  <*Z (0 1 2)>
      -MA→  <Z (*Z (0 1 2)>
      -PA→  <*Z (1 2)>
      -MA→  <Z (*Z (1 2))>
      -PA→  (1 2)
```

THE RED/1 LANGUAGE

Introduction

In this section we give the specifications of the language we call Red/1. Although there are many variations possible in the detailed syntax which might be used, we shall specify it completely except for the set of characters used to form atoms. Slightly more elaborate syntax will be introduced in subsequent Red languages. We have used a specific syntax for Red/1 rather than Landin's structure definitions [2] because in this case it is simpler and clearer to be specific. The variations an abstract presentation would permit are fairly obvious.

The notation and methods for describing functions will be the same as in earlier sections.

Syntax

Notation

We use BNF productions with underlined words for variables. Thus the usual BNF production:

<ATOM> ::= *<SYMBOL>* | *<STRING>*

would be written in what follows as:

ATOM ::= *SYMBOL* | *STRING*

Spaces in the productions are used only for appearance and may be ignored. An object language blank space is denoted by *BLANK*.

Basic syntax

QUAD ::= □

ATOM ::= *SYMBOL* | *STRING*

SET ::= (*LIST*) | *SYMBOL*(*LIST*)

APPLICATION ::= <*ITEM BLANK ITEM*>

ITEM ::= *QUAD* | *ATOM* | *SET* | *APPLICATION*

LIST ::= *ITEM* | *ITEM BLANK LIST*

Syntax of atoms

Assumptions

1 The sets *CHARACTER*, *SYSTEM-CHARACTER*, and *STRING-QUOTES* are mutually disjoint.

2 Except for the above assumption the set *CHARACTER* is formally unspecified. However, it is generally assumed to contain letters, digits, and various special characters. In particular it contains the following characters:

* ∅ ⍬

3 The set *RSC*, Reserved-System-Character, is empty for Red/1 but will contain additional system characters for extensions of Red/1.

SYSTEM-CHARACTER ::= (|) | < | > | *BLANK* | *QUAD* | *RSC*

S-ELEMENT ::= *CHARACTER* | *SYSTEM-CHARACTER* | *STRING*

S-LIST ::= *S-ELEMENT* | *S-ELEMENT S-LIST*

STRING ::= ⊏⊐ | ⊏*S-LIST*⊐

SYMBOL ::= *CHARACTER* | *CHARACTER SYMBOL*

STRING-QUOTES ::= ⊏ | ⊐

Constants

The constants of Red/1 are determined as follows:

1 □ is a constant.

2 An atom is a constant.

3 A set with constant members is a constant.

4 No application is a constant.

Semantics

General

We shall describe below a transition function ρ defined for all items. If an item $\underline{X}$ has a meaning, it is obtained by repeatedly applying ρ until a constant item, $\ddot{\rho}\underline{X}$, is produced; $\ddot{\rho}\underline{X}$ is the meaning of $\underline{X}$. The item □, "quad", appearing as the meaning of an item signifies that although the reduction of the item terminates, no useful meaning can be assigned to it. [E.g., an operator was applied to an operand for which it is not defined, except to yield □.] Thus a meaningful item may have a useful meaning or reduce to quad. The reduction of a meaningless item never terminates.

To describe ρ we shall use several auxilliary functions:

Δ The restricted transition function. $\Delta\langle\underline{X}\ \underline{Y}\rangle$ defined for all constant items $\underline{X}$ and $\underline{Y}$.

ψ The regular part function. $\psi\underline{X}$ defined for all constant items $\underline{X}$.

ϵ The primitive execution function. $\epsilon\langle \underline{X}\ \underline{Y}\rangle$ defined for all constant $\underline{X}$ and $\underline{Y}$ not otherwise treated in the definition of Δ.

Variables and their ranges

As in the previous section, underlined letters are used for variables to distinguish them from atoms. Ranges are associated with each variable as follows:

Variable	Range of variable
$\underline{X}$ $\underline{Y}$ $\underline{Z}$ $\underline{X}1$ $\underline{X}n$ etc.	Items
$\underline{R}$	Regular items [for which $\psi\underline{R}=\underline{R}$]
$\underline{M}$	Meta items [for which $\psi\underline{M}\neq\underline{M}$]
$\underline{A}$ $\underline{B}$	Atoms
$\underline{S}$ $\underline{T}$ $\underline{U}$	Symbols

Prefixed sets

Sets beginning with a symbol are called prefixed sets. We shall regard a set prefixed with the special symbol Φ as equivalent to the same set without a prefix. In general we shall not use Φ [denoting the empty symbol] as a prefix. In accordance with these conventions we shall use the expression

$$\underline{S}(\underline{X}1\ldots\underline{X}n)$$

to denote any set, with or without prefix, since when $\underline{S}$ has the value Φ, the expression is equivalent to an unprefixed set. The purpose of this convention is to facilitate the uniform treatment of prefixed and unprefixed sets.

The existence of prefixed sets provides the equivalent

of an unlimited number of types of brackets or of sets.

Description of functions

As before, functions are described by cases, each case giving the form of the argument to which it applies. Each case takes precedence over its successors.

Description of ρ

Let $\underline{C}$ $\underline{C}1$ . . .$\underline{C}n$ denote constant items. Let $\underline{X}1$ denote a non-constant item.

Then ρ is defined as follows:

$\rho\underline{C} = \underline{C}$

$\rho\underline{S}(\underline{X}1 \ldots \underline{X}n) = \underline{S}(\rho\underline{X}1 \ldots \underline{X}n)$

$\rho\underline{S}(\underline{C}1 \ldots \underline{C}n\ \underline{X}1 \ldots \underline{X}m) = \underline{S}(\underline{C}1 \ldots \underline{C}n\ \rho\underline{X}1 \ldots \underline{X}m)$

$\rho\langle\underline{C}1\ \underline{C}2\rangle = \Delta\langle\underline{C}1\ \underline{C}2\rangle$

$\rho\langle\underline{C}\ \underline{X}\rangle = \langle\underline{C}\ \rho\underline{X}\rangle$

$\rho\langle\underline{X}\ \underline{Y}\rangle = \langle\rho\underline{X}\ \underline{Y}\rangle$

The effect of this recursive definition is this: to apply ρ to an item $\underline{X}$, apply Δ to the leftmost innermost application within $\underline{X}$. Thus, for example,

$\rho(A\ \langle\langle \not\beta\ C\rangle\ \langle D\ E\rangle\rangle\ \langle F\ G\rangle) = (A\ \langle\Delta\langle \not\beta\ C\rangle\ \langle D\ E\rangle\rangle\ \langle F\ G\rangle)$

DBL DBL

$= (A\ \langle(C\ C)\ \langle D\ E\rangle\rangle\ \langle F\ G\rangle)$

where (C C) is the value of $\Delta\langle$DBL C$\rangle$.

Description of Δ

The description of Δ is exactly the same as given in the section on Red languages. We repeat it here without headings for easy reference. Let $\psi\underline{R}=\underline{R}$ and $\psi\underline{M}\neq\underline{M}$. Then the following describes Δ by cases:

$\Delta<(\underline{R})\ \underline{Y}> = <\underline{R}\ <\phi\ \underline{Y}>>$ $\quad [\rightarrow <\underline{R}\ \underline{Y}>]$

$\Delta<(\underline{R}\ \underline{X}2...\underline{X}n)\ \underline{Y}> = <\underline{R}\ <(\underline{X}2...\underline{X}n)\ \underline{Y}>>$

$\Delta<(\underline{M})\ \underline{Y}> = <\psi\underline{M}\ ((\underline{M})\ \underline{Y})>$

$\Delta<(\underline{M}\ \underline{X}2...\underline{X}n)\ \underline{Y}> = <\psi\underline{M}\ ((\underline{M}\ \underline{X}2...\underline{X}n)\ \underline{Y})>$

$\Delta<\underline{M}\ \underline{Y}> = <\psi\underline{M}\ (\underline{M}\ \underline{Y})>$

$\Delta<\underline{R}\ \underline{Y}> = \epsilon<\underline{R}\ \underline{Y}>$

Description of ψ

$\psi*(\underline{X}1...\underline{X}n) = (\underline{X}1...\underline{X}n)$

$\psi\underline{X} = \underline{X}$

Description of ϵ

We give below the description by cases of the primitive execution function ϵ. The cases for each primitive operator are headed by the name of the operator and the fifteen primitive operators are grouped under various major headings. In order to exhibit the usual simplicity of the operator's action, a separate case is usually given for the case of unprefixed sets, even though it is included in the cases for prefixed sets.

Note that the last case [16] covers all applications not previously covered in Δ or in ϵ. It assigns the value $\square$, denoting "undefined", to all those situations not otherwise

defined. Also, the following symbols are given a fixed significance in certain contexts by the primitive operators:

- $\emptyset$ Denotes the empty set and the identity operator.
- $\Diamond$ Denotes the empty symbol. [See the operators *PREFIX* and *UNPREFIX* below.]
- *T* Denotes "true".
- *F* Denotes "false".

PAIR-TO-PAIR OPERATORS

1 Reverse

ϵ<*REV* ($\underline{X}$ $\underline{Y}$)> = ($\underline{Y}$ $\underline{X}$)

ϵ<*REV* $\underline{S}$($\underline{X}$ $\underline{Y}$)> = $\underline{S}$($\underline{Y}$ $\underline{X}$)

2 Associate

ϵ<*ASSOC* ($\underline{X}$ ($\underline{Y}$ $\underline{Z}$))> = (($\underline{X}$ $\underline{Y}$) $\underline{Z}$)

ϵ<*ASSOC* $\underline{S}$($\underline{X}$ $\underline{T}$($\underline{Y}$ $\underline{Z}$))> = $\underline{S}$($\underline{T}$($\underline{X}$ $\underline{Y}$) $\underline{Z}$)

3 Apply

ϵ<*APPLY* (($\underline{X}$ $\underline{Y}$) $\underline{Z}$)> = (<$\underline{X}$ $\underline{Y}$> $\underline{Z}$)

ϵ<*APPLY* $\underline{S}$($\underline{T}$($\underline{X}$ $\underline{Y}$) $\underline{Z}$)> = $\underline{S}$(<$\underline{X}$ $\underline{Y}$> $\underline{Z}$)

PAIR-TO-ITEM OPERATORS

4 CAR or First-member-of-a-pair

ϵ<*CAR* ($\underline{X}$ $\underline{Y}$)> = $\underline{X}$

ϵ<*CAR* $\underline{S}$($\underline{X}$ $\underline{Y}$)> = $\underline{X}$

5 Condition

ϵ<*COND* (*T* ($\underline{X}$ $\underline{Y}$))> = $\underline{X}$

ϵ<*COND* (*F* ($\underline{X}$ $\underline{Y}$))> = $\underline{Y}$

ϵ<*COND* $\underline{S}$(*T* $\underline{T}$($\underline{X}$ $\underline{Y}$))> = $\underline{X}$

ϵ<*COND* $\underline{S}$(*F* $\underline{T}$($\underline{X}$ $\underline{Y}$))> = $\underline{Y}$

6 Equal atoms

ϵ<*EQAT* $\underline{S}(\square\ \underline{X})$> = $\square$

ϵ<*EQAT* $\underline{S}(\underline{X}\ \square)$> = $\square$

ϵ<*EQAT* $\underline{S}(\underline{A}\ \underline{A})$> = *T*

ϵ<*EQAT* $\underline{S}(\underline{X}\ \underline{Y})$> = *F*

Thus *EQAT* yields *T* for a pair of equal atoms and *F* for all other pairs not involving $\square$.

7 Union

ϵ<*UNION* $(\underline{X}\ \phi)$> = $(\underline{X})$

ϵ<*UNION* $(\underline{X}\ (\underline{Y}1...\underline{Y}n))$> = $(\underline{X}\ \underline{Y}1...\underline{Y}n)$

ϵ<*UNION* $\underline{S}(\underline{X}\ \phi)$> = $\underline{S}(\underline{X})$

ϵ<*UNION* $\underline{S}(\underline{X}\ \underline{T}(\underline{Y}1...\underline{Y}n))$> = $\underline{S}(\underline{X}\ \underline{Y}1...\underline{Y}n)$

ITEM-TO-PAIR OPERATORS

8 Double

ϵ<*DBL* $\underline{X}$> = $(\underline{X}\ \underline{X})$

9 Separate

ϵ<*SEP* $(\underline{X})$> = $(\underline{X}\ \phi)$

ϵ<*SEP* $(\underline{X}1...\underline{X}n)$> = $(\underline{X}1\ (\underline{X}2...\underline{X}n))$

ϵ<*SEP* $\underline{S}(\underline{X})$> = $\underline{S}(\underline{X}\ \phi)$

ϵ<*SEP* $\underline{S}(\underline{X}1\ .\ .\ .\underline{X}n)$> = $\underline{S}(\underline{X}1\ \underline{S}(\underline{X}2...\underline{X}n))$

ITEM-TO-ITEM OPERATORS

10 Identity operator: ϕ

ϵ<ϕ $\underline{X}$> = $\underline{X}$

11 Equals quad

ϵ<*QUAD* $\square$> = *T*

ϵ<*QUAD* $\underline{X}$> = *F*

OPERATORS FOR SET PREFIXES

12 Prefix

ϵ<*PREFIX* $\underline{S}$(◊ $\underline{T}$($\underline{X}$1...$\underline{X}$n))> = ($\underline{X}$1...$\underline{X}$n)

ϵ<*PREFIX* $\underline{S}$($\underline{T}$ $\underline{U}$($\underline{X}$1...$\underline{X}$n))> = $\underline{T}$($\underline{X}$1...$\underline{X}$n)

13 Unprefix

ϵ<*UNPREFIX* ($\underline{X}$1...$\underline{X}$n)> = (◊ ($\underline{X}$1...$\underline{X}$n))

ϵ<*UNPREFIX* $\underline{S}$($\underline{X}$1...$\underline{X}$n)> = ($\underline{S}$ ($\underline{X}$1...$\underline{X}$n))

OPERATORS TO CONVERT BETWEEN ATOMS AND SETS

Here we shall use $\underline{C}$1$\underline{C}$2...$\underline{C}$n to denote a sequence of n string-characters [where *STRING-CHARACTER* ::= *CHARACTER* | *SYSTEM-CHARACTER* | *STRING-QUOTES*]. In defining the operators *ENCODE* and *DECODE* we shall need the function α which maps string-characters into one- and two-character symbols:

α□ = *QU* α*BLANK* = *BL*

α⊧ = *LS* α⊣ = *RS*

α(= *LP* α) = *RP*

α< = *LA* α> = *RA*

$\alpha\underline{C}$ = $\underline{C}$ [where $\underline{C}$ ϵ *CHARACTER*]

Thus α is a one-one mapping of *STRING-CHARACTER* onto a set of one- and two-character symbols.

14 Decode

ϵ<*DECODE* $\underline{C}$1$\underline{C}$2...$\underline{C}$n> = ($\alpha\underline{C}$1 $\alpha\underline{C}$2...$\alpha\underline{C}$n)

DECODE is defined above only for atoms.

ϵ<*DECODE* $\underline{X}$> = □

Examples:

ϵ<*DECODE RED*> = (*R E D*)

ϵ<*DECODE* ⊧<(*A*) *B*>⊣> = (*LS LA LP A RP BL B RA RS*)

15 Encode

ϵ<*ENCODE* $\underline{S}(\alpha\underline{C}1\ \alpha\underline{C}2...\alpha\underline{C}n)$> = $\underline{C}1\underline{C}2...\underline{C}n$

When $\underline{C}1\underline{C}2...\underline{C}n$ is an atom [see syntax].

Thus *ENCODE* is the inverse of *DECODE* except that a prefix may be adjoined to the argument with no effect on the result.

Otherwise ϵ<*ENCODE* $\underline{S}(\underline{X}1...\underline{X}n)$> = $\square$

NON-PRIMITIVE OR IMPROPER ARGUMENT

16 $\epsilon<\underline{X}\ \underline{Y}> = \square$

Red/1 Summary

SYNTAX

[BNF with {*A*|*B*} for a sequence (possibly empty) of members of *A* or *B*]

Basic Syntax

ATOM ::= *SYMBOL* | *STRING*

SET ::= (*LIST*) | *SYMBOL*(*LIST*)

APPLICATION ::= <*ITEM BLANK ITEM*>

ITEM ::= □ | *ATOM* | *SET* | *APPLICATION*

LIST ::= *ITEM* {*BLANK ITEM*}

Syntax of atoms [with abbreviations: *CHAR* = *CHARACTER* etc.]

Notes: 1) *CHAR*∩*SYS-CHAR*∩*STR-QTS* is empty. 2) * ◊ ∅ ∈ *CHAR*
3) *RSC* is empty.

SYS-CHAR ::= (|) | < | > | □ | *BLANK* | *RSC*

STRING ::= ⊧{*CHAR* | *SYS-CHAR* | *STRING*}⊣

SYMBOL ::= *CHAR* {*CHAR*}

STR-QTS ::= ⊧ | ⊣

Constants: Atoms and sets with constant members.

Non-constants: Applications and sets with non-constant members.

SEMANTICS

Description of ρ Let *SEQ* denote *X*1...*X*n for n≥1, with at least one *X*i a non-constant. Let ρ*SEQ* denote *X*1...ρ*X*i...*X*n, where *X*i is the leftmost non-constant of *SEQ*. Let *C*, *C*1, *C*2 be constants.

ρ*C* = *C*

ρ(*SEQ*) = (ρ*SEQ*)

ρ<*SEQ*> = <ρ*SEQ*>

ρ<*C*1 *C*2> = Δ<*C*1 *C*2>

Description of Δ Let $\psi R = R$, $\psi M \neq M$

$\Delta<(R)\ Y> = <R\ <\phi\ Y>>$ [→ $<R\ Y>$]

$\Delta<(R\ X2 \ldots Xn)\ Y> = <R\ <(X2 \ldots Xn)\ Y>>$

$\Delta<(M)\ Y> = <\psi M\ ((M)\ Y)>$

$\Delta<(M\ X2 \ldots Xn)\ Y> = <\psi M\ ((M\ X2 \ldots Xn)\ Y)>$

$\Delta<M\ Y> = <\psi M\ (M\ Y)>$

$\Delta<R\ Y> = \epsilon<R\ Y>$

Description of ψ

$\psi*(X1 \ldots Xn) = (X1 \ldots Xn)$

$\psi X = X$

Description of ϵ Let X Y Z etc., be items, let A be an atom, and let S T U be symbols.

Operator	Operand	Result
REV	$S(X\ Y)$	$S(Y\ X)$
ASSOC	$S(X\ T(Y\ Z))$	$S(T(X\ Y)\ Z)$
APPLY	$S(T(X\ Y)\ Z)$	$S(<X\ Y>\ Z)$
CAR	$S(X\ Y)$	X
COND	$S(T\ T(X\ Y))$	X
	$S(F\ T(X\ Y))$	Y
EQAT	$S(\square\ X)$	$\square$
	$S(X\ \square)$	$\square$
	$S(A\ A)$	T
	$S(X\ Y)$	F
UNION	$S(X\ \phi)$	$S(X)$
	$S(X\ T(Y1 \ldots Yn))$	$S(X\ Y1 \ldots Yn)$
DBL	X	$(X\ X)$

Operator	Operand	Result
SEP	$\underline{S}(\underline{X})$	$\underline{S}(\underline{X}\ \emptyset)$
	$\underline{S}(\underline{X}1\ \underline{X}2...\underline{X}n)$	$\underline{S}(\underline{X}1\ \underline{S}(\underline{X}2...\underline{X}n))$
$\emptyset$	$\underline{X}$	$\underline{X}$
QUAD	□	*T*
	$\underline{X}$	*F*
PREFIX	$\underline{S}(\lozenge\ \underline{T}(\underline{X}1...\underline{X}n))$	$(\underline{X}1...\underline{X}n)$
	$\underline{S}(\underline{T}\ \underline{U}(\underline{X}1...\underline{X}n))$	$\underline{T}(\underline{X}1...\underline{X}n)$
UNPREFIX	$(\underline{X}1...\underline{X}n)$	$(\lozenge\ (\underline{X}1...\underline{X}n))$
	$\underline{S}(\underline{X}1...\underline{X}n)$	$(\underline{S}\ (\underline{X}1...\underline{X}n))$
DECODE	$\underline{C}1\underline{C}2...\underline{C}n$	$(\alpha\underline{C}1\ \alpha\underline{C}2\ ...\alpha\underline{C}n)$
	when $\underline{C}1\underline{C}2...\underline{C}n$ is an atom	
ENCODE	$\underline{S}(\alpha\underline{C}1\ \alpha\underline{C}2\ ...\ \underline{C}n)$	$\underline{C}1\underline{C}2...\underline{C}n$
	when $\underline{C}1\underline{C}2...\underline{C}n$ is an atom	

Where: $\alpha\square$ = *QU* $\alpha\underline{BLANK}$ = *BL* $\alpha\lfloor$ = *LS* $\alpha\rfloor$ = *RS*

$\alpha($ = *LP* $\alpha)$ = *RP* $\alpha<$ = *LA* $\alpha>$ = *RA*

$\alpha\underline{C} = \underline{C}$ [where $\underline{C} \in \underline{CHAR}$]

$\underline{X}$	$\underline{Y}$	□

Remarks

The above definition of the Red/1 language is believed to be entirely complete and unambiguous once the set *CHARACTER* is decided upon. By this we mean that one can decide whether any finite string is an item or not and that the meaning of every meaningful item is uniquely and unambiguously determined. [Recall that F is meaningful whenever $\rho^n F$ is a constant for some integer n, in which case $\rho^n F$ $[=\ddot{\rho}F]$ is the meaning of F.]

Although several current programming languages are perhaps similarly well-defined, their syntax and semantics are considerably more complex. Their semantics always require one to understand transitions in a more complicated space than the space of formulas.

The semantics of Red/1 involve only transitions on the space of items. To perform a transition on an item one need only find its leftmost application with constant members and transform it using one of the four basic transition rules of Δ.

Every Red/1 item is a formally valid operator [the first member of an application] or operand [the second member]. There are no constructs in the language solely relating to function definition; there is no equivalent of a lambda-expression and none are needed, since there are no variables. A non-primitive function is simply a composite operator, a set of simpler operators.

The number of primitive operators may seem unduly large. However, this is the price which is paid partly for the lack of variables and partly for certain not-entirely-necessary completeness features: for prefixed sets, which provide the equivalent of an endless variety of brackets or types of sets, and for the ability to deal with strings containing the system-characters of Red/1. There are, of course, smaller but sufficient sets of primitive operators.

A proof that Red/1 is a parenthetic--and hence by Th/PA7--a reduction language is not included in this paper. The proof is straightforward if one notes that the definition of subformula excludes as subformulas proper substrings of an atom within a formula.

Examples of defined operators

Notation

In the following examples we shall write, as before, for example:

$$\underline{X} \text{ -RC}\rightarrow \underline{Y}$$

to indicate that $\underline{Y}$ is obtained from $\underline{X}$ by using regular composition, RC, on some innermost application within $\underline{X}$. We shall write

$$\underline{X} \rightarrow \underline{Y}$$

to indicate that $\underline{Y}$ is obtained by one or more uses of Δ within $\underline{X}$.

We shall adopt some variations in the way we write

items in order to make them easier to read. Thus we shall use the following shorthand notation for certain Red/1 items:

Shorthand notation	Red/1 item
X.*Y*	<*X* *Y*>
W.*X*.*Y*.*Z*	<*W* <*X* <*Y* *Z*>>>
X:*Y*	(*X* *Y*)
W:*X*:*Y*:*Z*	(*W* (*X* (*Y* *Z*)))
Etc.	Etc.
X:*Y*.*Z*	<(*X* *Y*) *Z*>

The last example is to indicate that we shall assume ":" to take precedence over "."

We shall also allow ourselves to use the following synonyms for the system-characters of Red/1 whenever their use is convenient or clearer:

Character	Synonyms
BLANK	, ; ← → |
(	[⊂
)	] ⊃

Of course, we shall always use matched bracket-pairs. The above conventions mean that the following expression

[A:B C→(D.E,⊂F G H⊃)]

is a shorthand way of writing

((A B) C (<D E> (F G H)))

Definitions of operators

Although Red/1 has no facilities for employing defined operators, we shall nevertheless define new operators, e.g.,

ROT = (ASSOC REV ASSOC REV)

and regard such a definition as meaning that any item containing the symbol *ROT* is a shorthand notation for the same item with *ROT* everywhere replaced by the set on the right.

Prefixes

The examples below will show the general properties of the defined operators except how they deal with prefixed sets [all operand sets will be assumed to be without a prefix]. The generalization of the behavior of defined operators for prefixed sets is direct and obvious.

Analysis of examples

Some of the operators whose effects we analyze in detail below make use of others whose definitions and effects are given subsequently in table form. The examples below are also included in the later tables.

First member of a set

Definition

∘1 = (*CAR SEP*)

Effect

<∘1 ($\underline{X}$1 . . . $\underline{X}$n)> =

<(*CAR SEP*) (*X*1 . . . *X*n)>

-RC→ <*CAR* <(*SEP*) (*X*1 . . .*X*n)>>

-RC→ <*CAR* <*SEP* (*X*1 . . . *X*n)>>

-PA→ <*CAR* (*X*1 *W*)> where *W*=∅ or *W*=(*X*2 . . . *X*n)

-PA→ *X*1

The same sequence of reductions looks like this in our shorthand notation:

(*CAR SEP*).(*X*1 . . . *X*n)

-RC→ *CAR*.(*SEP*).(*X*1 . . . *X*n)

-RC→ *CAR.SEP*.(*X*1 . . . *X*n)

-PA→ *CAR*.(*X*1 *W*)

-PA→ *X*1

Tail

Definition

TL = (*CAR REV SEP*)

Effect

TL.*X* =

(*CAR REV SEP*).*X*

-RC→ *CAR*.(*REV SEP*).*X*

-RC→ *CAR.REV*.(*SEP*).*X*

-RC→ *CAR.REV.SEP*.*X*

-PA→ *CAR. REV*.□ [*X* ∈ *ATOM*]

CAR.REV.(*X*1 ∅) [*X* = (*X*1)]

CAR.REV.(*X*1 (*X*2...*X*n)) [*X* = (*X*1...*X*n)]

-PA→ *CAR*.□ [*X* ∈ *ATOM*]

CAR.(∅ *X*1) [*X* = (*X*1)]

CAR.((*X*2 . . . *X*n) *X*1) [*X* = (*X*1...*X*n)]

$$-PA\rightarrow \square \qquad [\underline{X} \in \underline{ATOM}]$$

$$\phi \qquad [\underline{X} = (\underline{X}1)]$$

$$(\underline{X}2\ .\ .\ .\ \underline{X}n) \qquad [\underline{X} = (\underline{X}1 \ldots \underline{X}n)]$$

In applying *SEP* to $\underline{X}$ above, there are three cases, depending on the nature of $\underline{X}$. Each case is followed by the condition which determines it. The order of the cases is preserved within each transition and the conditions may be omitted after they are given initially. When a case later has subcases, the conditions for the subcases will make this apparent.

The above reductions prove that the effect of *TL* is given by the following description by cases:

$TL.\underline{A} \rightarrow \square$

$TL.(\underline{X}) \rightarrow \phi$

$TL.(\underline{X}1\ .\ .\ .\ \underline{X}n) \rightarrow (\underline{X}2\ .\ .\ .\ \underline{X}n)$

Adjoin

Definition

$ADJ = *(CAR\ ROT)$

Effect

$ADJ:\underline{X}.\underline{Y} =$

$(ADJ\ \underline{X}).\underline{Y} =$

$(*(CAR\ ROT)\ \underline{X}).\underline{Y}$

$-MC\rightarrow (CAR\ ROT).((ADJ\ \underline{X})\ \underline{Y})$

$\rightarrow CAR.((\underline{X}\ \underline{Y})\ ADJ)$

$\rightarrow (\underline{X}\ \underline{Y})$

Thus the effect of *ADJ* is:

$ADJ:\underline{X}.\underline{Y} = (ADJ\ \underline{X}).\underline{Y} \rightarrow (\underline{X}\ \underline{Y})$

And

Definition

AND = (*COND COND REV FST*:[*TRPRS REV ADJ*:((*T F*) (*F F*)) *DBL*])

Effect

AND.(*T T*) → *T*

AND.(*T F*) → *F*

AND.(*F T*) → *F*

AND.(*F F*) → *F*

AND.(*X Y*) → □

Remark

AND is symmetric and so defined that it always yields quad or a truth value. *AND* is thus a "clean" operator. Contrast this with the asymmetric LISP definition,

$$p \wedge q = (p \rightarrow q, T \rightarrow F)$$

from which we get, for example:

$$T \wedge (A.B) = (A.B)$$

It is important that operators have clean definitions, i.e., yield □ in non-useful situations, in order to simplify the effect-proofs of operators which employ them.

Conditional expression from argument

Definition

CONDARG = ⋆(*APP COND APPLY SEC*:*DISTFUNC DISTFUNC*)

Effect

(*CONDARG*).*Z*

→ *APP.COND.APPLY.SEC*:*DISTFUNC.DISTFUNC.*((*CONDARG*) *Z*)

→ *APP.COND.APPLY.SEC*:*DISTFUNC.*□

→ □

(CONDARG $\underline{X}1...\underline{X}n).\underline{Z}$

→ Ψ*CONDARG.[(CONDARG* $\underline{X}1...\underline{X}n$) $\underline{Z}$]

→ *H1.*(($\underline{X}1$ $\underline{Z}$) [(*CONDARG*) $\underline{Z}$]) [n=1]

H1.(($\underline{X}1$ $\underline{Z}$) [(*CONDARG* $\underline{X}2...\underline{X}n$) $\underline{Z}$]) [n>1]

where *H1 = (APP COND APPLY SEC:DISTFUNC)*

→ *H2.*(($\underline{X}1$ $\underline{Z}$) □) [n=1]

H2.(($\underline{X}1$ $\underline{Z}$) [($\underline{X}2$ $\underline{Z}$) [(*CONDARG*) $\underline{Z}$]]) [n=2]

H2.(($\underline{X}1$ $\underline{Z}$) [($\underline{X}2$ $\underline{Z}$) [(*CONDARG* $\underline{X}3...\underline{X}n$) $\underline{Z}$]]) [n>2]

where *H2 = (APP COND APPLY)*

→ *H3.*($\underline{X}1.\underline{Z}$ □) [n=1]

H3.($\underline{X}1.\underline{Z}$ [($\underline{X}2$ $\underline{Z}$) [(*CONDARG*) $\underline{Z}$]]) [n=2]

H3.($\underline{X}1.\underline{Z}$ [($\underline{X}2$ $\underline{Z}$) [(*CONDARG* $\underline{X}3...\underline{X}n$) $\underline{Z}$]]) [n>2]

where *H3 = (APP COND)*

→ *APP.*□ [n=1 or n>1 and $\underline{X}1.\underline{Z} \neq T$ and $\neq F$]

APP.($\underline{X}2$ $\underline{Z}$) [n=2 $\underline{X}1.\underline{Z} = T$]

APP.[(*CONDARG*) $\underline{Z}$] [n=2 $\underline{X}1.\underline{Z} = F$]

APP.($\underline{X}2$ $\underline{Z}$) [n>2 $\underline{X}1.\underline{Z} = T$]

APP.[(*CONDARG* $\underline{X}3...\underline{X}n$) $\underline{Z}$] [n>2 $\underline{X}1.\underline{Z} = F$]

→ □ [n=1]

$\underline{X}2.\underline{Z}$ [n=2 $\underline{X}1.\underline{Z} = T$]

(*CONDARG*).$\underline{Z}$ [n=2 $\underline{X}1.\underline{Z} = F$]

$\underline{X}2.\underline{Z}$ [n>2 $\underline{X}1.\underline{Z} = T$]

(*CONDARG* $\underline{X}3...\underline{X}n$).$\underline{Z}$ [n>2 $\underline{X}1.\underline{Z} = F$]

Thus the above reductions prove that the effect of *CONDARG*

is the following:

(*CONDARG*).$\underline{Z}$ → □

(*CONDARG* $\underline{X}$1).$\underline{Z}$ → □

(*CONDARG* $\underline{X}$1 $\underline{X}$2).$\underline{Z}$ → □ when $\underline{X}$1.$\underline{Z}$ = *F*

(*CONDARG* $\underline{X}$1...$\underline{X}$n).$\underline{Z}$ → $\underline{X}$2.$\underline{Z}$ when n≥2 and $\underline{X}$1.$\underline{Z}$ = *T*

(*CONDARG* $\underline{X}$1...$\underline{X}$n).$\underline{Z}$ → (*CONDARG* $\underline{X}$3...$\underline{X}$n).$\underline{Z}$

when n>2 and $\underline{X}$1.$\underline{Z}$ = *F*

(*CONDARG* $\underline{X}$1...$\underline{X}$n).$\underline{Z}$ → □ when n≥2 and $\underline{X}$1.$\underline{Z}$ ≠*T* and ≠*F*

Remarks

Note that the proof of the effect of operators like Tail or Adjoin is easily mechanizable: there is a not-too-complicated process which, given the definition of such an operator, will produce a description by cases, as illustrated. It is not surprising that this should be so for operators whose definitions are not recursive in nature. It is pleasing to note that the same process appears to apply to operators such as *CONDARG* which invoke self-references.

At this writing there has not been time to make a thorough study of the implications of Red-like languages for simplifying proofs about programs [i.e., operators]. There is much work to be done to review existing results concerning proofs about programs in the new context of Red languages. However, the above examples indicate that there is hope for some useful simplifications.

Defined operators

The following list gives the definition and effect of a number of operators. The effect of each is given by cases,

with each case given as:

operand → result

Several cases may appear on one line separated by semicolons. A final implicit case is assumed for each description: $X \rightarrow \sqcup$, indicating that for all remaining cases the operator yields $\sqcup$. As before, X, Y, Z, etc., denote items and A denotes an atom.

Full name

Definition

Operator | Effect

First member of a set

∘1 = (*CAR SEP*)

∘1 | $A \rightarrow \sqcup$; $(X1 \ldots Xn) \rightarrow X1$

Tail

TL = (*CAR REV SEP*)

TL | $A \rightarrow \sqcup$; $(X) \rightarrow \emptyset$; $(X1\ X2 \ldots Xn) \rightarrow (X2 \ldots Xn)$

Rotate

ROT = (*ASSOC REV ASSOC REV*)

ROT | $((X\ Y)\ Z) \rightarrow ((Y\ Z)\ X)$

Adjoin

ADJ = *(*CAR ROT*)

(*ADJ* X) | $Y \rightarrow (X\ Y)$

First

FST = *(*APPLY ASSOC CAR ROT*)

(*FST* X) | $(Y\ Z) \rightarrow (\langle X\ Y\rangle\ Z)$

Second

*SEC = *(REV APPLY ASSOC REV FST:REV REV CAR ROT)*

(SEC X) | (Y Z) → (Y <X Z>)

Constant

*C = *(CAR REV CAR)*

(C X) | Y → X

Not

NOT = (COND REV ADJ:[F T])

NOT | T → F; F → T

Equals empty set

EQ∅ = (EQAT ADJ:∅)

EQ∅ | □ → □; ∅ → T; X → F

Atom

ATOM = (EQAT DBL)

ATOM | □ → □; A → T; X → F

And

AND = (COND COND REV FST:[TRPRS REV ADJ:((T F) (F F)) DBL])

AND | (T T) → T; (T F) → F; (F T) → F; (F F) → F

(X Y) → □

Transpose pairs

TRPRS = (FST:REV REV ROT FST:[ASSOC REV] ASSOC)

TRPRS | ((X1 X2) (Y1 Y2)) → ((X1 Y1) (X2 Y2))

Application

APP = (CAR APPLY DBL)

APP | (X Y) → <X Y>

Two applications

2APP = (REV APPLY REV APPLY)

2APP | ((X1 X2) (Y1 Y2)) → (<X1 X2> <Y1 Y2>)

Two functions

```
2F = *(2APP TRPRS SEC:DBL FST:TL)
(2F X1 X2) | Y → (<X1 Y> <X2 Y>)
```

Three functions

```
3F = *(SEC:[2APP TRPRS] APPLY TRPRS
        SEC:[CAR ROT DBL DBL] FST:[SEP TL])
(3F X1 X2 X3) | Y → (<X1 Y> (<X2 Y> <X3 Y>))
```

Second member of a set

```
∘2 = (∘1 TL)
∘2 | A → □;  (X1) → □; (X1 X2...Xn) → X2
```

Set to quad

```
SETQUAD = (COND SEC:ADJ:□ FST:QUAD:∘1 DBL)
SETQUAD | A → □;  (□) → □;  (□ X2...Xn) → □;  X → X
```

Distribute function

```
DISTFUNC = (TRPRS SEC:DBL FST:SETQUAD:[2F ∘2 F1])
        where  F1 = (UNION SEC:TL SEP)
DISTFUNC | ((X Y1) Z) → ((Y1 Z) ((X) Z));
        ((X Y1...Yn) Z) → ((Y1 Z) ((X Y2...Yn) Z))
```

Conditional expression from argument

```
CONDARG = *(APP COND APPLY SEC:DISTFUNC DISTFUNC)
(CONDARG) | Z → □
(CONDARG X1) | Z → □
(CONDARG X1 X2) | Z → □  when <X1 Z> = F
(CONDARG X1...Xn) | Z → <X2 Z>   when n≥2 and <X1 Z> = T
(CONDARG X1...Xn) | Z → <(CONDARG X3...Xn) Z>
                    when n>2 and <X1 Z> = F
(CONDARG X1...Xn) | Z → □  when n≥2 and <X1 Z> ≠T and ≠F
```

RED/2 LANGUAGES

Introduction

The principal difference between the Red/1 language and Red/2 languages is that the semantics of the latter depend upon a defining set. Red/2 languages have a syntax corresponding to that of Red/1 augmented by the shorthand notation and system-character synonyms used in the examples of the last section. To each defining set there corresponds a particular Red/2 language. The Red/2 language for the empty defining set is precisely Red/1 with augmented syntax and two new primitive operators which give information about the defining set.

The semantics of Red/2 languages are not affected by the augmented syntax: an item is first converted into its Red/1 equivalent before it is reduced. Furthermore, the descriptions of the functions ρ and Δ for Red/2 languages are identical to those for Red/1. Only the definitions of the auxilliary functions ψ and ϵ are different and only by the addition of cases depending on the defining set and for the new primitives.

Language extensions

Red/2 languages typify our simple notion of the extension of a language, in this case, of Red/1. The formulas of the extension are merely synonyms of formulas of the original. The semantics of the original are given by primary functions [ρ and Δ] which describe the basic rules of interpretation and of functional composition. These

employ secondary functions [ψ and ϵ] which describe the behavior of the primitive operators of the language and the regular/meta classification. The semantics of the extension are given by the same primary function descriptions employing altered--or preferably--extended, secondary functions.

We hope to convince the reader that such simple extensions of Red/1 can yield very expressive languages. That is to say that the syntax of Red/1 and the ρ and Δ part of its semantics are rich enough to contain synonyms for the formulas of very powerful languages.

Syntax

Basic syntax

QUAD, *ATOM*, *APPLICATION*, and *LIST* are defined by the same productions as in Red/1, except that *MARK* replaces *BLANK* therein.

PAIR ::= *ITEM*:*ITEM*

TERM ::= *ITEM*.*ITEM*

PLAINSET ::= (*LIST*) | [*LIST*] | ⊂*LIST*⊃

SET ::= *PLAINSET* | *SYMBOL* *PLAINSET*

ITEM ::= *QUAD* | *ATOM* | *SET* | *PAIR* | *APPLICATION* | *TERM*

MARK ::= *BLANK* | , | ; | ← | → | [the sign "|"]

[Recall that a term *X*.*Y* is equivalent to <*X* *Y*> and that a pair *X*:*Y* is equivalent to (*X* *Y*). Both associate to the right and pairs take precedence over terms.]

Syntax of atoms

Again, all assumptions and productions remain the same as in Red/1 except that *RSC* [reserved system character] is now specified to the extent that the following belong to it:

[ ] ⊂ ⊃ , ; ← → | . :

[And α, in the definition of *ENCODE* and *DECODE* under ϵ, is extended and appropriately defined for these new system-characters of Red/2.]

Constants

The constants of Red/2 are those items which convert into Red/1 constant synonyms under the process described below.

Semantics

General

There are a number of ways of treating the new syntax of Red/2. Some of these can preserve the integrity of the new set of brackets so they could be made to reappear in the result, by encoding them using a particular prefix for each. In the present treatment we obliterate the distinction between all new brackets in converting Red/2 items into Red/1 items.

Conversion to Red/1 synonyms

The first step in finding the meaning of a Red/2 item *X* is to convert it into its Red/1 synonym as follows:

1) Replace every occurence within *X* of a non-Red/1 system-character of Red/2, which is not enclosed in

string-quotes, as follows:

Character to be replaced	Replacement
[⊂	(
] ⊃	)
, ; ← → |	*BLANK*

2) Replace every *PAIR*, *Y*:*Z*, within *X* which is not enclosed in string-quotes by the pair (*Y* *Z*), always choosing a *PAIR* *Y*:*Z* such that *Z* is not itself a *PAIR*.

3) After every *PAIR* within *X* has been replaced by a pair [a set with two members], replace every *TERM*, *Y*.*Z*, which is not enclosed in string-quotes, by <*Y* *Z*>, always choosing a *TERM* *Y*.*Z* such that *Z* is not itself a *TERM*.

The defining set ∇ of a Red/2 language

A Red/2 item after the above conversion is syntactically a Red/1 item. Its meaning is determined by the functions ρ, Δ, ψ, and ϵ, as in Red/1. The definitions of ρ and Δ are identical to those for Red/1. But ψ and ϵ depend upon a defining set, ∇, which determines the meaning of the defined primitives of the language. A defining set is a set, all of whose members are pairs, each having one of the following two forms:

(*SYMBOL* *BLANK* *CONSTANT*)

or

*(*SYMBOL* *BLANK* *CONSTANT*)

where *CONSTANT* is a constant Red/1 item. [We may write a Red/2 constant in a definition and assume the above conversion.]

A pair ($\underline{S}$ $\underline{X}$) belonging to ∇ is a <u>definition of</u> $\underline{S}$. And $\star(\underline{S}\ \underline{X})$ belonging to ∇ is called a <u>meta definition of</u> $\underline{S}$. In each case $\underline{X}$ is the <u>definiens</u> of $\underline{S}$. We shall usually write definitions and meta definitions as we did in Red/1 [see conversion of Red/1 definitions below].

A defining set may not have two members whose first members are the same. Thus two definitions of the same symbol are prohibited.

Note that a definition of *A* may be self-referencing, that is, *A* may occur in its own definiens. And of course *A* may occur in the definiens of *B* while *B* occurs in the definiens of *A*, and so on. Such circular definitions of operators may of course result in non-terminating reductions of items employing them.

<u>Example</u>

The following defining set adjoins the defined primitives $\circ 1$ and *FST* to the primitives of Red/1:

∇ = (($\circ 1$ (*CAR SEP*)) $\star$(*FST* (*APPLY ASSOC CAR ROT*)))

<u>Description of</u> ρ <u>and</u> Δ

The descriptions of ρ and Δ for Red/2 are identical to those given for Red/1. It should be noted, however, that this ρ is not precisely the transition function of a Red/2 language. If ω is the conversion function which produces synonomous Red/1 items from Red/2 items, then, strictly speaking, the transition function of a Red/2 language is $\rho\omega$. After the first use of $\rho\omega$, $\rho\omega$ is identical to ρ, since

non-Red/1 system-characters cannot occur thereafter outside of string-quotes [note the change in *ENCODE* and *DECODE* below].

Description of $\underline{\psi}$

$\psi*(\underline{X}1...\underline{X}n) = (\underline{X}1...\underline{X}n)$

$\psi\underline{S} = \underline{X}$ when $*(\underline{S}\ \underline{X}) \in \nabla$

$\psi\underline{Y} = \underline{Y}$

Thus $\psi\underline{X}$ is the same as for Red/1 except when $\underline{X}$ is the name of a defined primitive meta operator.

Description of ϵ

Let ϵ' denote the primitive execution function of Red/1 [denoted there by ϵ] except that *ENCODE* and *DECODE* are altered by suitably extending α to assign codes to the new system-characters of Red/2. Here ϵ is the primitive execution function of Red/2.

The primitive execution function ϵ for Red/2 is the same as for Red/1 except: (1) It defines two new primitives, and (2) it depends on the defining set ∇. The two new primitives, *PSI* and *DEF*, give information about the defining set. Let $\underline{S}$ be a symbol. Then:

$\epsilon<DEF\ \underline{S}> = \underline{Y}$ when $(\underline{S}\ \underline{Y}) \in \nabla$

$\epsilon<DEF\ \underline{X}> = \underline{X}$ otherwise

$\epsilon<PSI\ \underline{X}> = \psi\underline{X}$ [ψ as defined above]

$\epsilon<\underline{S}\ \underline{Z}> = <\underline{X}\ \underline{Z}>$ when $(\underline{S}\ \underline{X}) \in \nabla$

$\epsilon<\underline{X}\ \underline{Z}> = \epsilon'<\underline{X}\ \underline{Z}>$

In other words $\epsilon<\underline{Y}\ \underline{Z}>$ is the same as for Red/1 except when $\underline{Y}$ is *PSI* or *DEF* or when $\underline{Y}$ is the name of a regular defined

operator, namely, when $\underline{Y}$ is a symbol and $(\underline{Y}\ \underline{X}) \in \nabla$. In this case $\epsilon\langle\underline{Y}\ \ \underline{Z}\rangle$ is obtained by replacing $\underline{Y}$ by its definiens $\underline{X}$, yielding $\langle\underline{X}\ \underline{Z}\rangle$.

Remarks

It is interesting to note the difference between the definitions used for Red/1 operators and the effect of definitions in the defining set ∇ of a Red/2 language. Under the conventions used in the section on Red/1, if we make the definition

$$OP = *(CAR\ CAR)$$

then we regard the item

$$\langle OP\ A\rangle$$

as shorthand for the item

$$\langle *(CAR\ CAR)\ A\rangle$$

and this reduces as follows:

$$\rightarrow CAR.CAR.(*(CAR\ CAR)\ A)$$

$$\rightarrow CAR.*(CAR\ CAR)$$

$$\rightarrow CAR$$

If we now look at the reduction of $\langle OP\ A\rangle$ in a Red/2 language such that

$$*(OP\ (CAR\ CAR)) \in \nabla$$

we get the following:

$\langle OP\ A\rangle$

$\text{-MA}\rightarrow \langle(CAR\ CAR)\ (OP\ A)\rangle$ [since $\psi OP = (CAR\ CAR)$ and OP is not, in this case, a shorthand for something else but merely a symbol]

$$\rightarrow CAR.OP$$

$$\rightarrow \square$$

The different results in the two instances of course result from the dependency of *OP* on its own structure and the fact that that structure is different in the two cases, being **(CAR CAR)* in the first case and *OP* in the second. Except for situations which depend on the structure of defined operators, the two methods will yield the same result when both are applicable.

Under the Red/1 shorthand conventions, it is impossible to write a definiens for the definition

LAST = (CONDARG (EQØ TL) ∘1 (C T) (LAST TL))

which does not have in it an occurrence of the definiendum, *LAST*. Hence the definition is ~~meaningless~~ useless. However, the Red/2 definition

(LAST (CONDARG (EQØ TL) ∘1 (C T) (LAST TL)))

occurring in the defining set ∇ along with definitions of all the other needed operators, is perfectly valid and will yield, for example,

LAST.(A B C) → *C*

Although one can easily construct self-referencing operators in Red/1 [see the definition of *CONDARG*], Red/2 definitions in ∇ offer a new and more direct method for self-referencing, as in the above example for *LAST*.

Conversion of Red/1 definitions to Red/2 definitions

A Red/1 definition of the form

SYMBOL = (*X*1...*X*n)

converts to the following Red/2 definition

(*SYMBOL* (*X*1...*X*n))

for inclusion in ∇. The Red/1 definition of the form

SYMBOL = *(*X*1...*X*n)

converts to the Red/2 meta definition

*(*SYMBOL* (*X*1...*X*n))

By converting all the definitions of operators given in the section on Red/1 and including them in ∇, we obtain a Red/2 language in which all those operators will have the effects given for them. Red/1 items using those operators are often shorthand for Red/1 items of very considerable size. No such expansion of items occurs in Red/2, even during the reduction of such items.

Variable-free programming techniques, universal operators, parsing, and controlling order of execution

Universal operators

By the term, "universal operator", [or program, or function] we usually mean an operator of a language L which explains the interpretation of an arbitrary formula of L. For complete languages a universal operator U produces the meaning M of a formula F:

$$\langle U\ F\rangle \rightarrow M$$

However, for Red languages, the identity operator has this same property:

$$\langle \phi\ \underline{X}\rangle \rightarrow \ddot{\rho}\underline{X}$$

if $\underline{X}$ is a meaningful formula. Of course this does not help explain the interpretation of $\underline{X}$. The problem is that any non-constant $\underline{X}$ denotes its meaning. Hence one cannot operate, for example, on an application $\langle \underline{X}\ \underline{Y}\rangle$ to show how composition rules work, because $\langle \underline{X}\ \underline{Y}\rangle$ will have been reduced to a constant before any operation can be performed on it.

Thus, to be meaningful, a universal operator for Red/2 must operate on a constant item, $\omega\underline{X}$, which encodes a non-constant item $\underline{X}$. We also want $\omega\underline{C}=\underline{C}$ for constants $\underline{C}$. For Red/2, we shall restrict the definition of ω to items which do not contain a pair with the prefix A. If $\underline{X}$ is such an item, then $\omega\underline{X}$ is obtained by replacing every application

$$\langle \underline{Y}\ \underline{Z}\rangle$$

within $\underline{X}$ by the prefixed pair

$$A(\underline{Y}\ \underline{Z})$$

We shall call such a pair a pseudo-application. Thus ω

replaces all applications by pseudo-applications and produces the pseudo-item $\omega\underline{X}$ of pseudo-Red/2, which is a constant of Red/2. We can now define the operator, *MEANING*, in Red/2 such that, if $\underline{X}$ is any Red/2 item without any α-prefixed pairs, then the meaning of $\underline{X}$, if any, is given by:

$$\ddot{\rho}\underline{X} = \langle MEANING\ \omega\underline{X}\rangle$$

The meaning of $\underline{X}$ in Red/2 is the same as the *MEANING* of $\omega\underline{X}$ in pseudo-Red/2. Note that *MEANING* is not a proper universal operator in the sense given above, since it is an operator of Red/2 which does not map Red/2 items into their meaning directly, as required, but only after the application of ω; and ω is not expressible in Red/2.

Variable-free programming techniques

The definitions of example operators given in the section on Red/1 show how the primitives of Red/1 can be used to construct a variety of composite operators. In defining these operators, no systematic programming methods are much in evidence. But now that we have partially surmounted the primitive state of Red/1 by constructing these new operators, we can focus on developing better organized and more readable programming methods in Red/2. In doing this we shall take advantage of Red/2 self-referencing definitions. The techniques illustrated below are to be regarded as early efforts in a new field. It is to be expected that better methods will be found after more time has elapsed.

We shall illustrate variable-free programming techniques [which rely heavily on the operators *2F* and *CONDARG*] by defining a number of operators leading to the definition of the operator *MEANING*. For each defined operator to follow, three descriptions are given:

1) A LISP-like definition with variables and conditional expressions. The mark "==" is used in this definition to indicate its informal nature.

2) The corresponding Red/2 definition for use in ∇ [written in the Red/1 format with an equal sign and using the usual shorthand]. We take a few syntactic liberties with arrangement and with the use of blanks.

3) A description-by-cases of the effect of the defined operator. Every such description has an implicit first case: □→□, and an implicit last case: *X*→□.

Preliminary definitions

Prefix of

PREF.*X* == ∘1.*UNPREFIX*.*X*

PREF = (∘1 *UNPREFIX*)

PREF | (*X*1...*X*n) → ⍬; *S*(*X*1...*X*n) → *S*

Unprefixed set

```
USET.X == ATOM.X → F
          EQAT.[∅ PREF.X] → T
          T → F
USET = (CONDARG ATOM → C:F
            EQAT:[2F C:∅ PREF] → C:T
            C:T → C:F)
USET | A → F;  (X1...Xn) → T;  X → F
```

Pseudo application

```
PSAP.X == ATOM.X → F
          EQ∅.TL.X → F
          EQ∅.TL.TL.X → EQAT.(A PREF.X)
          T → F
PSAP = (CONDARG
            ATOM → C:F   (EQ∅ TL) → C:F
            (EQ∅ TL TL) → (EQAT ADJ:A PREF)
            C:T → C:F)
PSAP | A(X Y) → T;  X → F
```

Pseudo set

```
PSET.X == AND.[NOT.ATOM.X  NOT.PSAP.X]
PSET = AND:[2F (NOT ATOM) (NOT PSAP)]
PSET | A → F;  A(X Y) → F;  X → T
```

Prefix with A

```
PREFA.X == PREFIX.[A X]
PREFA = (PREFIX ADJ:A)
PREFA | S(X1...Xn) → A(X1...Xn)
```

Equals

```
EQ.(X Y) == ATOM.X → EQAT.(X Y)
            ATOM.Y → EQAT.(X Y)
            NOT.EQAT.[PREF.X  PREF.Y] → F
            T → AND.[EQ.(∘1.X  ∘1.Y) EQ.(TL.X  TL.Y)]
EQ = (CONDARG
        (ATOM ∘1) → EQAT
        (ATOM ∘2) → EQAT
        (NOT EQAT [2F (PREF ∘1) (PREF ∘2)]) → C:F
        C:T → AND:[2F EQ:⊂2F (∘1 ∘1) (∘1 ∘2)⊃
                EQ:⊂2F (TL ∘1) (TL ∘2)⊃])
EQ | (X Y) → □   when X or Y contains □
     (X X) → T;  (X Y) → F
```

Pseudo constant

```
PSCON.X == ATOM.X → T
           PSET:X → AND.[PSCON.∘1.X  PSCON.TL.X]
           PSAP.X → F
PSCON = (CONDARG
            ATOM → C:T
            PSET → AND:[2F (PSCON ∘1) (PSCON TL)]
            PSAP → C:F)
PSCON | A → T
        X → F   when X contains a pseudo-application
        X → T
```

The semantic operator for Red/2 and auxilliary operators

Meaning

```
MEANING.X == PSCON.X → X
             T → MEANING.RHO.X
MEANING = (CONDARG PSCON→φ C:T→[MEANING RHO])
```

MEANING | $\underline{X} \rightarrow \underline{Y}$ where $\underline{X}=\omega\underline{Z}$ and $\underline{Y}=\ddot{\rho}\underline{Z}$

Thus *MEANING* maps pseudo-items $\underline{X}$ $[=\omega\underline{Z}]$ into their meaning, $\underline{Y}$, which is the same as the meaning of $\underline{Z}$. If $\underline{X}$ corresponds to a meaningless $\underline{Z}$, then *MEANING*.$\underline{X}$ does not terminate.

Rho

```
RHO.X == ATOM.X → X
         PSET.X →
             PSCON.∘1.X → UNION.[∘1.X  RHO.TL.X]
             C:T → UNION.[RHO.∘1.X  TL.X]
         PSAP.X → DELTA.X
RHO = (CONDARG
          ATOM → φ
          PSET → [CONDARG
                     (PSCON ∘1) → UNION:[2F ∘1 (RHO TL)]
                     C:T → UNION:[2F (RHO ∘1) TL]]
          PSAP → DELTA)
```

RHO | $\underline{X} \rightarrow \underline{Y}$ where $\underline{Y}=\omega\rho\underline{Z}$ when $\underline{X} = \omega\underline{Z}$

Delta

```
DELTA.A(X Y) == USET.X →
                    EQ.[PSI.∘1.X  ∘1.X] →
                          A(∘1.X  A(TL.X  Y))
                    T → A(PSI.∘1.X  (X Y))
                  NOT.EQ.[PSI.X  X] → A(PSI.X  (X Y))
                  T → EPSILON.A(X Y)
DELTA =
   (CONDARG
      (USET ∘1) →
         [CONDARG
            ⊂EQ (2F PSI ∅) ∘1 ∘1⊃ →
                              PREFA:⊂2F (∘1 ∘1) FST:TL⊃
            C:T → PREFA:⊂2F (PSI ∘1 ∘1) (∘2 UNPREFIX)⊃]
      (NOT EQ [2F PSI ∅] ∘1) →
            PREFA:(2F [PSI ∘1] [∘2 UNPREFIX])
      C:T → EPSILON
DELTA | A((X1...Xn) Y) → A(X1  A(W Y))
            when PSI.X1 = X1 and W = ∅ or (X2...Xn)
        A((X1...Xn) Y) → A(PSI.X1  ((X1...Xn) Y))
            when PSI.X1 ≠ X1
        A(X Y) → A(PSI.X  (X Y))
            when PSI.X ≠ X
        A(X Y) → EPSILON.A(X Y)
```

Psi

```
PSI = a primitive of Red/2
PSI | *(X1...Xn) → (X1...Xn)
      S → X   when  *(S X) ∈ ∇
      X → X
```

Epsilon

EPSILON.A(*X Y*) == <*DEF.X Y*>

EPSILON = (*APP FST:DEF*)

EPSILON | *A*(*S Y*) → <*Z Y*> when (*S Z*) ∈ ∇

A(*X Y*) → <*X Y*> otherwise

Parsing

A complete self-description of Red/2 would, in addition to the specification of *MEANING*, show how to *PARSE* an item *X* enclosed in string-quotes and produce either the corresponding pseudo-item *ωX* or the output: "not well formed". Thus, *PARSE* would transform the string

⊧(*X* <*Y Z*>)⊣

into the pseudo-item

(*X A*(*Y Z*))

ready for interpretation by *MEANING*.

Controlling order of execution

The device of pseudo-applications can be used along with ordinary applications and with various operators to "execute" them to control the order of evaluation of expressions, since applications will always get done before pseudo-applications. They can also be used to permit alternating between evaluation and substitution of variables (see Manna et al, [5]: computation rules).

RED/3 LANGUAGES

Introduction

In this section our purpose is to look briefly at the potential for "high level" Red languages, those with sophisticated primitives rather than the primitive primitives of Red/1. It is much too early in the development of Red languages to propose a definite set of high level primitives with any confidence that it is complete and well-coordinated. We hope to show, however, that the framework provided by Red languages is a rather good one in which to place such high level primitives.

We shall be much less precise than heretofore in specifying Red/3 languages, since our purpose is only to sketch some possibilities. We shall specify only some of their primitives and assume others, such as "×", which have not been specified.

A number of the operators given below were suggested by operations in APL [6], a language full of good ideas, as is the practice of using one-character or short symbols for common modifiers. Indeed, some of the motivation for Red languages comes from a desire to improve on APL's good ability to deal with compound objects as a whole, and its use of operators which combine with others [e.g.: +/, "reduction"; and +.×, "inner product"]. Roughly speaking, APL has only two "modifiers", "/" and "."; it cannot define others and these combine only with primitives, but not with defined operators.

APL also has "EE", elementwise-extend application
e.g. 1 2 3+4 5 6 ≡ 5 7 9

Syntax and semantics

The syntax and semantics of Red/3 languages are the same as for Red/2 languages except for the set of primitives. Thus a Red/3 language is determined by its set of basic primitives and its defining set ∇. Red/3 languages will employ numbers represented in various ways by symbols composed of digits, decimal points, and perhaps other marks.

All that follows should be regarded as merely a sketch of possible Red/3 languages. There may be ambiguities or inaccuracies in some details.

Sampler of Red/3 operators

Name	Symbol

operand → result

Transpose *TRANS*

$[(X11 \ldots X1m) \ldots (Xn1 \ldots Xnm)] \rightarrow [(X11 \ldots Xn1) \ldots (X1m \ldots Xnm)]$

Distribute from left *DISTL*

$(X\ (Y1 \ldots Yn)) \rightarrow ((X\ Y1) \ldots (X\ Yn))$

Distribute from right *DISTR*

$((X1 \ldots Xn)\ Y) \rightarrow ((X1\ Y) \ldots (Xn\ Y))$

Union left *UNL*

$((X1 \ldots Xn)\ Y1 \ldots Ym) \rightarrow (X1 \ldots Xn\ Y1 \ldots Ym)$

Union *UN*

$((X11 \ldots X1m) \ldots (Xn1 \ldots Xnm)) \rightarrow (X11 \ldots X1m \ldots\ldots Xn1 \ldots Xnm)$

Cumulative sequences *CUM*

$(X1 \ldots Xn) \rightarrow ((X1)\ (X1\ X2) \ldots (X1 \ldots Xn))$

Plus +

(i j) → i+j

Sampler of Red/3 modifiers

Name Symbol

operator | operand → result

Apply left αL

αL:X | (Y1...Yn) → (X.Y1...Yn)

Apply right αR

αR:X | (Y1...Yn) → (Y1...X.Yn)

Apply all α

α:X | (Y1...Yn) → (X.Y1...X.Yn)

Apply ith αI

(αI X i) | (Y1...Yn) → (Y1...X.Yi...Yn)

(αI X (i)) | (Y1...Yn) → (Y1...X.Yi...Yn)

(αI X (i...j k)) | (Y1...Yn) → (Y1...(αI X (i...j)).Yk...Yn)

Example

(αI X (1 2)).((A B) (C D)) → ((A B) (X.C D))

Adjoin left ADJL

ADJL:X | (Y1...Yn) → (X Y1...Yn)

Select ⊥

⊥:i | (Y1...Yn) → Yi

⊥:(i) | (Y1...Yn) → Yi

⊥:(i...j k) | (Y1...Yn) → ⊥:(i...j).Yk

Example

⊥:(1 2).(A (B C)) → ⊥:(1).(B C) → B

Insert left ∧L

∧L:X | (Y1...Yn) → (X.(Y1 Y2) Y3...Yn)

Insert ^

^:X | (Y1...Yn) → X.(Y1 X.(Y2...X.(Yn-1 Yn)...))

Example

^:+.(1 2 3) → 6

Up ↑

(↑ X1...Xn) | (Y1...Ym) → (X1...Xn Y1).(Y2...Ym)

Down ↓

(↓ X1...Xn) `| (Y1...Ym) → (X1...Xn-1).(Xn Y1...Ym)

Binary operators with a constant

If ○ is the symbol for an operator which requires a pair for its operand, then ○↓ [the symbol with the suffix ↓] is the symbol for a modifier such that

○↓:X.Y → ○.(X Y)

For example,

+↓:1.X → +.(1 X) → 1+X

Operators with function-determined operands

If ○ is an operator which takes operands (X1...Xn), then ○∘ is a modifier such that

(○∘ X1...Xn).Y → ○.(X1.Y...Xn.Y)

Example

(+∘ ⊥:3 ⊥:1).(5 6 7 8) → +.(7 5) → 12

Replace *REPL*

(*REPL* X Y) | Z → (α*I* *C*:<Y Z> <X Z>).Z

Thus X gives the "address" of a subitem of Z to be replaced by Y.Z.

Example

Suppose we wish to replace *Ai*+1 in (*A*1...*A*n) by (*B* *B*) in

Z = (*B* i (*A*1...*A*n))

Then

```
(REPL (++:1  ⍳:2)  (DBL  ⍳:1)).Z
      → (B i (A1...Ai  (B B)  Ai+2 ...An))
```

While *WHILE*

(WHILE X; Y) | Z → [(WHILE X; Y) Y].Z if X.Z → T

Z if X.Z → F

Thus the effect of *(WHILE X; Y)* is to apply *Y* repeatedly to the operand as long as the condition given by *X* is true.

Example

Suppose we wish to sum the initial segment of *(A1...An)* as long as the sum is less than or equal to the element following the segment. If

```
R = (WHILE  [≤∘  ⍳:1  ⍳:2]  ∧L:+)
```

then

```
R.(1 2 4 6)
    → R.(3 4 6)
    → R.(7 6)
    → (7 6)
```

Examples of Red/3 defined operators

Inner product *IP*

```
IP = (∧:+  ∝:×  TRANS)
(∧:+  ∝:×  TRANS).((2 3) (4 5))
    → (∧:+  ∝:×).((2 4) (3 5))
    → ∧:+.(8 15)
    → 23
```

Matrix multiply *MM*

```
MM = (α:α:IP  α:DISTL  DISTR  αR:TRANS)

MM.[[(1 2) (3 4)] Y]

      where  Y = [(5 6) (7 8)]

   → (α:α:IP α:DISTL DISTR).[[(1 2) (3 4)] W]

      where  W = [(5 7) (6 8)]

   → (α:α:IP α:DISTL).[[(1 2) W] [(3 4) W]]

   → α:α:IP.[DISTL.[(1 2) W] DISTL.[(3 4) W]]

   → α:α:IP.[[⊂(1 2) (5 7)⊃ ⊂(1 2) (6 8)⊃]
             [⊂(3 4) (5 7)⊃ ⊂(3 4) (6 8)⊃]]

   → [[19 22] [43 50]]
```

Number of doubly restricted partitions *DR/PART*

We give here the key part of a procedure for calculating

$$P = ((P11)\ (P21\ P22)\ldots(Pn1\ldots Pnn))$$

from (n k), where Pi,j is the number of distinct sets of integers m, such that $k \le m \le j$, whose sum is i. The procedure given here is a rearrangement of CACM Algorithm #373 [7]. Algorithm 373 computes each Pij by adding $Pi,j-1$ and $Pi-j,r$ where r=i-j when i-j<j and r=j otherwise; and i varies from k to n and j varies from k to i.

The initial part of our procedure, which we shall not give in detail, produces the following from (n k):

[n (Y Y)]

where Y = (Y1...Yk)

and Y1 = (1); Y2 = (0 0); Y3 = (0 0 0); etc.

Yk = (0...0)

This is the input to the process we give below which repeats the following iteration n times. The first iteration begins with ($\underline{Y}$ $\underline{Y}$). Each subsequent iteration begins with ($\underline{Z}$ $\underline{Y}$) and produces ($\underline{Z}'$ $\underline{Y}'$) as follows:

1 Form $\underline{Z}''$ from $\underline{Z}$ by dropping the first member from each member-set of $\underline{Z}$ which has two or more members. E.g.,

$\underline{Z}$ = ((1) (2 3) (4 5 6)) → $\underline{Z}''$ = ((1) (3) (5 6))

2 From $\underline{Z}''$ form the set $\underline{T}$ of all first members of the sets of $\underline{Z}''$. E.g.,

$\underline{Z}''$ = ((1) (2 3) (4 5 6)) → $\underline{T}$ = (1 2 4)

3 Reverse the elements of $\underline{T}$, adjoin a zero on the left and then replace each element, starting on the right, by the sum of all the elements to its left including itself, yielding $\underline{U}$. E.g.,

$\underline{T}$ = (1 2 3 4) → $\underline{U}$ = (0 4 7 9 10)

4 Adjoin $\underline{U}$ as the rightmost set of $\underline{Z}''$ and $\underline{Y}$, giving $\underline{Z}'$ and $\underline{Y}'$. E.g.,

($\underline{Z}$ $\underline{Y}$) = ([(1) (2) (3 4) (5 6 7)] [(8) (9 10)])

→ ($\underline{Z}'$ $\underline{Y}'$) = ([(1) (2) (4) (6 7) ~~(0 9 11 12)~~ (0 6 10 12 13)]

[(8) (9 10) ~~(0 9 11 12)~~ (0 6 10 12 13)])

The above iteration [1 through 4] is repeated n times, yielding ($\underline{Z}$ $\underline{Y}$). The result P is $\underline{Y}$ with its first set dropped and the first member of each non-solitary member set dropped.

Drop the first member of a non-solitary set

*DROP*1 = (*CONDARG* (*EQ*∅ *TL*)→∅ *C*:*T*→*TL*)

*DROP*1 | $\underline{A}$ → □; ($\underline{X}$) → ($\underline{X}$);

($\underline{X}$1...$\underline{X}$n) → ($\underline{X}$2...$\underline{X}$n)

Form (Z" Y) from (Z Y)

```
ZY/ZPPY = αL:α:DROP1
ZY/ZPPY  |  ([Z1...Zn] Y) → ([Z1'...Zn'] Y)
        where  Zi' = DROP1.Zi
```

Form U from Z"

```
ZPP/U = (α:∧:+  CUM  ADJL:0  INVERT)
ZPP/U  |  (Z1...Zn) → (0, Zn, Zn+Zn-1,...,Zn+...+Z1)
```

Adjoin U as the last member of Z" and Y

```
ADJ/U/ZY = (α:UNL DISTR)
ADJ/U/ZY  |  ((Z" Y) U) → (Z' Y')
```

One iteration

```
ITER = (ADJ/U/ZY  [2F ϕ (ZPP/U ⊥:1)] ZY/ZPPY)
ITER  |  (Z Y) → (Z' Y')
```

Number of doubly restricted partitions

```
DR/PART = (α:DROP1 TL ⊥:2 REP:ITER)
DR/PART  |  [n (Y Y)] → P
```

OTHER RED-LIKE LANGUAGES; COMPARISONS WITH OTHER SYSTEMS

A Red-like complete secondary language

Red items such as

$$(<X1\ Y1>...<Xn\ Yn>)$$

can be reduced in parallel because the lack of variables prevents any possibility of interaction or communication between the non-overlapping applications in the item. Thus the order of reductions is to a large extent immaterial, provided only that one cannot operate on an application in a way which requires access to its elements or to properties of its value.

Red-like languages with variables can be constructed which make possible communication between disjoint applications. Such languages can then be used to study problems of concurrent program execution in which the result may depend on the order of performing reductions.

A simple language of the above sort is obtained by adjoining to Red/1 two new primitive operators, *FETCH* and *STORE*, which refer to a single implicit variable ω. The effect of *FETCH* is

$$\epsilon<FETCH\ X> = Y$$

where Y is the current value of ω. And

$$\epsilon<STORE\ X> = X$$

and the value of ω becomes X.

This new language is neither a reduction language nor

primary. To assign a meaning to an item, a state language is needed in which a state is

$$(\underline{X}\ \omega)$$

where $\underline{X}$ is an item and ω is our single variable. Normally a state transition is simply

$$(\underline{X}\ \omega) \rightarrow (\rho\underline{X}\ \omega)$$

unless the transition $\rho\underline{X}$ involves a primitive application of *FETCH* or *STORE*. In that case the state transition is

$$(\underline{X}\ \omega) \rightarrow (\underline{X}'\ \omega')$$

If the transition corresponds to an application of *FETCH*, $\underline{X}'$ will depend on ω; and for *STORE*, ω' will be changed. The details are as indicated by the descriptions of *FETCH* and *STORE*. This completes our sketch of the state language of the complete secondary language we shall call Red/1/S. It remains only to give α_i and α_t:

$$\alpha_i(\underline{X}\ \square) = \underline{X}$$

$$\alpha_t(\underline{Y}\ \omega) = \underline{Y}$$

α_i is defined for any item $\underline{X}$; α_t is defined only when $\underline{Y}$ is a constant.

We can now say that Red/1/S is the complete secondary language

$$(I, L, \alpha_i, \alpha_t)$$

where I is the set of Red/1 items, L is the state language described above, and α_i and α_t are as given. The meaning of an item $\underline{X}$ is found by performing transitions on states beginning with $(\underline{X}\ \square)$ until a state $(\underline{X}'\ \omega)$ is reached with $\underline{X}'$ constant. $\underline{X}'$ is then the meaning of $\underline{X}$.

allowable variation

The only ~~ambiguity~~ in the state transition function of Red/1/S is the choice of which subitem to reduce at each step; but the result may now depend on that choice. It is of interest then to determine which items X always have the same meaning no matter what choices are made in the reduction sequence. This corresponds to determining what processes in an asynchronous parallel computer will always give the same result.

Since the value of ω can be regarded as the store of a computer, Red/1/S should be adequate to model any computer. The variable ω can be restricted to be a set of pairs, each denoting

(address, contents)

and its use can be confined to defined operators which serve as fetch and store operators for various addressing systems.

Comparison of combinators and Red operators

It appears that the basic difference between Curry's combinators [3] and Red operators is that the former are based at a higher level of abstraction. Thus the operator *REV* reverses the two elements of a pair whereas the combinator C, when applied to a function of two arguments, produces another function which is the original with its arguments reversed. Apparently the lower level of Red operators suffices because of the availability of meta composition and hence definable modifiers and because ordered sets are objects and hence every operator can be regarded as unary. A brief comparison of some basic

combinators and some primitive and defined Red/1 operators gives the flavor of the differences:

combinator	operator
Ix ≥ x	$\phi.\underline{X} \rightarrow \underline{X}$
Cfxy ≥ fyx	$(\underline{F}\ REV).(\underline{X}\ \underline{Y}) \rightarrow \underline{F}.(\underline{Y}\ \underline{X})$
Wfx ≥ fxx	$(\underline{F}\ DBL).\underline{X} \rightarrow \underline{F}.(\underline{X}\ \underline{X})$
Bfgx ≥ f(gx)	$(\underline{F}\ \underline{G}).\underline{X} \rightarrow \langle \underline{F}\ \langle \underline{G}\ \underline{X} \rangle \rangle$
Kxy ≥ x	$(C\ \underline{X}).\underline{Y} \rightarrow \underline{X}$

Note that association is to the left in combinatory expressions, thus Cfxy = ((Cf)x)y, and juxtaposition denotes application of the left object to the right object.

Comparison of AE's and Red languages

Since Curry's system of combinators is, roughly speaking, equivalent to Church's system of lambda-conversion, Landin's AE's [2], which are based on Church's system, have a number of characteristics in common with combinatory expressions. To represent a function as an AE, a higher level of abstraction is often needed than to represent the function as a Red operator. Thus definitions of self-invoking AE functions require the use of the function Y which takes functions as arguments and yields functions as values.

It is a simple matter to define Red self-invoking operators in a direct, procedural way without employing concepts at the level of the Y function. And the use of meta composition makes it possible to define *CONDARG* in a way that properly evaluates conditional expressions and yields a

This is because of 'call-by-value' order of evaluation

direct result. The AE formulation, in contrast, is at a higher level: the result of a conditional expression is a function [of no arguments] which must then be applied to the null list of arguments to obtain the desired result.

Finally, a set of Red operators is essentially a simple procedural description which says, in effect, do this, then this, and so on. Such a description can be easily read, after some practice, even when it is long and complex. In contrast, reading an AE is complicated by an intermingling of procedural descriptions ["combinations"] and procedures for substituting arguments for variables. The latter kind of process can be a very complex one with its intrinsic problems concerning confusion and collision of variables. Although the SECD machine can resolve all these questions, they are nonetheless not simple ones for the reader to resolve.

Relation to other work

Basically the work in this paper is a descendent of a long line of efforts inspired by McCarthy [8]. In first demonstrating the existence of an elegant, powerful, and concise programming language whose complete syntax and semantics could be briefly and formally described, McCarthy established a new field within programming. Following his example, the works of logicians such as Church, Curry, Kleene, Rosser, and others, were further studied and the new field produced a large number of other languages and ideas, a few of which, notably Landin's [2], have been discussed

herein. The syntax and some of the semantics of Red languages have clearly drawn upon this work.

The variable-free approach to programming clearly owes much to LISP. And, as noted earlier, some of the notions and operators in the section on Red/3 are influenced by the APL language [6].

But so far as I know, the elements of this paper which may be novel, at least to some degree, are: (a) the exhibition of a simple primary programming language; (b) the axioms for the various classes of languages, particularly reduction languages and parenthetic languages; (c) the proof that the former class includes the latter [even though this proof serves somewhat the same ends as the Church-Rosser theorem [10]]; (d) some of the variable-free programming methods used in the examples; and finally, (e) the concept of meta composition and the consequent notion of modifiers.

Acknowledgements

I am indebted to D. Bjørner for writing a program to reduce Red items which was used to test some of the operators in this paper. This work helped to clarify a few mistakes in the early development of Red/1, and Dr. Bjørner contributed some helpful suggestions. I am also grateful to W. H. Burge for some valuable discussions and to Howard Smith for some modifications to the APL Text Editor used in preparing this paper.

References

1. Davis, M. Computability and Unsolvability. McGraw-Hill, New York, 1958.

2. Landin, P. J. The mechanical evaluation of expressions. The Computer Journal, (Jan. 1964) 308-320

3. Curry, H. B., and Feys, R. Combinatory Logic. Vol. 1. North-Holland, Amsterdam, 1968

4. Petznick, G. W., Jr. Combinatory Programming. Ph.D. Thesis, Univ. of Wisconsin, Computer Science, 1970

5. Manna, Z., Ness, S., Vuillemin, J. Inductive methods for proving properties of programs. Proc. ACM Conf. on Proving Assertions about Programs (Jan. 1972) 27-50

6. Falkoff, A. D., and Iverson, K. E. APL\360 User's Manual. IBM, 1968

7. White, J. S. Algorithm 373, Number of doubly restricted partitions. Comm. ACM 13,2 (Feb. 1970), 120

8. McCarthy, J. Recursive functions of symbolic expressions and their computation by machine. Comm. ACM, 3,4 (April 1960) 184-195

9. Church, A. The calculi of lambda-conversion. Annals of Mathematics Studies, No. 6, Princeton Univ. Press, Princeton, 1941

10. Church, A., and Rosser, J. B. Some properties of λ-conversion. Trans. Amer. Math. Soc. 39, (1936), 472-482