Paul McJones
[redacted]

IBM Research

Programming Language Semantics and Closed Applicative Languages

John Backus

July 5, 1973

RJ 1245

Yorktown Heights, New York

San Jose, California

Zurich, Switzerland

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

PROGRAMMING LANGUAGE SEMANTICS AND CLOSED APPLICATIVE LANGUAGES

John Backus

IBM Research Laboratory
San Jose, California

ABSTRACT: A hierarchy of language-classes is axiomatically defined, plus the notion of the "realization" of a language. The most general class is that of "programming language", the most restricted, "closed applicative language". The latter class has a number of attractive abstract properties which are absent in most languages. This new class employs a simple but radically different interpretation of "applications" and makes a precise distinction between "meaning of" and "function represented by" an expression.

The simplest examples of closed applicative languages are called Red [Reduction] languages. These emphasize basic semantics rather than the set of primitive operators. The Red language with the simplest set of operators comprises the Turing machine of programming languages, yet a Red language with the same basic semantics but a sophisticated set of operators is a powerful high-level language. Red languages use no variables or labels, no GO TO statements, and no built-in comprehension of recursively defined functions. [These Red languages are simpler than those of an earlier paper [4].]

Another set of closed applicative languages is that of λ-Red languages. These resemble (continued next page)

RJ 1245 (#19774)
July 5, 1973
Computer Sciences

ABSTRACT, CONTINUED

the λ-calculus but have many important differences, for example: (1) every innermost application can be evaluated at once, and (2) bound variables need no changing [no I-conversion].

The axioms defining Red and λ-Red languages are simple enough that it may be possible to treat them with a mathematical rigor which has heretofore been inapplicable to a complete programming language.

ERRATA

PROGRAMMING LANGUAGE SEMANTICS AND CLOSED APPLICATIVE LANGUAGES

July 17, 1973

Page 9, line 4 from below:

...kn ε $[E^n \to E]$. := ...kn: Skn $\to$ E, where Skn$\subseteq E^n$.

Page 11, line 3:

Let A E. := Let A $\subseteq$ E.

Page 18, line 7 from below:

= ([ρDBL]A,B) := = ($\mu \cdot$[ρDBL]A,B)

Page 18, line 6 from below:

by def ρDBL := by def ρDBL, AL3, AL1, & CL3

Page 21, rules τ2 and τ3:

While the transition function τ is correct as it stands, it can be simplified by removing "δ" from the result of the transition in subrules τ2 and τ3 only. Thus, for example, the last expression of τ2 is: <c1,(c,d)>. This simplification is assumed in the two examples as corrected.

Page 21, line 4 from below. Insert before "Since all...":

[Provided the expansion of ρCN does not involve ρF.]

Page 22, line 11 from below:

<(Ø,APP)((NT),X)> := <NT,((NT),X)> $\to$ <(Ø,APP),((NT),X)>

Page 23, line 5:

...those f that depend... := ...those a that depend...

Page 23, line 11 from below:

...of a meaningful... := ...within a meaningful...

Page 27, line 2:

...<c,A>. := ...<c,A> when ϕc=Ø.

Page 29, lines 11 and 12 from below:

...introduction: non-idempotency... :=
...introduction. Non-idempotency...

ERRATA

PROGRAMMING LANGUAGE SEMANTICS AND CLOSED APPLICATIVE LANGUAGES

August 8, 1973

Page 3, line 1:

...yield... := ...yields...

Page 15, line 8 from below:

"undefined" := "unspecified", as distinguished from "undefined", "non-terminating", or "having no meaning"; $\perp$ is a valid meaning.

Page 16, line 14:

undefined := unspecified

Page 22, line 2 from below:

one transition := two transitions

Page 23, line 9 from below:

$((\lambda x M)N) := ((\lambda y(\lambda x M))N)$

Page 23, line 8 from below:

$\langle(\lambda x M),N\rangle := \langle(\lambda y(\lambda x M)),N\rangle$

All pages:

Every occurrence of the proper inclusion symbol, $\subset$, should be replaced by the inclusion symbol $\subseteq$.

See also the Errata sheet dated July 17, 1973.

965-IBM-03

INTRODUCTION

This paper proposes axioms to define a sequence of language-classes; the most general is that of "programming language", the most restricted has some simple and attractive properties. Here "language" is used in its traditional sense as referring to a set of interpreted expressions. We are concerned with the syntax of an expression only to the degree needed to relate its structure to its "meaning". A clear distinction is drawn between a "language" and the many possible "realizations" of that language.

This introduction comprises a survey and opinionated discussion of the contents of the paper, therefore the reader who wishes to get on with the technical exposition can skip to the next section.

Complete languages

The most-general-but-one class of languages in our hierarchy is that of "complete" languages. These are self-contained languages which do not depend on another for their semantics. The description of the semantics of any language requires the formulation of a complete language or something very similar.

Closed applicative languages

The most restrictive set of axioms defines "closed applicative languages". In these languages an expression e has two distinct notions associated with it: (a) the "meaning" or "value" of e [which is always an expression when e has a meaning], and (b) the function "represented" by e. These languages have the following interrelated properties:

(1) Idempotency of meaning. "Constant" expressions are those which are their own meaning, the fixedpoints of the function called "meaning". The meaning of any expression, if it exists, is a

constant. Thus if f is the meaning of the expression e, then f is an expression and is its own meaning. In languages such as McCarthy's LISP [3] where meaning is not idempotent, we have chains of meanings, e.g.: (QUOTE e) → e → f...etc.. Could define geval[e]=list[QUOTE; eval[e]].

Note: λ-calculus, ISWIM, PAL, etc. are idempotent.

(2) The "anti-quote" property. Most languages are "quote" languages: an expression never denotes itself, but a specially marked expression denotes the unmarked expression: "e" denotes e, and e denotes something else. In a closed applicative language, an expression without "applications" as subexpressions always denotes itself, i.e., is its own meaning, hence a constant. An "application", e.g., <f,g>, denotes the result of applying the function represented by the expression f to the expression g. Finding the meaning of an expression e is simply a matter of replacing each innermost application in e by the result it denotes.

Lisp: (QUOTE e)
Snobol4: *e
Euler: 'e'

In a CAL, it is impossible to talk about an expression with applications!

Macro processors, format languages, & pattern matching languages are sometimes "antiquote" languages.

(3) Non-extensionality. In an extensional system, if g and h denote the same function, then f(g) and f(h) denote the same thing. In a closed applicative language one can have expressions g and h which have different meanings but represent the same function and, for some f, <f,g> and <f,h> can have different meanings and represent different functions, since the expressions g and h are involved, not [necessarily] the functions they represent.

(4) Single-type functions. The functions represented by expressions are all of the same type: they all map constant expressions into expressions. Thus no infinite hierarchy of function types is required as it is in Scott's model of the lambda-calculus [1]. But not all constants make sense as functions, and not all functions produce outputs which make sense as functions, and so on.

(5) The extended Church-Rosser property: (a) Every terminating

sequence of reductions on an expression yields the same meaning for it, and (b) if an expression has a meaning, then every sequence of reductions on it terminates. The λ-calculus of Church [2] has properties (a) and (b), but the λ-K-calculus and LISP do not have property (b). Consider $\langle 1, (T, \langle (WHILE, \phi, \phi), T\rangle)\rangle$. It would be reasonable to say its meaning is T, but (b) would be violated. Thus the "innermost application" rule!

(6) The reduction property: (a) If an expression e has a meaning, then so do all the subexpressions of e, and (b) if any subexpression of e is replaced in e by its meaning, the meaning of e is unchanged.

One can construct more general systems in which (a) fails but (b) still holds.

Property (b) is built in to CAL's. Question: Is it desirable?

Two examples of closed applicative languages are specified: Red languages ["Red" for "Reduction"] and λ-Red languages. Emphasis in these languages is on the language framework, the fundamental semantics, rather than on the particular primitive operations provided. One of these languages corresponds to every set of primitive operators, the rest of the semantics for both types being fixed.

Red languages

A Red language with a simple set of primitives comprises the Turing machine of programming languages: its syntax and semantics are extremely simple, yet it can specify any computable function. On the other hand, a Red language with the same basic semantics but a sophisticated set of primitive operators comprises a powerful high-level programming language.

The basic semantics of Red languages, beyond those of all closed applicative languages, involve only one notion: functional composition. Two kinds are provided, the classical composition of functions and a new form called meta composition. The latter makes possible the definition of functions with parameters and those defined

by iteration or recursion. Red languages employ no variables or labels, no GO TO statements, no lambda-expressions, and have no built-in mechanism for evaluating recursively defined functions. The Red languages described in this paper are different from, and simpler than those given in an earlier report [4].

λ-Red languages

λ-Red languages are closed applicative languages which resemble the λ-calculus. They differ from the λ-calculus in properties (2), (3) and (4) of the six properties listed above for closed applicative languages and they differ from LISP and Landin's "Applicative Expressions" or AEs [5] in all six. In addition, λ-Red languages have the following properties which are not shared by any of the three other languages:

(1) Every innermost application can be immediately evaluated.

(2) There is no need for changing bound variables when evaluating an expression.

The differences between λ-Red languages and LISP and AEs give the former simpler semantics and hence simpler realizations.

Red programming

Although space does not allow much, an appendix gives primitive operators and some programming examples for Red languages.

LANGUAGES

A language [or programming language] L comprises:

L1) A set of expressions, E.

L2) A domain of discourse, D.

L3) A semantic relation, $\sigma \subseteq E \times D$

When $(e,d) \in \sigma$, we say that $d \in D$ is a consequent of the expression $e \in E$. If σ is a function, i.e., there is only one consequent d for any e, then we say that d is the meaning of e.

Example: Algol

Let E be the set of Algol programs. Call s an e-initial state if s is any state of the "Algol Machine" in which the program e is about to be executed. Let μs denote the terminal state, if any, which results from the state s. Let D be the set of pairs or "computations" $(s,\mu s)$ for states s having a terminal state μs. Let $(e,(s,\mu s)) \in \sigma$ whenever s is an e-initial state. Then $L = (E,D,\sigma)$ is one possible formulation of Algol as a language in our sense. The semantic relation indicates, for any program e, what states s are e-initial states and terminate in μs. Thus the "consequences" of a program are the terminating "computations" it engenders.

Discussion

Although it is a popular idea, we do not try to assign a function as the "meaning" of a program. In the case of Algol, for example, the domain of such functions must be the states of the Algol machine and these involve one too much in notions of implementation rather than of meaning. Furthermore, we do not want expressions like "3+4" to be "meaningless" or to have artificial and useless "meanings" simply because they do not denote functions. Thus we use "meaning" in the sense of "value"; but we shall also pay careful attention later on to associate functions with expressions in an orderly way.

The real problem in describing the semantics of Algol as above is that of describing "states" and the function μ which carries a state s into the corresponding terminal state μs. This is a special case of formulating what we shall call a "complete language". Complete

languages thus play a central role in specifying the semantics of any language. Later we shall encounter several examples of complete languages which are independent and powerful languages in their own right.

COMPLETE LANGUAGES

A language $L=(E,C,\mu)$ is complete iff:

CL1) $C \subseteq E$

CL2) μ is a partial function from E onto C.

CL3) C is the set of fixedpoints of μ.

The domain of discourse, C, of L is thus a subset of the expressions of L and the expressions $c \varepsilon C$ are called constants, since $\mu c=c$. If e is an expression of L and μe is defined, μe is the [unique] meaning of e. μ is the semantic function of L. If a language is not complete, it is incomplete.

Example: The state language of Algol

Let E be the set of states of the "Algol machine". Let μe be the state in which the machine stops after starting in state e; if it does not stop, let μe be undefined. Let C be the set of terminal states, μe. Then we call $L=(E,C,\mu)$ the state language [see below] of Algol. It is a complete language and any "realization" of it [see below] is a major part of a realization of Algol.

Example: The λ-calculus of Church [2]

Let E be the set of well-formed formulas of the λ-calculus. Let C be the set of principal normal forms. Let μe be defined for $e \varepsilon E$, when e has a normal form, and be the principal normal form of e. Then $L=(E,C,\mu)$ is a complete language in which the meaning of a well-formed formula is its principal normal form.

State languages

If $L=(E,D,\sigma)$ is a language and if $\Sigma=(S,C,\mu)$ is a complete language such that $D=\{(s,\mu s) \mid s\varepsilon S \ \& \ \mu s \text{ defined}\}$, then we say that Σ is the state language of L.

Thus a language has a state language $\Sigma=(S,C,\mu)$ only when its domain of discourse is the set of pairs $(s,\mu s)$ ["computations"] which comprise the semantic function μ of Σ.

see also definition of process in "Process Structuring" by Randell & Horning (Computing Surveys 5 1 [March 1973])

LANGUAGE REALIZATIONS

A complete realization R is a triple (E,C,τ) such that: (Vol. 1, p. 7)

Knuth defines a computational method (Q,I,Ω,f) where $I,\Omega \subseteq Q$, $f: Q \to Q$, $f(q)=q \ \forall q \in \Omega$.

I corresponds to E (and must have I=Q)
C " " Ω
f " " τ

CR1) $C \subseteq E$

CR2) τ is a total function from E into E.

CR3) C is the set of fixedpoints of τ.

As with complete languages, E comprises the expressions of R, and C the constants of R. τ is called the transition function of R.

If $R=(E',C',\tau)$ is a complete realization and $L=(E,C,\mu)$ is a complete language, we say that R realizes L, or that R is a realization of L iff:

R1) $E \subseteq E'$

R2) $C \subseteq C'$

R3) μe is defined for $e\varepsilon E$ iff there is an integer n such that $\tau^n e \varepsilon C$, in which case $\mu e = \tau^n e$.

We say that R strictly realizes L or that R is a strict realization of L if $E=E'$ and $C=C'$.

Thus to find μe in $L=(E,C,\mu)$, one computes τe, $\tau(\tau e)$, $\tau(\tau(\tau e)),\ldots$ in some realization $R=(E',C',\tau)$ of L. If this sequence

reaches a fixedpoint of τ ["converges"] within C, then that is μe. If no fixedpoint of τ is reached, or if it lies in C'-C, then μe is undefined. For every complete realization $R=(E,C,\tau)$ there is a corresponding complete language $L=(E,C,\mu)$ which R realizes.

The principal task in realizing an incomplete language L is that of finding a complete realization of a state language of L, if L admits one, as indicated earlier. We shall not try to formalize this notion further at present.

Example: A strict realization of the λ-calculus

Let E be the well-formed formulas [wffs] and C the principal normal forms of the λ-calculus [2] as in the earlier example. Let τ be any definite rule which when applied to a wff e: (a) performs a contraction on a selected part $((\lambda xM)N)$ of e, or (b) converts a normal form e into its principal normal form. Then $R=(E,C,\tau)$ is a strict realization of the λ-calculus language of the earlier example. Thus if e has a normal form, τe, $\tau(\tau e)$,... will converge at μe, otherwise it will not converge. The Church-Rosser theorem states that any two such rules τ and τ' will give the same result.

Example: A realization of an APL sublanguage

We give here a partial sketch of a complete language $L=(E,C,\mu)$, whose expressions are single APL [6] statements except branches, which probably does not have a strict realization. The following are sample expressions $e\varepsilon E$ of L followed by their meanings:

e	μe
3+4	7
N←5	5
B+B←2+A←1	6

[Recall that APL evaluation is from right to left and that "A←1" means "assign 1 to A".] We observe that if R=(E',C,τ) is to realize L with a simple transition function, then E' must contain special expressions in addition to those in E. These special expressions have an "environment" part to keep the values of variables. We indicate such a realization R by giving examples of transitions in E' produced by τ:

```
3+4 → [3+4;0] → [7;0] → 7
N←5 → [N←5;0] → [N;(N=5)] → [5;(N=5)] → 5
B+B←2+A←1 → [B+B←2+A←1;0] → [B+B←2+A;(A=1)]
     → [B+B←2+1;(A=1)] → [B+B←3;(A=1)]
     → [B+B;(A=1,B=3)] → [B+3;(A=1,B=3)]
     → [3+3;(A=1,B=3)] → [6;(A=1,B=3)] → 6
```

CONSTRUCTOR SYNTAX [word algebras]

We say that (A,K) is a constructor syntax for E [or for L=(E,C,μ)] iff:

CS1) $A \subseteq E$

CS2) Each $k \in K$ is a [partial function from Sk] function from a subset Sk of E^n [for some non-negative integer n] into E.

CS3) For every $e \in E$, either $e \in A$ or there is a unique $k \in K$ and unique $e1,\ldots,en \in E$ such that:

$$k[e1,\ldots,en] = e$$

CS4) Every member of E can be obtained by a finite number of applications of CS1-CS3 above.

Each $a \in A$ is called an atom of E [or of L=(E,C,μ)]. Following Landin [5], each $k \in K$ is called a constructor of E [or of L]. We write kn to indicate that $kn \in [E^n \to E]$. Thus if E has a constructor syntax, every expression $e \in E$ is either atomic or has a unique representation $kn[e1,\ldots,en]$, and every such representation, provided $[e1,\ldots,en] \in Skn$, denotes a valid expression. When e is not atomic, the

$kn: Skn \to E$, where $Skn \subseteq E^n$

kn is bad notation, since n is not subscript. Would like K_n^m - the nth m-ary constructor

expressions e1,...,en are called the components of e. A subexpression of e is either (a) e itself, or (b) a component of e, or (c) a subexpression of a component of e. Note that CS2 allows constructors k0 which construct a non-atomic expression e with no components: k0[] = e. [In this and similar cases we consider the existence of the unique e1,...,en postulated in CS3 to be vacuously fulfilled.]

APPLICATIVE LANGUAGES

A complete language L=(E,C,μ) with constructor syntax (A,K) is applicative iff:

AL0) $\mu\omega = \omega$

AL1) $A \subseteq C$

AL2) There is a two-place constructor ap ε K such that for all e,f∈E:

$$\mu\cdot ap[e,f] = \mu\cdot ap[\mu e, \mu f]$$ [both defined or both undefined]

AL3) For every kn∈K except ap, and every [e1,...,en] ε Skn:

$$\mu\cdot kn[e1,\ldots,en] = kn[\mu e1,\ldots,\mu en]$$ [when left side defined]

Expressions constructed by ap are called applications. No construction is defined if a component is undefined. In an applicative language atoms are their own meaning, and the meaning of any constructed expression e is found by first replacing its components by their meanings; if e is not an application, this yields its meaning completely; if it is, its meaning is unchanged and e has been simplified [in general]. The semantics of all expressions except applications are thereby determined; such non-applications cannot themselves specify any computation, only applications may do so. Finally, applicative languages have the reduction property [see

Introduction, property (6) of closed applicative languages].

Example: The constructor syntax for the λ-calculus

Let A be an infinite set of variables. Let $A \subseteq E$. Let Fe denote the set of free variables of e. For $a \in A$, let $Fa=\{a\}$. For all $f,b \in E$ let $ap[f,b] = (fb)$ and let $F(fb) = Ff \cup Fb$. Let $S\lambda = \{[a,e] \mid e \in E \ \& \ a \in Fe\}$. Let λ be the constructor such that $\lambda[a,e] = (\lambda ae)$ for all $[a,e] \in S\lambda$, and let $F(\lambda ae) = Fe - \{a\}$. Then $(A,\{ap,\lambda\})$ is a constructor syntax for E, the set of well-formed formulas of the λ-calculus.

λa.b not allowed

The λ-calculus fails to satisfy axiom AL3 for applicative languages since μe produces the principal normal form of e. Thus if "a" and "x" are two variables in A and a is the "first" variable, then by definition $\mu\cdot\lambda[x,x] = \lambda[a,a]$ where the second expression is the principal normal form of the first. But by AL3 we should have:

$$\mu\cdot\lambda[x,x] = \lambda[\mu x,\mu x] = \lambda[x,x]$$

Examples of applicative languages appear later in the paper as examples of closed applicative languages; one of them is very similar to the λ-calculus.

CLOSED APPLICATIVE LANGUAGES [CALs]

Applicative languages specify the semantics for all expressions but applications. Closed applicative languages, or CALs, prescribe the semantics of applications. A "representation function" associates a function with each constant expression.

An applicative language $L=(E,C,\mu)$ is closed iff there is a function $\rho \in [C \to [C \to E]]$ such that:

CAL1) ρ is total over C.

CAL2) For every $c \varepsilon C$, $\rho c \varepsilon [C \to E]$ is total over C.

CAL3) For all $c,d \varepsilon C$: $\mu \cdot ap[c,d] = \mu \cdot [\rho c]d$

The function ρ is the representation function of L.

To find the meaning of an application ap[e,f] in a CAL, one notes that by AL2 this is the same as the meaning of ap[c,d], where $c=\mu e$ and $d=\mu f$, if these exist; otherwise ap[e,f] has no meaning. Since $c,d \varepsilon C$, the meaning of ap[c,d] is found, by CAL3, by applying the function ρc to d to get the result g and then finding the meaning of g, if it exists. [The function ρc and the result g are guaranteed to exist by CAL1 and CAL2, but g may be without meaning.] Thus the meaning of ap[e,f] is the meaning of g if it exists. Note that A, K, and ρ completely determine $L=(E,C,\mu)$. Finally, if, for every $c \varepsilon C$, $\rho c \varepsilon [C \to C]$, then μe is defined for every finite expression e, and such a CAL with all expressions finite is trivial.

Discussion

We should now be more precise about some of the properties of closed applicative languages listed in the introduction [we use the same numbers as used there].

3) Non-extensionality

CALs are non-extensional. In extensional systems, applications, ap[f,g], have the following interpretation: "apply the function denoted by the expression f to the function denoted by the expression g". In CALs, on the other hand, ap[f,g] has the interpretation: "apply the function represented by f to the expression g itself". In an extensional system no function can yield different results for distinct arguments g and g', provided g and g' denote the same thing. In such systems, as in the λ-calculus, one needs a "Gödel-numbering"

G(g) of expressions g to distinguish between different expressions which denote the same thing; however, no Gödel-numbering function can be denoted by any expression G within the system, for ap[G,g] and ap[G,g'] must denote the same object if g and g' denote the same object. In accordance with this observation one observes that a LISP Gödel-numbering is achieved by QUOTE which cannot be, and is not, a function. In CALs each expression serves as its own Gödel number, for if g and g' are distinct but represent the same function, then ap[f,g] and ap[f,g'] do not, in general, denote the same expression.

4) Single-type functions

All functions represented by CAL expressions are of a single type [they map C into E]. Thus the infinite hierarchy of function types required in models of the λ-calculus, as in Scott [1], is avoided. In a lattice-theoretic treatment of CALs, the hierarchy of function types is replaced by a hierarchy of orderings on E. Let $\leq_0$ be the basic ordering of expressions [$\leq_0$ is defined on C in the usual way and extended to E by: $e \leq_0 f$ iff μe, μf exist and $\mu e \leq_0 \mu f$] then we define

$$f \leq_{i+1} g \quad \text{iff} \quad [\rho f]e \leq_i [\rho g]e \quad \text{for all} \quad e \in E$$

Thus, while all orderings are on E, one can think of $\leq_1$ as ordering functions and $\leq_2$ as ordering functionals, etc. For if $f \leq_2 g$ and $g \leq_2 f$, then f and g represent equivalent functionals, that is, $[\rho f]e$ represents the same function that $[\rho g]e$ does, for every expression e. [A rather pretty Scott-like theory for CAL semantics should be achievable beginning along these lines.]

5) The extended Church-Rosser property

If e is the CAL expression ap[c,d], where $c,d \in C$, then by a "reduction" τe of e we mean the replacement of ap[c,d] by [ρc]d. For any other expression e, let τe be the result of replacing any

non-constant component f of e by τf. When we say CALs have the Church-Rosser property, we mean that any two converging sequences of reductions on an expression e will converge to the same constant μe. By the extended Church-Rosser property we mean, in addition, that if μe is defined, then every sequence of reductions on e will converge. This property implies that every subexpression of a meaningful expression is meaningful. CALs have this property, while the λ-K-calculus and LISP do not ["reduction" has a slightly different meaning in these latter cases].

DESCRIPTION OF CLASSES OF EXPRESSIONS AND OF FUNCTIONS

We assume the context relates to some CAL $L=(E,C,\mu)$ with constructor syntax (A,K). We let an expression involving meta-linguistic [underlined] variables denote the class of proper expressions obtained by replacing these variables by any expression in their range. Thus if we assign the ranges: $\underline{a}\varepsilon A$ and $\underline{e}\varepsilon E$, then the following definition by cases,

$$\alpha\underline{x} \equiv \underline{x}=(\underline{a},\underline{a}) \rightarrow \emptyset;\ \underline{x}=(\underline{a},\underline{e}) \rightarrow (\underline{x},\underline{e});0$$

means: " if $\underline{x}$ is an expression of the form $(\underline{a},\underline{a})$ [i.e., both components the same element of A], then $\alpha\underline{x}$ is the atom $\emptyset$, if $\underline{x}$ is of the form $(\underline{a},\underline{e})$ then $\alpha\underline{x}$ is $(\underline{x},\underline{e})$ $[=((\underline{a},\underline{e}),\underline{e})]$, and if $\underline{x}$ is of neither of these forms, then $\alpha\underline{x}$ is the atom 0". The first applicable clause governs the result. The connectives in a definition by cases are: $\{\equiv = \rightarrow ;\}$. Non-underlined marks and strings other than connectives denote themselves.

RED LANGUAGES

A Red language ["Red" for "Reduction"] is any of the CALs satisfying the definitions below. One Red language may differ from another in its atoms and in its "primitive functions", those represented by atoms. The Red languages described here differ from, and are simpler than those described in an earlier report [4].

Red syntax

Although it is not necessary, we give particular constructions for each Red constructor simply to be specific.

Constructors: $K=\{ap; \sigma n, n=1,2,...; \omega\}$

Let $e,f,e1,...,en \in E$. Then:

$ap[e,f] = \langle e,f \rangle$

$\sigma n[e1,...,en] = (e1,...,en)$ for $n=1,2,...$

$\omega[] = \perp$

Expressions constructed by σn are called sequences of length n. Sequences of length 2 are called pairs. [A simpler kind of Red language would have $K=\{ap,\sigma 2,\omega\}$ since sequences can be built from pairs as in LISP. We prefer the present approach because pairs are then always sequences.] There are thus four types of Red expressions: atoms a; applications $\langle e,f \rangle$; sequences $(e1,...,en)$; and $\perp$ ["bottom", or "undefined"]. The first component of an application is called its operator, the second, its operand.

If A is a set of atoms which do not use the special characters of the above constructions, then A and the above K generate the set of expressions E of a Red language. We assume further that: Ø, T, F $\in$ A;

and in general we use non-underlined strings of letters, digits and other characters as atoms. We use Ø for "empty sequence" and to identify certain "meta expressions" [see below]. Ø also represents

the identity function. The atoms T and F are used for "true" and "false".

Red semantics

The semantic function μ of a Red language $L=(E,C,\mu)$ is determined by the CAL axioms and by the representation function ρ. The definitions of ρ and of the predicate ψ, on which ρ depends, are circular and thus undefined for all atoms but $\emptyset$. The definition of a particular Red language comprises the extension of the definitions of ρ and ψ to all atoms. For the following definitions we let $\underline{c}1,\ldots,\underline{c}n \in C$ and $\underline{a}\in A$.

Auxilliary functions

1) Tail: t

$t\underline{x} \equiv \underline{x}=(\underline{c}1) \rightarrow \emptyset;\ \underline{x}=(\underline{c}1,\underline{c}2,\ldots,\underline{c}n) \rightarrow (\underline{c}2,\ldots,\underline{c}n)$

2) The everywhere ~~undefined~~ unspecified function: Ω

$\Omega\underline{x} \equiv \perp$

3) The pairing function: π

For every $\underline{c},\underline{d}\in C$, $\pi\underline{c}$ is the function such that $[\pi\underline{c}]\underline{d} = (\underline{c},\underline{d})$.

4) The meta predicate: ψ

$\psi\underline{x} \equiv \underline{x}=\underline{a} \rightarrow \psi\underline{a};\ \underline{x}=(\emptyset) \rightarrow T;\ \underline{x}=(\emptyset,\underline{c}1,\ldots,\underline{c}n) \rightarrow T;\ F$

and $\psi\emptyset = F$

The representation function for Red languages: ρ

$\rho\underline{x} \equiv \underline{x}=\underline{a} \rightarrow \rho\underline{a};$

$\underline{x}=(\underline{c}1,\ldots,\underline{c}n)\ \&\ \psi\underline{c}1 \rightarrow [\rho\underline{c}1]\cdot[\pi\underline{x}];$

$\underline{x}=(\underline{c}1,\ldots,\underline{c}n) \rightarrow [\rho\underline{c}1]\cdot\mu\cdot[\rho\cdot t\underline{x}];\ \Omega$

and $\rho\emptyset$ = the identity function

This completes the general definition of Red languages.

Remarks

The first clause in the definition of ρ is the primitive function rule and depends on how ρ is defined on A for the particular Red language. The second clause is the meta composition rule and depends on ψ. The third clause is the regular composition rule. The final clause applies only to $\perp$, and yields $\rho\perp\equiv\Omega$.

If $\psi\underline{c}=T$, we say that $\underline{c}$ is a meta expression, otherwise $\underline{c}$ is a regular expression. When $\underline{c}1$ is a meta expression the function represented by $\underline{c}=(\underline{c}1,\ldots,\underline{c}n)$ satisifies:

$$[\rho\underline{c}]\underline{d} = [\rho\underline{c}1](\underline{c},\underline{d}) \qquad \text{for all } \underline{d}\varepsilon C$$

In regular composition, as in $[\rho(\underline{c}1,\ldots,\underline{c}n)]\underline{d}$ [with $\underline{c}1$ regular], the argument of $\rho\underline{c}1$ is the result: $\mu\cdot[\rho(\underline{c}2,\ldots,\underline{c}n)]d$ [if it exists]; whereas in meta composition [with $\underline{c}1$ meta] the argument of $\rho\underline{c}1$ is the constant: $((\underline{c}1,\ldots,\underline{c}n),\underline{d})$. Meta composition allows the definition of functions with parameters: a meta expression f may be constructed so that the meaning of $\langle(f,g),h\rangle$ depends on g and on h in a manner entirely controlled by ρf. When a meta operator f is designed to use the functions represented by its parameters, rather than the parameters themselves, it is called a modifier.

Examples

We let ρ and ψ be defined for the following primitive operators for use in the subsequent examples. Let $\underline{c},\underline{d},\underline{e},\underline{f} \varepsilon C$. Then we define:

$$[\rho DBL]\underline{x} \equiv \underline{x}=\perp \rightarrow \perp;\ \underline{x} \rightarrow (\underline{x},\underline{x})$$

$$\psi DBL \equiv F$$

$$[\rho FST]\underline{x} \equiv \underline{x}=((\underline{c},\underline{d}),(\underline{e},\underline{f})) \rightarrow (\langle\underline{d},\underline{e}\rangle,\underline{f});\ \perp$$

$$\psi FST \equiv T$$

We now compute in detail [for much simpler rules, see below] the

meanings of the following expressions, giving references to the appropriate axioms [e.g., CAL3], the meta composition rule [MCR] or the regular composition rule [RCR], and to the definitions [def] of functions:

(1) $\mu\langle(DBL,DBL),A\rangle$

$= \mu\cdot[\rho(DBL,DBL)]A$ by CAL3

$= \mu\cdot[\rho DBL]\cdot\mu\cdot[\rho\cdot t(DBL,DBL)]A$ by def ψDBL & RCR

$= \mu\cdot[\rho DBL]\cdot\mu\cdot[\rho(DBL)]A$ by def t=tail

$= \mu\cdot[\rho DBL]\cdot\mu\cdot[\rho DBL]\cdot\mu\cdot[\rho\emptyset]A$ by def ψDBL, RCR, def t

$= \mu\cdot[\rho DBL]\cdot\mu\cdot[\rho DBL]\cdot\mu A$ by def $\rho\emptyset$

$= \mu\cdot[\rho DBL]\cdot\mu\cdot[\rho DBL]A$ by AL1 & CL3

$= \mu\cdot[\rho DBL]\cdot\mu(A,A)$ by def ρDBL

$= \mu\cdot[\rho DBL](\mu A,\mu A)$ by AL3

$= \mu\cdot[\rho DBL](A,A)$ by AL1 & CL3

$= ((A,A),(A,A))$ by def ρDBL, AL3 & AL1 & CL3

(2) $\mu\langle(FST,DBL),(A,B)\rangle$

$= \mu\cdot[\rho(FST,DBL)](A,B)$ by CAL3

$= \mu\cdot[\rho FST]\cdot[\pi(FST,DBL)](A,B)$ by def ψFST & MCR

$= \mu\cdot[\rho FST]((FST,DBL),(A,B))$ by def π

$= \mu(\langle DBL,A\rangle,B)$ by def ρFST

$= (\mu\langle DBL,A\rangle,\mu B)$ by AL3

$= ([\rho DBL]A,B)$ $(\mu\cdot[\rho DBL]A,B)$ by CAL3 and AL1 & CL3

$= ((A,A),B)$ by def ρDBL, AL3, AL1, & CL3

Red realizations

The above examples illustrate the detailed operation of the axioms and definitions governing Red semantics. These semantics are more simply given by the following transition rule τ for a strict realization of a Red language.

To find the meaning of a Red expression e:

(τ) Find an innermost application $\langle c,d\rangle$ in e and apply the appropriate one of the following transitions to $\langle c,d\rangle$:

$\tau 1$) If c is an atom: $\langle c,d\rangle \rightarrow [\rho c]d$

$\tau 2$) If $c=(c1,\ldots,cn)$ & $\psi c1=T$: $\langle c,d\rangle \rightarrow \langle c1,(c,d)\rangle$

$\tau 3$) If $c=(c1,\ldots,cn)$ & $\psi c1=F$:

if $n>1$: $\langle c,d\rangle \rightarrow \langle c1,\langle(c2,\ldots,cn),d\rangle\rangle$

if $n=1$: $\langle c,d\rangle \rightarrow \langle c1,d\rangle$

$\tau 4$) If $c=\perp$: $\langle c,d\rangle \rightarrow \perp$

Repeat (τ) until e contains no more applications; the resulting expression is μe. If this process does not terminate, μe is undefined [this does not mean that μe is $\perp$].

[in the partial function theory, it does mean $\mu e=\perp$!]

The meanings of the above two examples are found by the following transitions:

(1) $\langle(DBL,DBL),A\rangle \rightarrow \langle DBL,\langle(DBL),A\rangle\rangle \rightarrow \langle DBL,\langle DBL,A\rangle\rangle$

$\rightarrow \langle DBL,(A,A)\rangle \rightarrow ((A,A),(A,A))$

(2) $\langle(FST,DBL),(A,B)\rangle \rightarrow \langle FST,((FST,DBL),(A,B))\rangle$

$\rightarrow (\langle DBL,A\rangle,B) \rightarrow ((A,A),B)$

Definition of new operators; the definition function: δ

When ρ and ψ have been defined for all atoms, a Red language has been defined. Normally, ρa will be a useful "primitive function" for relatively few atoms a [called primitive operators]; for the remaining atoms ρa will be Ω, the everywhere ~~undefined~~ unspecified function. Similarly, ψa will usually be F [i.e., a is regular]. In such a language there are often lengthy constants c which represent very useful functions. One would like to set aside some atom a to represent the function which is represented by some long c. For this purpose we provide a definition function δ, a total function from C into C. For all $c \in C$, δ is the

identity function until a "definition" is made. A definition is an assertion of the form: $\delta\underline{a}=\underline{c}$; thereafter δ maps $\underline{a}$ into $\underline{c}$ [definitions can be given only for atoms $\underline{a}$]. And thereafter $\rho\underline{a} = \rho\cdot\delta\underline{a} = \rho\underline{c}$, and any former definition of $\rho\underline{a}$ is superceded. The effect of the definition function is entirely subsumed in the following

Definition rule:

The functions ρ and ψ satisfy the following for all $\underline{c}\varepsilon C$:

DR) $$\rho\underline{c} = \rho\cdot\delta\underline{c}$$
$$\psi\underline{c} = \psi\cdot\delta\underline{c}$$

The effect of this rule is nil for all $\underline{c}\varepsilon C-A$ and all $\underline{a}\varepsilon A$ for which there is no definition. Thus if SIN is a primitive operator and ρSIN=sine, and ψSIN=F, then the definition: δSIN2=(SIN,SIN) results in:

$$\rho\text{SIN2} = \rho\cdot\delta\text{SIN2} = \rho(\text{SIN,SIN}) = [\rho\text{SIN}]\cdot\mu\cdot[\rho\text{SIN}] = (\text{sine})^2$$
$$\psi\text{SIN2} = \psi\cdot\delta\text{SIN2} = \psi(\text{SIN,SIN}) = \text{F}$$

Starting with a Red language L with its ρ and ψ, each definition $\delta\underline{a}=\underline{c}$ yields a new language L' and a new ρ' and ψ' which differ from the originals only in that the function $\rho'\underline{a}$ represented by $\underline{a}$ is now $\rho\underline{c}$ instead of $\rho\underline{a}$, and $\psi'\underline{a}$ now equals $\psi\underline{c}$, and if $\delta\underline{b}=(\ldots\underline{a}\ldots)$ was an earlier definition, then $\rho'\underline{b}$ would also differ from $\rho\underline{b}$.

It is possible and convenient to make unambiguous recursive definitions $\delta\underline{a}=\underline{c}$ in which $\underline{a}$ occurs in $\underline{c}$ [see below]. We observe, however, that given a Red language with its μ and ρ, and with a suitable set of primitive operators but no definitions, then, given any computable function $\alpha:C \to C$, there is a constant $\underline{c}$ such that

$$\beta\underline{c} = \mu\cdot[\rho\underline{c}] = \alpha$$

[We call $\beta\underline{c}$ the basic function of $\underline{c}$.] Thus our introduction of the definition function δ and its modification of Red semantics is not essential to the definitional power of Red languages. This power arises from the existence of composite operators and the two composition rules. [actually, metacomposition is the key]

In a Red language realization, the effect of the definition function δ on the transition function τ [see above] is to change the first three subrules of τ as follows. For any innermost application $\langle\underline{c},\underline{d}\rangle$:

$\tau 1)$ If $\underline{c}\varepsilon A$ and $\delta\underline{c}\neq\underline{c}$ then: $\langle\underline{c},\underline{d}\rangle \rightarrow \langle\delta\underline{c},\underline{d}\rangle$

if $\delta\underline{c}=\underline{c}$ then: $\langle\underline{c},\underline{d}\rangle \rightarrow [\rho\underline{c}]\underline{d}$

$\tau 2)$ If $\underline{c}=(\underline{c}1,\ldots,\underline{c}n)$ & $\psi\cdot\delta\underline{c}1=T$, then: $\langle\underline{c},\underline{d}\rangle \rightarrow \langle\not\delta\underline{c}1,(\underline{c},\underline{d})\rangle$

$\tau 3)$ If $\underline{c}=(\underline{c}1,\ldots,\underline{c}n)$ & $\psi\cdot\delta\underline{c}1=F$:

if $n>1$, then: $\langle\underline{c},\underline{d}\rangle \rightarrow \langle\not\delta\underline{c}1,\langle(\underline{c}2,\ldots,\underline{c}n),\underline{d}\rangle\rangle$

if $n=1$, then: $\langle\underline{c},\underline{d}\rangle \rightarrow \langle\not\delta\underline{c}1,\underline{d}\rangle$

can remove in $\tau 2$ and $\tau 3$ (only)

Recursive definitions

To see why a recursive definition $\delta\underline{a}=\underline{c}$ with $\underline{a}$ occurring in $\underline{c}$ need not result in a circular definition of $\rho\underline{a}$, consider the following definition:

$$\delta F = (CN,P,F,G)$$

If CN is meta, then

$$\rho F = \rho(CN,P,F,G) = \rho CN\cdot\pi(CN,P,F,G)$$

Thus $[\rho F]\underline{x} = [\rho CN]((CN,P,F,G),\underline{x})$. Hence ρF is not circularly defined since ρF does not occur on the right hand side. Since all represented functions are total, $[\rho CN]((CN,P,F,G),\underline{x})$ is always defined, hence $[\rho F]\underline{x}$ is always defined and yields some expression $\underline{y}$. Now $\underline{y}$ may or may not be meaningful; i.e., βF may be partial or

[Provided the expansion of ρCN does not involve ρF.]

undefined. When we say here that βF may be undefined, we mean that it may be everywhere non-terminating. This is quite different from saying that βF is Ω; the latter is a total function which always yields the expression ⊥, called "bottom".

Examples

(1) Let $[\rho R]\underline{x} \equiv \underline{x}=((\underline{c},\underline{d}),\underline{e}) \rightarrow (\underline{d},\underline{e})\,;\ \perp$. Let $\psi R=F$. Let $\delta ADJ=(\emptyset,R)$ be the definition of ADJ. Then:

$<(ADJ,B),C> \rightarrow <ADJ,((ADJ,B),C)>$ by τ2, since $\psi\cdot\delta ADJ=T$

$\rightarrow <(\emptyset,R),((ADJ,B),C)>$ by τ1, since $ADJ\varepsilon A$ & $\delta ADJ\neq ADJ$

$\rightarrow <\emptyset,<(R),((ADJ,B),C)>>$ by τ3, since $\psi\cdot\delta\emptyset = \psi\emptyset = F$

$\rightarrow <\emptyset,<R,((ADJ,B),C)>>$ by τ3, n=1, since $\psi R=F$

$\rightarrow <\emptyset,(B,C)>$ by τ1, since $R\varepsilon A$ & $\delta R=R$ & def ρR

$\rightarrow (B,C)$ by τ1, since $\emptyset\varepsilon A$ & $\delta\emptyset=\emptyset$ & def $\rho\emptyset$

(2) The following is an example of a non-terminating operator. Let $[\rho APP]\underline{x} \equiv \underline{x}=(\underline{c},\underline{d}) \rightarrow <\underline{c},\underline{d}>\,;\ \perp$. Let $\delta NT=(\emptyset,APP)$ be the definition of NT. Then:

$<(NT),X> \rightarrow <(\emptyset,APP)((NT),X)>$

$\rightarrow <\emptyset,<(APP),((NT),X)>> \rightarrow <\emptyset,<APP,((NT),X)>>$

$\rightarrow <\emptyset,<(NT),X>> \rightarrow \ldots \rightarrow \ldots$

Here ρNT is everywhere defined, but βNT is everywhere non-terminating.

Remarks

There are special circumstances under which a definition $\delta\underline{a}=\underline{c}$ does not have the same effect on the meaning of an expression $\underline{e}$ as the replacement in $\underline{e}$ of every occurrence of $\underline{a}$ by $\underline{c}$. For example consider the definition $\delta\underline{a}=(\emptyset,\underline{f})$ in connection with the expression $<(\underline{a}),\underline{c}>$. After two transitions the latter becomes

$<(\emptyset,\underline{f}),((\underline{a}),\underline{c})>$

On the other hand, replacing a by (Ø,f) in the original expression yields <((Ø,f)),c> and one transition gives:

<(Ø,f),(((Ø,f)),c)>

For those f whose represented function depends on the first member of the operand, i.e., for those f that depend on their own structure, the two expressions will have different meanings.

λ-RED LANGUAGES

λ-Red languages are closed applicative languages which resemble the λ-calculus. They serve to show the merit of the CAL distinction between "meaning" and "represented function": there is no longer a need to require that (λxx) be "equal" [convertible] to (λyy) simply because they represent the same function. As with all CALs it is normal that there should be expressions which represent the same function but have different meanings. Other pleasant properties of λ-Red languages are: (1) they have simple strict realizations, (2) no bound variables need changing [i.e., no I-conversion or α-conversion], and (3) every innermost application of a meaningful expression can be immediately evaluated. Realizations are not so simple for the λ-calculus because (2) and (3) do not hold. In ((λxM)N), which we would write as <(λxM),N>, if x is a free variable of N, then x must be replaced in the operator by another variable before the application can be evaluated. In the expression <(λx<x,y>),(λzz)> the innermost application cannot be dealt with until the outer one is reduced.

λ-Red syntax

To begin with we are given two disjoint sets: V, the set of variables, and O, the set of objects. We are also given the set K of four two-place constructors: pair, lambda, ap [application], and fap

[formal application], and the zero-place constuctor ω which constructs $\perp$. To describe the range of each constructor we also need the function ϕ from expressions into subsets of V, where $\phi\underline{e}$ is the set of free variables of $\underline{e}$. The expressions constructed by fap are called formal applications; they are not applications but may be later changed into applications when all free variables have been eliminated from their components. The expressions constructed by the other constructors are: pairs, lambda-expressions, and applications. For convenience we give particular constructions for each constructor:

$$\text{pair}[\underline{e},\underline{f}] = (\underline{e},\underline{f})$$

$$\text{lambda}[\underline{e},\underline{f}] = (\lambda\underline{e},\underline{f})$$

$$\text{ap}[\underline{e},\underline{f}] = \langle\underline{e},\underline{f}\rangle$$

$$\text{fap}[\underline{e},\underline{f}] = \lceil\underline{e},\underline{f}\rfloor$$

$$\omega[] = \perp$$

We now give a recursive description of E for λ-Red languages. With $A=V\cup O$, this defines the constructor syntax (A,K) for the expressions E of a λ-Red language.

1) If $\underline{v}\in V$, then $\underline{v}\in E$ and $\phi\underline{v}=\{\underline{v}\}$
2) If $\underline{o}\in O$, then $\underline{o}\in E$ and $\phi\underline{o}=\emptyset$
3) If $\underline{e},\underline{f}\in E$, then $(\underline{e},\underline{f})\in E$ and $\phi(\underline{e},\underline{f}) = \phi\underline{e} \cup \phi\underline{f}$
4) If $\underline{v}\in V$ and $\underline{e}\in E$, then $(\lambda\underline{v},\underline{e})\in E$ and $\phi(\lambda\underline{v},\underline{e}) = \phi\underline{e} - \{\underline{v}\}$
5) If $\underline{e},\underline{f}\in E$ and $\phi\underline{e} \cup \phi\underline{f} = \emptyset$, then $\langle\underline{e},\underline{f}\rangle\in E$ and $\phi\langle\underline{e},\underline{f}\rangle = \emptyset$
6) If $\underline{e},\underline{f}\in E$ and $\phi\underline{e} \cup \phi\underline{f} \neq \emptyset$, then $\lceil\underline{e},\underline{f}\rfloor\in E$ and $\phi\lceil\underline{e},\underline{f}\rfloor = \phi\underline{e} \cup \phi\underline{f}$
7) $\perp\in E$ and $\phi\perp=\emptyset$

The difference between $\langle\underline{e},\underline{f}\rangle$ and $\lceil\underline{e},\underline{f}\rfloor$ is that the first is well-formed only if $\underline{e}$ and $\underline{f}$ have no free variables and that the second is well-formed only otherwise. We say that an occurrence of a free variable $\underline{v}$ of $\underline{e}$ is a free occurrence, if it is a free variable of

every subexpression of $\underline{e}$ in which it occurs.

λ-Red semantics

Since λ-Red languages are CALs, their semantics are determined by a representation function ρ. However, we summarize the effects of the CAL axioms for λ-Red expressions. For all $\underline{e},\underline{f}\varepsilon E$, $\underline{v}\varepsilon V$, and $\underline{a}\varepsilon A$ [=V∪O]:

$\mu\underline{a} = \underline{a}$

$\mu(\underline{e},\underline{f}) = (\mu\underline{e},\mu\underline{f})$

$\mu(\lambda\underline{v},\underline{e}) = (\lambda\mu\underline{v},\mu\underline{e}) = (\lambda\underline{v},\mu\underline{e})$

$\mu\langle\underline{e},\underline{f}\rangle = \mu\cdot[\rho\cdot\mu\underline{e}][\mu\underline{f}]$

$\mu\ulcorner\underline{e},\underline{f}\urcorner = \ulcorner\mu\underline{e},\mu\underline{f}\urcorner$

C is the set of expressions containing no applications [they may contain formal applications].

The auxilliary function: $\lambda(\underline{v},\underline{c})$

For every variable $\underline{v}$ and every constant $\underline{c}$, $\lambda(\underline{v},\underline{c})$ is a function from C into E. For any $\underline{d}\varepsilon C$, $[\lambda(\underline{v},\underline{c})]\underline{d}$ is obtained as follows:

λ1) Replace every free occurrence of $\underline{v}$ in $\underline{c}$ by $\underline{d}$. This yields $\underline{f}$ [$\underline{f}$ may have a non-well-formed subexpression $\ulcorner\underline{p},\underline{q}\urcorner$ which has no remaining free variables after the substitution of $\underline{d}$ for $\underline{v}$].

λ2) Replace every [non-well-formed] subexpression of $\underline{f}$ of the form $\ulcorner\underline{p},\underline{q}\urcorner$ which has no free variables by the expression $\langle\underline{p},\underline{q}\rangle$. The resulting expression is well-formed and is $[\lambda(\underline{v},\underline{c})]\underline{d}$.

The representation function for λ-Red languages: ρ

As before, ρ is circular and undefined for objects. The extension of ρ to objects supplies the primitive functions of a particular language. Let $\underline{o}\varepsilon O$, $\underline{v}\varepsilon V$, and $\underline{c},\underline{d}\varepsilon C$, then:

$$\rho\underline{x} \equiv \underline{x}=\underline{o} \rightarrow \rho\underline{o};\ \underline{x}=\underline{v} \rightarrow \Omega;$$
$$\underline{x}=\ulcorner\underline{c},\underline{d}\urcorner \rightarrow \Omega;$$
$$\underline{x}=(\underline{c},\underline{d}) \rightarrow [\rho\underline{c}]\cdot\mu\cdot[\rho\underline{d}];$$
$$\underline{x}=(\lambda\underline{v},\underline{d}) \rightarrow \lambda(\underline{v},\underline{d});\ \Omega$$

This completes the general definition of λ-Red languages.

Remarks

The fourth clause of the definition of ρ is the regular composition rule as in Red languages, here applied to pairs only. The fifth clause is the abstraction rule.

Examples

We show in detail the operation of λ-Red semantics. At each step we cite the appropriate CAL axioms or the abstraction rule [AR] or the two rules for computing $[\lambda(\underline{v},\underline{c})]\underline{d}$, λ1 and λ2. [Between the use of λ1 and λ2 non-well-formed expressions may appear.] Our examples do not show regular composition since it is similar to that of Red languages. We use lower case letters for variables and upper case strings or letters for objects.

(1) $\mu\langle(\lambda y,(\lambda x,\ulcorner x,y\urcorner)),A\rangle$

$= \mu\cdot[\rho(\lambda y,(\lambda x,\ulcorner x,y\urcorner))]A$ by CAL3

$= \mu\cdot[\lambda(y,(\lambda x,\ulcorner x,y\urcorner))]A$ by AR

$= \mu(\lambda x,\ulcorner x,A\urcorner)$ by λ1

$= (\lambda\mu x,\ulcorner\mu x,\mu A\urcorner)$ by AL3, twice

$= (\lambda x,\ulcorner x,A\urcorner)$ by AL1 and CL3

The resulting expression represents a function which when applied to some $\underline{c}$ yields $\langle \underline{c}, A \rangle$.

(2) $\mu\langle(\lambda x,\ulcorner(\lambda x,(x,x)),x\urcorner),A\rangle$

$= \mu\cdot[\rho(\lambda x,\ulcorner(\lambda x,(x,x)),x\urcorner)]A$ by CAL3

$= \mu\cdot[\lambda(x,\ulcorner(\lambda x,(x,x)),x\urcorner)]A$ by AR

$= \mu\ulcorner(\lambda x,(x,x)),A\urcorner$ by $\lambda 1$; not wf; the one free occurrence of x is replaced.

$= \mu\langle(\lambda x,(x,x)),A\rangle$ by $\lambda 2$

.

.

.

$= (A,A)$

λ-Red realizations

As with Red languages, the semantics of λ-Red languages are best described by the transition function of a strict realization. To find the meaning of a λ-Red expression $\underline{e}$:

(τ) Find an innermost application $\langle \underline{c},\underline{d}\rangle$ in $\underline{e}$ and apply the appropriate one of the following transitions:

$\tau 1$) If $\underline{c}\varepsilon A$: $\langle \underline{c},\underline{d}\rangle \rightarrow [\rho \underline{c}]\underline{d}$

$\tau 2$) If $\underline{c}=(\underline{e},\underline{f})$: $\langle \underline{c},\underline{d}\rangle \rightarrow \langle \underline{e},\langle \underline{f},\underline{d}\rangle\rangle$

$\tau 3$) If $\underline{c}=(\lambda \underline{v},\underline{e})$: $\langle \underline{c},\underline{d}\rangle \rightarrow [\lambda(\underline{v},\underline{e})]\underline{d}$

$\tau 4$) If $\underline{c}=\perp$: $\langle \underline{c},\underline{d}\rangle \rightarrow \perp$

Repeat (τ) until $\underline{e}$ contains no applications; the resulting expression is $\mu\underline{e}$; if the process does not terminate, $\mu\underline{e}$ is undefined. There is no subrule of τ for $\underline{c}=\ulcorner \underline{e},\underline{f}\urcorner$ because $\langle\ulcorner \underline{e},\underline{f}\urcorner,\underline{d}\rangle$ is not well-formed due to the necessary presence of a free variable in the operator. As in Red languages, there is no subrule for $\underline{c}=\langle \underline{e},\underline{f}\rangle$

because <c,d> is an innermost application.

The meanings of the above two examples may be found using these transition rules:

(1) $\langle(\lambda y,(\lambda x,\ulcorner x,y\urcorner)),A\rangle \rightarrow (\lambda x,\ulcorner x,A\urcorner)$

(2) $\langle(\lambda x,\ulcorner(\lambda x,(x,x)),x\urcorner),A\rangle \rightarrow \langle(\lambda x,(x,x)),A\rangle \rightarrow (A,A)$

Remarks

It is possible to also define another class of languages, called λ-M-Red languages, very much like λ-Red languages which have added to the definition of ρ a meta composition rule, thereby allowing the definition of recursive functions without the Y operator.

Note: While the Y operator can be defined within the λ-calculus, it cannot be defined in λ-Red. The problem is that innermost applications are always evaluated first, and Y has no normal-form.

CONCLUDING REMARKS

This paper is essentially an outline for a number of study projects. It proposes precise definitions for various language-classes and their semantics and for their "realizations". It contains many assertions and no proofs. Much work remains to be done on formulating other assertions and on careful proofs. Hopefully, this work will lead to a semantic theory for a new class of programming languages, one which possesses an axiomatic foundation of the simplicity required for rigorous mathematical treatment.

One little-explored area is that of proving assertions about Red and λ-Red programs; several of their properties seem to make them particularly amenable to various proof techniques. Certain examples in an earlier report [4] suggest that it may be possible to mechanize an exhaustive analysis of the behavior of any operator in a large class. There is also the question of constructing a Scott-like theory for the semantics of these languages along the lines suggested in the discussion of "single-type functions" in the section on closed

applicative languages. I hope to clarify some of these issues and hope others will too.

The notion of a "complete language" is believed to be useful in discussing the semantics of any language. As "state languages" they comprise a precise general formulation of a widely used method of semantic description. As independent languages, they have few existing examples.

Closed applicative languages introduce several precise concepts, some of which may be novel. The distinction between "meaning of" and "function represented by" and the non-extensional interpretation of applications are basic notions which make Gödel-numbering and hence manipulation of expressions expressible within the system. Unlike most languages, these new ones all have "strict realizations": their semantics can be described by transitions entirely within the space of expressions.

Almost all existing languages have the opposites of the six properties of closed applicative languages listed in the introduction: non-idempotency of meaning, the "quote" property, extensionality, multiple function-types, and the lack of the extended Church-Rosser and reduction properties are all such traditional language features that their significance has scarcely been noted. The absence to date of languages with the properties of closed applicative languages may make these properties appear radical to some readers; but I believe that it is difficult to deny their attractiveness, at least in the abstract. Their viability is demonstrated to some degree by the existence of two languages [Red and λ-Red] which possess them and which are unusually simple, clean and expressive.

The notion of meta composition is thought to be novel. Its use

in defining functions with parameters is central to the definitional power of Red and λ-M-Red languages.

In the future I hope to show that Red, λ-Red, λ-M-Red and similar languages have attractive properties as very-high-level programming languages. I also hope to show that they can be used to study and model the following: naming, indexing, and addressing systems; parallel computation; the semantics of other programming languages.

My earlier report [4] discusses a variety of topics relating to Red-like languages for which there is no space here, including: variants of Red languages, high-level Red programming, examples, comparisons with other systems, and related work. There are slight differences between the Red languages of that report and those described in this paper, and also some changes in notation. The appendix of the present paper gives some sample Red primitive operators and programs.

Acknowledgements

I am particularly indebted to two people for a great amount of help, interest, criticism, and suggestions: Peter Naur and John C. Reynolds. It is a pleasure to express my gratitude to them. In particular, John Reynolds observed that what I called "languages" in my earlier report were really "realizations" of languages [although he is not to be blamed for my formulation of that notion].

I am also grateful to the following people for various helpful dicussions: Dines Bjørner, Bill Burge, Val Schorre, Phil Summers, and John Williams; and to Howard Smith for providing the APL editing programs used to prepare this paper.

References

[1] Scott, D. Lattice-theoretic models of the λ-calculus. [unpublished report?] Distributed to IFIP WG2.2

[2] Church, A. The calculi of lambda-conversion. Annals of Mathematics Studies, No. 6, Princeton Univ. Press, Princeton 1941

[3] McCarthy, J. Recursive functions of symbolic expressions and their computation by machine. Comm. ACM, 3,4 (April 1960) 184-195

[4] Backus, J. Reduction languages and variable-free programming. IBM Research Report RJ1010, Yorktown Heights, NY, April 7, 1972

[5] Landin, P.J. The mechanical evaluation of expressions. The Computer Journal, Jan. 1964, 308-320.

[6] Falkoff, A.D., and Iverson, K.E. APL/360 User's Manual. IBM, 1968

APPENDIX: PRIMITIVES AND PROGRAMMING EXAMPLES FOR RED LANGUAGES

Function descriptions

In this appendix we describe functions as in the following example. Let $\underline{a}\varepsilon A$, $\underline{x},\underline{x}1,\ldots,\underline{x}n\varepsilon C$. Then

OP: $(\underline{a},\underline{x}) \rightarrow \underline{a}$; $(\underline{x}1) \rightarrow \emptyset$; $(\underline{x}1,\underline{x}2,\ldots,\underline{x}n) \rightarrow \underline{x}2$; $\perp$

describes the function α represented by the atom OP, where $\alpha\underline{x}=\underline{a}$ when $\underline{x}$ is a pair whose first component is the atom $\underline{a}$, and $\alpha\underline{x}=\emptyset$ when $\underline{x}$ is a sequence of length 1, and $\alpha\underline{x}$ is the second element of $\underline{x}$ when $\underline{x}$ is a sequence of length 2 or more which does not satisfy the first clause, and $\alpha\underline{x}=\perp$ in all other cases.

Red/1 primitive operators

The following operators comprise one proposal for a medium-length list of medium-power primitive operators. Some operators can be defined in terms of others in the list. All can be defined in terms of the primitives of Red/0 [see below]. Red/1 is the Red language with these operators and with all other atoms representing the function Ω. If a primitive operator is meta, the function represented by the primitive with its parameters is given. Thus, for the meta operator WHILE, the regular operator (WHILE,$\underline{p}$,$\underline{f}$) is described.

META OPERATORS

(1) NF [N Functions] SNF=(ϕ,(AA,APPLY),TAIL,DISTR)

(NF,$\underline{f}1,\ldots,\underline{f}n$): $\perp \rightarrow \perp$; $\underline{x} \rightarrow (\langle\underline{f}1,\underline{x}\rangle,\ldots,\langle\underline{f}n,\underline{x}\rangle)$

(2) CN [Condition]

(CN,$\underline{p}$,$\underline{f}$,$\underline{g}$): $\underline{x} \rightarrow \langle\underline{f},\underline{x}\rangle$ when $\mu\langle\underline{p},\underline{x}\rangle=T$;

$\underline{x} \rightarrow \langle\underline{g},\underline{x}\rangle$ when $\mu\langle\underline{p},\underline{x}\rangle=F$;

undefined when $\mu\langle\underline{p},\underline{x}\rangle$ is undefined; $\perp$

(3) C [Constant] SC=(ϕ,2,1)

(C,$\underline{x}$): $\underline{y} \rightarrow \underline{x}$

WHILE ≡ ≪CN, APPLY∘[2∘1, 2], APPLY∘[1, APPLY∘[3∘1, 2], 2]≫

(4) WHILE, [While]

(WHILE,p,f): x → <(WHILE,p,f),<f,x>> when μ<p,x>=T;

x → x when μ<p,x>=F;

undefined when μ<p,x> is undefined; ⊥

(5) AA [Apply to All]

δAA = (φ, (CN, (NULL, 2), (C, φ), (UN, [(APPLY, [(2,1),(1,2)]), (APPLY, [1, (TAIL, 2)])])))

(AA,f): (y1,...,yn) → (<f,y1>,...,<f,yn>); ⊥

(6) AL [Apply to Left element] ≈ (UN, [(f, 1), 2], SEP)

(AL,f): (y1,..,yn) → (<f,y1>,...,yn); ⊥

(7) AR [Apply to Right element] ≈ (ROTL, (AL, f), ROTR)

(AR,f): (y1,...,yn) → (y1,...,<f,yn>); ⊥

(8) INSERT [Insert]

δINSERT = (φ, (CN, (NULL, TAIL, 2), (1, 2) (APPLY, [(2,1), [(1,2), (APPLY, [1, (TAIL, 2)])]])))

(INSERT,f): ~~(y1,y2) → <f,(y1,y2)>;~~ (y1) → (y1);

(y1,y2,..,yn) → <f,(y1,<(INSERT,f),(y2,..,yn)>)>; ⊥

[Would like: (INSERT, f, identity): φ → identity; ...]

REGULAR OPERATORS

(1) 1,2,... [Selectors, i=1,2,...] i>1 ⇒ i ≡ (1, ROTL^(i-1))

LENGTH: φ → 0; (x1, ..., xn) → n; ⊥

i: (x1,...,xn) → xi when i∈{1,...,n}; ⊥

(2) TAIL [Tail] (2, SEP)

TAIL: (x1) → Ø; (x1,x2,...,xn) → (x2,...,xn); ⊥

(3) EQ [Equal] ⇒ [not necessary by def of subexpression on p.10]

EQ: (x,⊥) → ⊥; (⊥,x) → ⊥; (x,x) → T when x does not have ⊥ as a subexpression;

(x,y) → F when neither x nor y have ⊥ as a subexpression; ⊥

(4) APPLY [Apply] δAPPLY = (1, FST1, DBL, ROTL)

APPLY: (x,y) → <x,y>; ⊥

Need predicate ATOM: a → T; ⊥ → ⊥; F

(5) NULL [Null]

NULL: Ø → T; ⊥ → ⊥; F

(6) AND [And] (CN 1 2 (C,F)) provided ∠2, ans> ∈ {T, F}

AND: (T,T) → T; (x,y) → F when x,y∈{T,F}; ⊥

(7) NOT [Not] δNOT = (CN, φ, (C,F), (C,T))

NOT: T → F; F → T; ⊥

(8) ROTL [ROTate Left]

ROTL: (x1) → (x1); (x1,x2,...,xn) → (x2,...,xn,x1); ⊥

(9) ROTR [ROTate Right] Note ROTR ≡ REVERSE∘ROTL∘REVERSE

ROTR: (x1) → (x1); (x1,...,xn,y) → (y,x1,...,xn); ⊥

(10) DISTL [DISTribute from the Left] δDISTL = (APPLY, [[(C,AA), [(C,NF), [(C,C),1]], (C,φ)]], 2])

DISTL: (x,(y1,...,yn)) → ((x,y1),...,(x,yn)); ⊥

(11) DISTR [DISTribute from the Right] δDISTR = ((AA,ROTR), DISTL, ROTL)

DISTR: ((x1,...,xn),y) → ((x1,y),...,(xn,y)); ⊥

(12) SEP [SEParate] δSEP = [1, TAIL]

SEP: (x1) → (x1,Ø); (x1,x2,...,xn) → (x1,(x2,...,xn)); ⊥

(13) ~~UN~~ [JoiN] ~~[UNion]~~

JN ~~UN~~: (x1,Ø) → (x1); (x1,(x2,...,xn)) → (x1,...,xn); ⊥

(14) TRANS [TRANSpose]

TRANS: (x1,...,xn) → (y1,...,ym) when each xi is a sequence of length m, and each yj is the sequence of jth elements of the x's; ⊥ $y_j = \langle (AA, (1, TAIL^{j-1})), (x_1, \cdots, x_n) \rangle$

Other primitive Red/1 operators

In addition to the above rather scanty list, one would like the usual arithmetic operators. The binary arithmetic operators would be defined as usual for pairs of special atoms called "numbers". One might also want operators which could form and decompose atoms as in the Red/1 of the report [4]. There are probably many other useful and powerful primitives which one might want to include.

Shorthand notation for NF

The modifier NF is so basic to the variable-free programming techniques needed in Red/1 that we use a special shorthand for it. We observe that many such shorthand notations can be used and then translated into the corresponding Red/1 form before evaluation. For

any sequence of the form (NF,f1,...,fn) we shall write instead:

[f1,...,fn]

Thus [f1,...,fn] is a regular operator, and when applied to x yields (<f1,x>,...,<fn,x>) , except when x=⊥ it yields ⊥.

Red/1 programming examples

Several definitions of example operators are given. The effect of these defined operators is then asserted in the same form as for the primitives. The reader may check the validity of these assertions by using the transition rules for Red languages with definitions plus the above definitions of Red/1 primitive operators.

We begin with two examples of definitions which are impossible to make in most languages: the definitions of two modifiers. [To obtain the equivalent effect in LISP, one must define function-valued or function-argumented functions.] Actually, LISP uses a scheme syntactically resembling Red. For example:

$$\text{mapcar}[(x_1\ x_2\ \cdots\ x_n);f] = (f[x_1]\ f[x_2]\ \cdots\ f[x_n])$$

Example

We define a meta operator, or modifier, BU [Binary operator to Unary] which creates a unary operator from a binary operator plus one parameter. Thus (BU,EQ,5) is a regular operator which yields T when its operand is 5.

δBU = (Ø,APPLY,[(2,1),[(3,1),2]])

The effect of this definition for two parameters is:

(BU,f,x): y → <f,(x,y)>

For example we have the following transitions:

<(BU,EQ,A),B> → <BU,((BU,EQ,A),B)> → ... → <Ø,<APPLY,(EQ,(A,B))>>

→ ... → <Ø,<EQ,(A,B)>> → ... → F

Example

EE [Element-by-Element] is a modifier which converts a binary operator into the corresponding element-by-element operator for vectors. Thus (EE,EQ) is "vector equal"; its result for ((A,B,C),(A,B,D)) is (T,T,F). (EE,+) is vector addition. The distinction between (EE,EQ) and EQ is difficult to make in some languages.

δEE = (Ø,APPLY,[[(C,AA),(2,1)],(TRANS,2)])

def ee f = (aa f)∘trans

The effect of this definition is the following for one parameter:

(EE,f): x → <(AA,f),<TRANS,x>>

If x=((y1,...,yn),(z1,...,zn)) then the result is:

(<f,(y1,z1)>,...,<f,(yn,zn)>)

Example

We now give two definitions for "reverse", one iterative [REVI], one recursive [REVR].

δREVI = (1,(WHILE,p,f),[(C,Ø),Ø])

where p = (NOT,NULL,2) and f = [(UN,[(1,2),1]),(TAIL,2)]

This gives the following [we use → for one or more transitions]:

<REVI,(A,B,C)> → <1,<(WHILE,p,f)),(Ø,(A,B,C))>>

→ <1,<(WHILE,p,f),<f,(Ø,(A,B,C))>>>

→ <1,<(WHILE,p,f),((A),(B,C))>>

→ <1,<(WHILE,p,f),((B,A),(C))>>

→ <1,<(WHILE,p,f),((C,B,A),Ø)>> → <1,((C,B,A),Ø)> → (C,B,A)

δREVR = (CN,p,Ø,f)

where p=(NULL,TAIL) and f=(ROTL,UN,[1,(REVR,TAIL)])

This gives the following:

<REVR,(A,B,C)> → <f,(A,B,C)> → <ROTL,<UN,(A,<REVR,(B,C)>)>>

→ <ROTL,<UN,(A,<ROTL,<UN,(B,<REVR,(C)>)>>)>>

→ <ROTL,<UN,(A,<ROTL,<UN,(B,<Ø,(C)>)>>)>>

→ <ROTL,<UN,(A,<ROTL,<UN,(B,(C))>>)>>

→ <ROTL,<UN,(A,(C,B))>>

→ <ROTL,(A,C,B)> → (C,B,A)

Example

We give definitions for inner product, IP, and matrix multiply, MM. IP gives the sum of the element-by-element products of a pair of vectors. MM gives the product of a pair of compatible matrices, where each matrix is the sequence of its rows.

δIP = ((INSERT,+),(EE,×))

δMM = ((AA,(AA,IP)),(AA,DISTL),DISTR,(AR,TRANS))

The first operator of MM, (AR,TRANS), transposes the right matrix, thus the new right matrix is the set of columns of the old. The next operator pairs the set of columns with each row of the left matrix; the next transforms each of these pairs: in each one, it pairs the particular row of the left matrix with every column of the right one. Thus each element of the result so far is a set of pairs of row-and-column. The last applies inner product to each of these pairs in each set. The result is the product matrix. Note that this operator does not prescribe as much unnecessary order in the calculation as most programs do; that the process is unusually transparent to analysis; and that it is the composition of operations that involve more than a single "element" or pair of elements. The availability of appropriate modifiers makes it possible to put large-scale operators together to form larger-scale ones.

Primitive operators for Red/0

We give a list of eight "primitive" primitive operators for Red/0. They are sufficient, I believe, to define those of Red/1. If two operators to form and decompose atoms are also provided [see [4]], then arithmetic operators on "number" atoms could also be defined. All of the eight operators are regular.

(1) DBL: $\perp \to \perp$; $\underline{x} \to (\underline{x},\underline{x})$

(2) 1: $(\underline{x}1,\ldots,\underline{x}n) \to \underline{x}1$; $\perp$ [same as for Red/1]

(3) ROTL [same as for Red/1] $(x_1 x_2 \cdots x_n) \to (x_2 \cdots x_n x_1)$

(4) SEP " $(x_1,x_2,\cdots x_n) \to (x_1,(x_2,\cdots,x_n)$

(5) UN " $(x_1,(x_2,\cdots,x_n)) \to (x_1,x_2,\cdots x_n)$ UN = SEP^{-1}

(6) COND: $(T,(\underline{x},\underline{y})) \to \underline{x}$; $(F,(\underline{x},\underline{y})) \to \underline{y}$; $\perp$

(7) EQAT: $(\underline{x},\perp) \to \perp$; $(\perp,\underline{x}) \to \perp$; $(\underline{a},\underline{a}) \to T$ when $\underline{a} \in A$; $(\underline{x},\underline{y}) \to F$; $\perp$

(8) FST1: $((\underline{x},\underline{y}),(\underline{z},\underline{w})) \to (\langle \underline{y},\underline{z} \rangle,\underline{w})$; $\perp$

It is rather cumbersome to define some of the Red/1 operators in terms of the above operators, but there is not enough space here to list an appropriate set of intermediate operators.

---

* Let $(a,b) = ((a_1,\cdots,a_m),(b_1,\cdots b_p))$ where $a_i = (a_{i1},\cdots,a_{in})$ and $b_j = (b_{j1},\cdots,b_{jp})$.
Then (a,b) $-(AR,TRANS)\to$ $(a,\hat{b})$ where $\hat{b}_{ij} = b_{ji}$ [i.e. $b = ((b_{11},\cdots,b_{1n}),\cdots,(b_{p1},\cdots,b_{pn}))$]
$-DISTR\to$ $((a_1,\hat{b}),\cdots,(a_m,\hat{b}))$
$-(AA,DISTL)\to$ $(((a_1,\hat{b}_1),\cdots,(a_1,\hat{b}_p)),\cdots,((a_m,\hat{b}_1),\cdots,(a_m,\hat{b}_p)))$
$-(AA,(AA,IP))\to$ c where $c_{ij} = \langle IP,(a_i,\hat{b}_j)\rangle$, $1 \le i \le m$, $1 \le j \le p$.