PROGRAMMING IN THE META-LANGUAGE: A TUTORIAL

Dines Bjørner

Abstract:

This paper provides an informal introduction to the "art" of abstractly specifying software architectures using the *VDM* meta-language*. A formal treatment of the semantics, as well as a BNF-like concrete syntax, of a large subset of the meta-language is given in [Jones 78a] following this paper.

* *colloquially known as: META-IV*

CONTENTS

PART I: PRELUDE

Section 0 frames the subject, and gives an extensively annotated exam=
ple. This example illustrates many of the important aspects of the me=
ta-language. The particular abstraction choices made are, however, not commented upon, nor explicitly singled out.

Section 1 forms an overview of the various meta-language constructs. Although this primer is almost 200 pages (long), the meta-language, as it transpires from section 1, is not 'big'! Certain meta-language no=
tions are considered so common, or simple, that they are not treated beyond section 1. We think here, in particular on sections 1.5 - 1.7.

0. INTRODUCTION

This tutorial teaches you the meta-language. The primary aim is to render you fluent in the notation and its meaning, both as a reader and as a writer. That is: both as a user of software architectures abstractly specified by other people, and as a producer of such documents oneself. The secondary aim is to make you use the meta-language as per its intentions, i.e. in good style. We wish that you be able to produce suitable software abstractions. Thus we emphasize giving a rather complete, yet informal coverage of the syntax & semantics of the meta-language. We refrain however, from presenting a comprehensive introduction to the kind of abstraction principles and techniques which the meta-language is especially designed to facilitate and express. The tutorial contains an extensive set of examples. A careful study of these is intended to give you a rather complete overview of the fundamentals of abstract modeling -- as we see this activity. See also [Bjørner 78c].

The meta-language, as already mentioned in the introduction to this volume, evolved in the course of documenting a readable, denotational semantics of a PL/I subset [Bekić 74]. It has subsequently, in addition to denotational semantics definitions of other languages, been applied to similarly abstracted, formal definitions of relational data base systems [Hansal 76, Nilsson 76], components of operating systems and their command & (job) control languages [Bjørner 78a, Madsen 77], as well as more applications oriented software. These all exemplify, or imply, large complexes of software, with many, and intricate facilities. Individual examples of this tutorial, will, however, also illustrate applications to basic algorithmic functions.

The examples of this tutorial can be grouped according to either of two criteria: free-standing examples (illustrating an isolated software concept); versus examples strung together over several sections (usually illustrating systems software). The latter can (furthermore) be grouped according to the kind of systems software being abstracted. The following is a reference guide to these: *

Programming Language Constructs:

Examples: 30-34, 36-38, 40, 42-43, 47, 49-62, 64-68.

(Most of these examples are mere transcriptions and annotations of parts of the mini-language definition of appendix III [Jones 78a].)

---

* *See pages 213-217 for a complete, annotated index to all examples.*

File System Concepts
Examples: 3-8, 12-13, 17-18, 20-25, 27-28, 35.
(Operating or File System) Catalogues/Directories
Examples: 19, 29, 63.

Usually major sections will start with a large example illustrating all the main notions introduced by the section. The examples interspersed in the text outlining the individual language constructs are usually far simpler than the more encompassing introductory example. Finally, many sections are trailed by yet additional, larger examples. If you find difficulties in comprehending the introductory examples you may wish to skip these intially. If you find difficulty in understanding the interspersed examples this tutorial will have failed!

(We mention, in passing, that no examples will be given of concurrent system architectures: the subject meta-language was not designed to cater for this [sadly neglected] area.) Finally some examples will be of a rather formal, or schematized nature; not illustrating 'practical' notions, but paraphrasing principal, or 'theoretical', aspects of the meta-language.

We stress here, as was stressed in the introduction to this volume, that the meta-language is to be used, not for solving algorithmic problems (on a computer), but for specifying, in an implementation-independent way, the architecture (or models) of software. Instead of using informmal English mixed with technical jargon, we offer you a very-high-level 'programming' language. We do not offer an interpreter or compiler for this meta-language. And we have absolutely no intention of ever wasting our time trying to mechanize this meta-language. We wish, as we have done in the past, and as we intend to continue doing in the future, to further develop the notation and to express notions in ways for which no mechanical interpreter system can ever be provided. Given a terse and readable abstract model of some software item, and as e.g. expressed in this meta-language, we see it as the foremost and almost solely distinguishing task of programming to carefully turn the abstraction, in stages of, at most semi-automated, development, into an efficient realization. Once such an implementation has been reached we let the entire, possibly annotated, documentation: the abstract model, as well as all intermediate development stages, serve as the only documentation of the realized software.

To give you a flavor of the meta-language, but not really the kind of software we primarily intend it aimed at, we now give you a rather comprehensive example.

Example 0

We will present this example completely formally: first the abstract model, then the annotations. Other than these annotations, which represent a mere reading of the formulae, we do not here explain the formulae. Thus the example is brought to give you, from the very start, a capsule view of core aspects of the meta-language; as well as an idea about what a model is: its parts, their purpose and interrelations.

Semantic Domains

1 *GROCER* :: *SHELVES STORE CASH CATALOGUE*
2 *SHELVES* = *Wno* $\overset{m}{\rightarrow}$ N_1
3 *STORE* = *Wno* $\overset{m}{\rightarrow}$ N_1
4 *CASH* = N_0
5 *CATALOGUE* = *Wno* $\overset{m}{\rightarrow}$ *Description*
6 *Description* :: *Price Minimum Maximum Size*
7 *Price* = N_1
8 *Minimum* = N_1
9 *Maximum* = N_1
10 *Size* = N_1

-- annotations:

The model is concerned with rather self-contained fragments of a grocery: its inventory, cash and catalogue subsystem. The model describes a domain of such groceries and exemplifies a few of the manipulations that groceries are subject to: customer purchases, and inventory/cash control.

The reader is, throughout this primer, well-adviced in reading the annotations with a finger following the formulae and trying, otherwise, to establish the connection!

1 A grocery is here selectively abstracted by abstractions of its shelves and store, i.e. inventory, its cash register, and its catalogue.

2 The shelves display a finite, non-zero number of items of a finite variety of merchandise. Merchandise presently being abstracted by ware number codes.

3 In the store-room is similarly kept a finite, non-zero number of boxed quantities of items of a finite selection of wares.

4 The cash register is simply abstracted by the cash it contains.

5 The grocers' catalogue lists a description of each sort of merchandise.

6 Such a description here consists of the unit (item) sales price; the minimum and maximum (lower- & upper-bound) numbers of items, of the described merchandise, which ought, respectively may be placed on the shelves; and finally the size of a stored box, measured in terms of the number of merchandise items it contains.

7 Prices are measured in integer (positive number) units of currency.

etc..

Comments

The above description 'read' the formulae as describing a grocery. The formulae, in fact, defines a whole domain (i.e. class) of such.

The formulae (1-10), and (12-15) below, constitute an abstract syntax (abstract syntaxes). Each line (1, 2, ..., 10) is an abstract syntax rule. The rules all have their left-hand sides being identifiers. The right-hand sides are so-called domain-expressions.

The described domains are said to constitute the semantic domains. Semantic domains are "what the whole thing is about". Syntactic domains, described below in formulae 12-14, are (just the) objects which denote manipulations of semantic objects.

Well-Formedness Constraints

```
11.0 is-wf-GROCER(mk-GROCER(shelves, store, cash, catalogue))=
  .1     (dom store ⊆ dom shelves ⊆ dom catalogue)
  .2    ∧(∀wno ∈ dom catalogue)
  .3        (let mk-Description(price,min,max,size) = catalogue(wno) in
  .4        (0 ≤ size ≤ max - min)
  .5          ∧((wno ∈ dom shelves)
  .6              ⊃(let items = shelves(wno) in
  .7                ((wno ∈ dom store) → (min ≤ items ≤ max),
  .8                 T                 → items ≤ max)))
```

-- annotations :

The domain descriptions captured the essence of how we abstractly view groceries, but the defined domains contain objects, i.e. groceries, which do not satisfy natural constraints:

11.0 For a grocery (which consists of shelves, a store room, the cash register, and a catalogue) to be well-formed, the following constraints must be satisfied:

11.1 There cannot be merchandise in the store room which is not also displayed on the shelves; and any merchandise on the shelves must be described in the catalogue. Furthermore:

11.2 For each type of merchandise described in the catalogue,

11.3 look-up the price, minimum & maximum shelf-, and box size quantities for that ware in the catalogue:

11.4 Now the maximum must be higher than or equal to the minimum, and their difference (in that order) must be lower than or equal to the box size. (This is a pragmatic constraint. It permits the update of shelves with the full contents of boxes, without violating (min, max) constraints.)

11.5 If, in addition, the ware is also, actually displayed on a shelf, then:

11.6 Let us call the number of items of that ware on the shelves for *items*. Now:

11. If the ware additionally is further stored in the back room, then the number of items on the shelves must actually fall between the

minimum, lower and maximum, upper bounds; otherwise

11.8 there can be no more items on the shelves than is maximally permitted.

Comment:

The *is-wf-* function is defined relative to some named domain, here the class of all *GROCER*ies. In the following we shall understand by *GROCER* the class of those objects which satisfy the predicate (11). In fact, any manipulation, i.e. any transformation of, or process on, a grocery, shall leave us a(nother) grocery also satisfying the well-formedness constraint. Thus the predicate (11) is also seen as forming a major part of the invariant according to which we 'program' our manipulations -- whether on the abstract, or concrete level.

Syntactic Domains

12 *Transaction* = *Purchase* | *Control* | ...
13 *Purchase* :: Wno^+
14 *Control* :: *Wno-set*
15 ...

-- annotations:

The grocer sees a customer purchase, his own daily check on the availability of certain wares, etc., as transactions. These are operations on the grocery. Syntactically speaking:

12 A transaction is either a customer purchase, a clerk control, or something else - presently undefined!

13 A customer purchase is presented at the check-out counter as a sequence of not necessarily distinct wares. In this sequence groups of items of the same kind need not occur together.

14 A store clerk control consists of a set of distinct ware types (for which their inventory amounts are asked).

Comment:

Observe that only the last parenthesized phrase above invoked semantic notions. That is: we have totally separated syntactic descriptions from those of their semantics.

Dynamic Consistency Constraints

```
16.0  is-well-formed-purchase(purchase, grocery) =
  .1      (let mk-Purchase(wl)                    = purchase
  .2           mk-GROCER(shelves, , , )           = grocery in
  .3       let mini-shelf = make-shelf(wl)([]) in
  .4       (dom mini-shelf ⊆ dom shelves)
  .5      ∧(∀wno ∈ dom mini-shelf)
  .6           (mini-shelf(wno) ≤ shelves(wno)))
```

-- annotations:

The customer can select merchandise only from the shelves, and in quantities bounded by what is displayed:

16.0 For the combination: a purchase and a grocery to be consistent, we must therefore require that

16.1 the ware list of which the purchase is made up, and

16.2 the shelves (from which the wares were selected) satisfy the constraints given in lines 16.4-16.6.

16.3 To express those constraints let us compute for each ware type, in the purchase, the number of times it occurs in the purchase. Since this corresponds, in the way we abstracted shelves, to an object of *SHELVES*, we call it *mini-shelf*. That is: *mini-shelf* displays, as do shelves, a functional association between ware codes and numbers of (*shelves*, *mini-shelf*) items of that ware. The function *make-shelf*, which takes the ware-list of the purchase and computes this association, is activated with an empty shelf. The function definition is given below (17).

Now to the constraints themselves:

16.4 The kind of wares purchased must form a (not necessarily proper) subset of the merchandise displayed on the shelves.

And:

16.5 For each ware purchased

16.6 the number of items purchased must be less than or equal to the number of wares of that kind on the shelves.

Comment

The abstraction given is not as elegant and transparent as we would have hoped. Thus we find the introduction of the *make-shelf* auxiliary function somewhat of a resort to an operational description of the constraint.

Auxiliary Function

```
17.0  make-shelf(wl)(shelf)=
  .1      if wl = <>
  .2        then shelf
  .3        else (let wno = hd wl in
  .4               let shelf' = ((wno ∈ dom shelf)
  .5                                → shelf+[wno→(shelf(wno)+1)],
  .6                              T→ shelf∪[wno→1]) in
  .7               make-shelf(tl wl)(shelf'))
      type: Wno+ → (SHELVES → SHELVES)
```

-- annotations :

The function takes, as arguments, a tuple (or list) of wares and a shelf (really: an object of *SHELVES*) and yields, as result, a *SHELVES* object. The *type* clause expresses this. The form after the *type* colon is a so-called domain expression. The function is recursively defined, cf. line 17.7. It performs its function by treating each item (*wno*) in the ware list separately: 17.3-17.6. That is: by 'chopping' off the list from its front, or *head*, leaving treatment of the rest, i.e. *tail*, of the list, till another recursion (17.7), passing it the shelves object (*shelf'*) so far computed. Thus:

17.1 If the (*tail* of the) ware list is the null tuple,

17.2 then the accumulated *shelf* computed up till now is delivered as the result.

17.3 Otherwise: Let us call the first item of the remaining ware list for *wno*.

17.4 And then update the *shelf* so far computed. If the item under inspection (being "tallied") has already occurred in the ware list,

17.5 then the updated *shelf* has one added to the number of items, of the same kind, so far purchased.

17.6 Otherwise, this is a first occurrence of this particular ware, and the *shelf* is augmented by the initialization of its tally count to *1*.

Semantic Functions

```
18.0  Elab-Purchase(mk-Purchase(wl), grocer)=
  .1     if wl = <>
  .2        then grocer
  .3        else (let mk-GROCER(shs,sto,cash,cat)     = grocer,
  .4                  mk-Description(p,min,max,size) = cat(hd wl),
  .5                  wno                            = hd wl      in
  .6              let cash' = cash+p,
  .7                  (shs',sto') =
  .8                      (let items = shs(wno),
  .9                           stored= ((wno ∈ dom sto) → sto(wno),
  .10                                    T             → 0) in
  .11                      if ((items=min)∧(stored>0))
  .12                         then
  .13                            (shs + [wno→items-1+size],
  .14                            ((stored=1) → sto\{wno},
  .15                             T          → sto+[wno→stored-1]))
  .16                         else
  .17                            (((items=1) → shs\{wno },
  .18                             T          → shs+[wno→items-1]),
  .19                             sto)) in
  .20              Elab-Purchase  (mk-Purchase(tl wl),
  .21                             mk-GROCER(shs',sto',cash',cat)))
     type:  Purchase GROCER ~→ GROCER
```

-- annotations:

To purchase a ware is to take a grocery store and deliver another. In

the resulting grocery store the shelf count for the purchased ware (wno) is diminished by 1 (18.13 & 18.18), and if the shelf display goes under minimum and there are supplies stored (18.11) then the shelf is replenished by *size* items (18.13). To the cash register is added the purchase price of the ware (18.6).

The *Purchase Elab*oration, as did the above auxiliary function, works by updating the grocery store, once completely for each item in the purchase (18.3-18.19). That is:

18.1 If the purchase has been completely serviced,

18.2 then the result is the 'input' grocery.

18.17 If the shelf display of the purchased ware would go to zero, cognizance of this ware is deleted from the shelf. Likewise:

18.14 If by replenishing the shelves from the store the supplies go to zero, this ware is deleted from the store room.

18.20 The rest of the purchase is elaborated.

18.21 with the updated grocery.

...

19.0 *Elab-Control(mk-Control(wno),grocer)=*
Tabulate(wno,grocer)([])

type: Control GROCER → (wno $\overrightarrow{m}$ N_0*)*

-- annotations :

To control the grocery for the quantities available of each of a given set of ware categories is to tabulate these, starting with an empty table.

The tabulate function is auxiliary. The resulting table associates to each ware category the possibly zero quantity on hand, on the shelves as well as in the supply.

Auxiliary Function

```
20.0  Tabulate(wnos,grocer)(table)=
  .1      if wnos={}
  .2         then table
  .3         else (let mk-GROCER(shs,sto, ,cat) = grocer,
  .4                   wno ∈ wnos in
  .5               let items = ((wno ∈ dom shs) → shs(wno),
  .6                            T               → 0),
  .7                   stored= ((wno ∈ dom sto) → sto(wno),
  .8                            T               → 0),
  .9                   size  = s-Size(cat(wno)) in
  .10              let sum   = items + (stored*size)  in
  .11              Tabulate(wnos\{wno},grocer)(table∪[wno→sum])

      type: Wno-set GROCER  →  ((Wno ⇀m N0)  →  (Wno ⇀m N0))
```

-- annotations :

are left to the reader!

1. OVERVIEW OF META-LANGUAGE

In this section we briefly survey the meta-language.The overview of sections 1.1-1.3 emphasizes only syntactic aspects. The aim of sections 1.1-1.3 is to give you a rather comprehensive feeling for the composition and extent of the meta-language.

1.1 Abstract Data Types

The elementary data types (except for booleans) of the meta-language are not fixed. For a suggestion of suitable base types we refer to section 13.

The composite, very-high level abstraction-facilitating, data types are:

Sets:

Domains:	*A-set*
Constructors	{...}
Operations	∪, ∩, ∖, ⊆, ⊂, ∈, *card*, *power*, =, ≠

Tuples:

Domains:	A^*, A^+
Constructors:	<...>
Operations:	*hd*, *tl*, *len*, *ind*, *elems*, ⌢, [.], *conc*, =, ≠

Maps:

Domains:	$A \xrightarrow[m]{} B$
Constructors:	[..→..]
Operations:	*dom*, *rng*, (.), ∪, +, ∖, †, =, ≠

Functions:

Domains:	$A \to B$, $A \xrightarrow{\sim} B$
Constructors:	λ...•...
Operations:	(.)

Trees:

Domains:	$A :: B_1\ B_2\ \ldots\ B_n,\quad (C_1\ C_2\ \ldots\ C_n)$
Constructors:	$mk\text{-}A,\ mk,\ (\ldots)$
Operations:	$s\text{-}B_i,\ =,\ \neq$

The 'domains' lines above exemplify the expression of domains of objects of the subject type. A, B, B_i and C_i are given (or defined) domains. The constructors hint at the following typical object construction expressions:

$\{a_1, a_2, \ldots, a_n\}$, $\quad \{\ F(d)\ |\ P(d)\ \}$,

$\langle a_1, a_2, \ldots, a_n \rangle$, $\quad \langle\ F(i)\ |\ P(i) \wedge m \leq i \leq n\ \rangle$

$[a_1 \rightarrow b_1, a_2 \rightarrow b_2, \ldots, a_n \rightarrow b_n]$, $\quad [\ F(o) \rightarrow G(o)\ |\ P(o)]$

representing respectively explicit & implicit set, tuple and map object denoting expressions. Also:

$$\lambda id.clause$$

abstracts the meta-language expression or statement *clause* into the function of *id* that *clause* is. Finally:

$mk\text{-}A(b_1, b_2, \ldots, b_n)$, $\quad mk(c_1, c_2, \ldots, c_n)$ and $(c_1, c_2, \ldots, c_n)$

denote trees.

1.2 Combinators: Statements and Structured Expressions

Declaration:	*dcl Var := expr type D*,
Assignment:	*Var := expr*,
Identity:	*I* (null statement)
Contents expressions:	*c Var*
Conditional Clauses:	*if expr then clause*$_1$ *else clause*$_2$ $(expr_1 \rightarrow clause_1,$ $expr_2 \rightarrow clause_2,$ $\ldots$ $expr_n \rightarrow clause_n)$

is the classical McCarthy conditional clause, with:

cases $expr_0$:
  ($expr_1$ → $clause_1$,
   $expr_2$ → $clause_2$,
   ...
   $expr_n$ → $clause_n$)

being a variant of the Hoare/Wirth cases clause.

while expr do stmt,

for all id ∈ *set do stmt*, and:

for i = m to n do stmt

are the (conventional) iteration statements. Compound statements use the semicolon operator: ($stmt_1$; $stmt_2$;...; $stmt_n$) -- where *stmt* are statements. Statement- & expression blocks may occur anywhere a statement, respectively expression, may occur:

(*let def = expr in clause*),

and:

(*let def : expr; clause*)

illustrate the two block forms: the former basically of the applicative kind, the latter of the imperative kind.

Function definitions are written:

$fid(id_1, id_2, \ldots, id_n)$=
  clause
  type: $D_1\ D_2\ \ldots\ D_n \rightarrow D$

-- or in some such form. The language has no gotos, but provides both imperative and applicative variants of a phrase-structured, block-oriented, un-labelled *exit* mechanism:

(*trap exit(id) with F(id) in clause*)

and:

(*tixe* [*G(o)* → *F(o)* | *P(o)*] *in clause*)

with:

exit, *exit(expr)*

being the exit causing constructs.

1.3 Abstract Syntax

Abstract syntaxes are used in defining named domains of objects:

A_0	=	$B_1 \mid B_2 \mid \ldots \mid B_n$	union
A_1	=	$B\text{-}set$	sets
A_2	=	B^*	tuples
A_3	=	B^+	tuples
A_4	=	$B \xrightarrow[m]{} C$	maps
A_5	=	$B \rightarrow C$	functions
A_6	=	$B \xrightarrow{\sim} C$	partial functions
A_7	::	$B_1\ B_2\ \ldots\ B_n$	trees

are typical rules. Objects, *o*, in the domains defined by some such rule satisfy:

$is\text{-}A_i(o)$.

1.4 Logic

Only one elementary data type is assumed:

BOOL

consisting of the two truth valued objects:

true, *false*

to which the following non-commutative operations apply:

$\wedge$, $\vee$, $\supset$ (and, or, implication)

as well as:

$\sim$, $\equiv$ (negation, identity).

Quantified Expressions

Predicates asserting truth properties:

(1) $(\forall o \in set)(P(o))$
(2) $(\exists o \in set)(P(o))$
(3) $(\exists! \ o \in set)(P(o))$

read as follows:

(1) for all objects [in the set] the predicate (P) holds.
(2) there exists an object [in the set] for which the predicate (P) holds.
(3) there exists a unique object [in the set] for which the predicate (P) holds.

1.5 Descriptor Expression

This subsection will be the only place in which the descriptor expression is treated.

$$(\iota o)(P(o)), \quad (\Delta o)(P(o))$$

ι (iota) and Δ (delta) are offered as alternate forms of representing the descriptor operator. The forms above denote (and read):

the unique object satisfying the predicate (P).

1.6 Undefined & Erroneous Situations

The form:

undefined

is offered as a notation expressing some further unspecified undefined object. Applying operations outside their domain is (likewise) said to yield undefined results. Similarly for descriptor expressions.

The form:

error

is offered as a notation for stating situations for which there are no clear definitions.

In this primer we consistently use *undefined* to denote undefined (non-state) objects, i.e. in basically applicative contexts. We consistently use *error* to denote undefined state transitions, i.e. in basically imperative contexts. Thus we consider *undefined* to be an expression; *error*, a statement!

1.7 User-Defined Identifiers

The meta-language sets no spelling standard for identifiers naming objects (defined functions, variables, parameters, etc.) or domains (as in abstract syntaxes).

The following outlines the conventions basically followed in this primer.

Identifiers are either single Greek letters or sequences of one or more Roman letters and Arabic digits. Identifiers might contain proper infix hyphens, and possibly be decorated with (simple) subscripts (digits and letters) and/or single, double, triple, ..., (superscript) primes.

The choice between Greek letters and other identifier forms is basically governed by these informal, enumerative rules:

(So-called) State Domain Names:	Σ, or Ξ
State Object Names:	σ, respectively ξ
(So-called) Environment Domain Names:	*ENV*
Environment Object Names:	*env*, or ρ
(So-called) Continuation Domain Names	*C*
Continuation Object Names:	θ

The criterium on when to use upper and lower case letters is basically this:

(So-called) Semantic Domain Names:

Sequences of one or usually more UPPER CASE LETTERS, possibly trailed by a digit.

(So-called) Syntactic Domain Names:

UPPER case letter followed by one or more lower case letters, possibly trailed by a digit.

Object Names:

Lower case counterparts of corresponding domain names, usually amounting to prefixes of such, and quite often decorated.

Assignable Variable Names:

-- in this primer usually written in *script* and underlined with tildes. When variable is global, first letter is upper case; otherwise lower-case.

Other conventions are:

Function Names:

Elaboration functions primarily applicable to objects of domain *Xyz* have names:

elab-Xyz	(for imperatively stated) respectively:
E-Xyz	for (applicatively expressed) elaboration functions;
int-Xyz	(for imperatively stated) respectively:
I-Xyz	for (applicatively expressed) interpretation functions;
eval-Xyz	(for imperatively stated) respectively:
V-Xyz	for (applicatively expressed) evaluation functions.

(Elaboration is a term comprising both interpretation and evaluation. Interpretation implies elaboration of constructs basically for the sake of their side-effects. EValuation implies elaboration of constructs basically for the sake of values explicitly yielded.)

Other Function Names:

Let *A* be some domain name, then

is-A, s-A, mk-A, is-wf-A,

are reserved names. *is-A* was explained in section 1.3. *s-A* and *mk-A* in 1.1. *is-wf-A* is to be the name for any static context

condition predicate function applicable to all *A* objects and yielding *true* only for those *A* objects one actually intends the definition of *A* to cover! *s-A* and *mk-A* may not actually be defined by some abstract syntax. The above reservation then intends to prevent confusion.

1.8 Structure of Tutorial

The structure of this primer is as follows.

In part II we cover data types: sets, tuples, maps, trees, functions and the notion of abstract syntax. Respectively sections 2,3,4,5,6 & 7. The language features covered enable us to express domains, objects and primitive operations on these -- in particular operator/operand expressions. Examples of earlier sections will necessarily employ meta-language notions only formally introduced in later sections. Insofar as this is the case we rely on your good-will and patience, attempting, on our behalf, however, to keep such uses to a reasonable minimum.

In part III we cover language constructs such as variables: their declaration, assignment and contents access; structure expressions and statements; blocks, and the *exit* mechanism. With these meta-language features we are now able to express composite transformations, respectively state compound processes, on objects, respectively state variables. The examples of this part take on a flavor distinct from that of the examples of part II. There the examples were explicitly tied to the individual subjects being covered. Explicit examples using the constructs formally introduced in part III are given already in part II. Instead the examples of part III tend to be rather more comprehensive, illustrating several aspects of abstract modeling simultaneously.

In part IV we wrap up the story on function definitions and abstract models.

This part is not comprehensive. It relies on the subject being already, albeit partially covered in parts II & III. Part V ties up loose ends concerning elementary data types.

...

In addition to the above contents survey we advise the reader to carefully study the contents listing in order to ascertain the logical structure of this primer.

PART II DOMAINS, OBJECTS & OPERATIONS

Function definitions describe transformations on data objects. As such the defined functions apply to a domain of objects and yield objects of the range. The specific transformation is expressed, in terms of forms (constructors) denoting objects, operators and structured combinators denoting operations on objects.

The above paragraph serves the purpose of delineating the three cornerstones of this part: the story on meta-language means of defining domains of objects, representing constructed objects and operations on objects. Most sections will be structured accordingly. These will have basically three subsections. One will outline and exemplify how domains are defined, i.e. represented by means of so-called logical type- (or, as we shall prefer to call them, domain) expressions. Another will outline and exemplify how instances of objects of the domain are constructed. Finally a third will outline and exemplify expressions formed around operators denoting operations on objects. This last is split into two subsections: one on primitive operations, the other on combinators.

Each section dealing with a composite abstract data type will be introduced by examples whose purpose is to capture the essence of the objects under discussion, thus motivating you right into the rather formally structured parts. And each section will be concluded with larger examples.

In bringing examples we intend to illustrate basic principles of abstract modeling using the meta-language. The examples take care to use the meta-language in "good style".

[In this way you will also be introduced to representational- & operational abstraction; to applicative- & imperative programming; and, in a few places, to configurational- & hierarchical abstraction.]

2 *SET*s

Example 1:

The situation which we wish to abstractly define is perhaps rather 'childish'. We are given a set of classes, each class consisting of a set of students. Let students be abstracted by their school registration code. Let the domain of all such codes be *Student*. Then:

Class = Student-set

is an abstract syntax rule whose left-hand side is an identifier, and whose right-hand side is a domain expression. It describes a class as being a set of (distinct) students. The equation gives the name *Class* to a domain, and this domain is defined, by *Student-set*, to be the domain of finite subsets of *Student*.

Let suitably decorated s's and c's denote respectively students and classes, i.e.:

is-Student(s)
is-Class(c)

Now:

$$\{s_1, s_2, \ldots, s_n\}$$

is a set constructor expression. It denotes a class.

To test whether a given student, s, attends a given class, c, we write:

$$s \in c$$

-- which we read: s is a member of c. If s is not a member of c, then the expression is *false*, otherwise *true*. If we call the function that tests class recordings for *is-in-class* then:

is-in-class(s,c) = s ∈ c

is a function definition. To compute the students attending two classes, c_1 and c_2, we write:

$c1 \cap c2$

-- which we read: the intersection of c_1 and c_2, i.e.: the set of students common to both c_1 and c_2. If no students attend both c_1 and c_2, then:

$$c_1 \cap c_2 = \{\}$$

their intersection is empty. The students following either c_1 or c_2 or both is denoted by:

$$c_1 \cup c_2$$

-- which we read: the union of c_1 and c_2. To record that a class, c, is diminished by the departure of a student, s, we write:

$$c \setminus \{s\} \qquad \text{or} \qquad c - \{s\}$$

-- which we read as: the complement of s with respect to $\{s\}$, or -- which might be more to your liking -- the set c subtracted $\{s\}$. We use the notation $\setminus$ and - interchangeably, although not in this primer, where $\setminus$ is preferred. To record that a class, c, is augmented by the entry of a student, s, we write:

$$c \cup \{s\}$$

The number of students in a class is:

$$\underline{card}\ c$$

-- for cardinality.

2.1 Defining Domains of *SET* Objects

Let *A* be the name of a class of objects i.e. a domain. To define the class of objects which are finite sets of *A* objects (i.e. finite subsets of *A*), we use the domain expression operator *-set: A-set.*

Example 2:

The domain whose objects are finite sets of integers, i.e. finite subsets of *INTG*, is definable as:

INTG-set

Call this domain *IS*. Then:

IS = INTG-set

is an abstract syntax rule formally defining this name. Then the domain whose objects are finite sets of (potentially overlapping) finite sets of integers is definable as:

IS-set, or: *(INTG-set)-set*

Example 3:

Let *R* represent the domain whose objects are abstractions of what, in a file system, is otherwise thought of as records. If files of such a system are, or can be, unordered, finite collections of distinct records, then:

R-set

is an abstraction of the class of files. The domain *R-set*, if *R* is an infinite class, is also infinite -- its objects are, however, all finite sets. The form *R-set* is a domain expression.

The file system thus modeled may seem a bit contrived. That is: you may never wish to design such a system. Note that at this stage the system has no notion of sequential files or keyed records. The examples, when expanded in subsequent sections will, however, appear more 'realistic'.

2.2 Representing Instances of *SET* Objects

2.2.1 Explicit Enumeration

Given distinct objects a_1, a_2, ..., a_n which are all *A* objects, the expression:

$$\{a_1, a_2, \ldots, a_n\}$$

is said to be an explicit enumeration of a set, which denotes an object of *A-set*.

Empty Set

-- is denoted by:

$$\{\}$$

2.2.2 Implicit Enumeration

Given a function F: $D \rightarrow A$, where *D* denotes some logical type (i.e. an arbitrary domain); and a predicate P: $D \rightarrow BOOL$, the expression:

$$\{F(d) \mid P(d)\}$$

is said to be an implicit set enumeration, and then denotes an object of *A-set*. Since *A* could be any logical type expression the above describes how arbitrary sets may be represented. The implicit set constructor expression can be read as: The set of objects *F(d)* such that the predicate *P(d)* holds. Thus we read | as 'such that!

Examples 4-5-6:

The constructor expression:

$$\{\{1,3,5,7,9\},\{2,5,11,13,15\},\ldots,\{\},\ldots\{2,1,8\}\}$$

denotes an object in *(INTG-set)-set* whose element sets, in this example, are not disjoint.

...

Let r_1, r_2, ..., r_n denote distinct objects of *R*, i.e. be abstractions of records, then:

$$\{r_1, r_2, \ldots, r_n\}$$

is (an abstraction of) a file, i.e. an object in *R-set*.

Let F_h be a total function from records into records, i.e.:
type: F_h: $R \rightarrow R$, and let *f* denote a file, e.g. the above, then:

$$\{ F_h(r) \mid r \in f \}$$

denotes a file derived from *f* having each of the records of *f* processed by F_h.

2.3 *SET* Operations:

The following special *SET* operations are defined:

∪	union
∩	intersection
∖,-	complement, difference, subtraction (two forms provided)
⊂	proper subset
⊆	subset
power	powerset
∈	membership
card	cardinality
union	distributed union

Each of these will now be individually and quite informally explained.

It is assumed below that *set*, *set1*, *set2* denote sets and *setsets* a set of sets.

set1∪*set2* denotes the set of those objects which are either in *set1*, or in *set2*, or in both.

set1∩*set2* denotes the set of objects which are both in *set1* and *set2*.

set1∖*set2* denotes the set of objects which are in *set1* but not in *set2*.

set1⊂*set2* denotes (the *BOOL*ean truth value) *true* if all members of *set1* are in *set2* and there is at least one member of *set2* not in *set1*, otherwise *false*.

set1⊆*set2* denotes (the *BOOL*ean truth value) *true* if all members of *set1* are in *set2*, otherwise *false*.

power set denotes the set of all finite subsets of *set*.

obj∈*set* denotes (the *BOOL*ean truth value) *true* if the object denoted by *obj* is a member of *set*, otherwise *false*.

card set denotes the (*N*atural) number of members of *set*. Read as *card*inality.

union setsets denotes the set consisting of all the objects of all the sets being elements of *setsets*.

The set operations applied to anything other that sets are *undefined*.

Examples 7:

Let f denote a file, i.e. be in *R-set*; let r and r_i denote records (i.e. both be in R). The operator-operand expressions and constructs:

(1) $f \cup \{r\}$
(2) $f \setminus \{r\}$
(3) $r \in f$
(4) *card* f
(5) $(f \setminus \{r\}) \cup \{F_h(r)\}$
(6) *let* $r_i \in f$
(7) *let* $r_i \in f$ *be s.t.* $P(r)$

express typical set manipulations. These pure expressions could be used to provide a model of the following typical operations on files: (1) writing a (most likely new) record to a file; (2) deleting a record -- most likely, but necessarily, contained in a file -- from that file; (3) asking whether a given record is in a file; (4) asking for the number of records in a file; (5) updating a record in a file to a new record, i.e. replacing it -- with the possibility that it might not already be in the file, in which case (1~5); (6) reading an arbitrary record from a file assumed not to be empty; and (7) reading an almost arbitrary record, namely one further satisfying the property expressed by the predicate function P.

Let *f1* and *f2* denote files, i.e. both be in *R-set*, then:

(8) $f1 \cap f2$
(9) $f1 \subseteq f2$
(10) $f1 \subset f2$
(11) $f1 = f2$
(12) $f1 \neq f2$

might be reasonable abstractions of the following, slightly hypothetical, operations among files: (8) collecting, into a file, the records common to two files; (9) asking whether all records of one file are contained in another file; (10) -- and, in addition to (9) -- asking whether some records of the latter are not in the former; (11) asking whether two files are identical; or (12) different!

The expression:

(13) *f1* ∪ *f2*

is a generalization of (1), in that {*r*} there denotes a singleton file, i.e. a file of exactly one record. (13) can be understood as the merge of the records of two files. The type of the object denoted by (13) is a file.

2.4 *SET*-oriented Combinators

The following combinators are applicable to *SET* objects:

let obj ∈ *Set*
let obj ∈ *Set be s.t. P(obj)*

which you have already seen applications of, and:

for all obj ∈ *set do S(obj)*

The *let* clauses occur in expression- or statement blocks:

(let obj ∈ *Set ... in*
C(obj))

where C represents either an expression or a statement. The *for all* clause is a statement, and so must *S* be.

For the general meaning of *let* clauses we refer to section 10.1. The specific *let* clauses shown above bind the identifier *obj* to an arbitrary, etc., member of the set, or domain, *Set*, anywhere in C where *obj* occurs free.

Note that *Set* in *let* clauses may either be an expression denoting a finite set; or a domain-expression, potentially denoting infinite sets.

In the *for-all* statement *set* must however be restricted to denote a finite set, in particular: the expression: *set* must not be a domain expression.

The meaning of the *for-all* clause is given here, but more systematically repeated in section 9.2.5. Let *set* denote the finite set whose n objects we may arbitrarily name: $id_1, id_2, \ldots, id_n$, where no id_j is free in $S(id)$; then:

$$//\{S(id_1), S(id_2), \ldots, S(id_n)\}$$

is a sufficient transliteration of the *for-all* statement into a quasi-parallel 'compound' statement, each of whose statements, $S(id)$, arises from $S(obj)$ by replacing all free occurrences of *obj* in $S(obj)$ by *id*. For the meaning of compound statements rely on your intuition, or look it up in section 10.4. Since the set element naming was arbitrary one can permute the above statements arbitrarily.

Example 8:

The constructor expression:

$$\{ F_h(r) \mid r \in f \}$$

is an applicative expression -- provided F_h does not rely on any state component. An imperative analogue can be achieved by means of the *for-all* statement:

```
del file := {} type R-set
(for all r ∈ f do
    file := c file ∪ {F_h(r)};
c file)
```

The above can be seen to be a process-oriented abstraction of parts of a file manipulation system architecture. The global variable *file* is initialized to the empty file, {}, of no records; and fixed to contain only finite sets of distinct records *(type R-set)*. File now denotes a constant reference, to an *R-set* object, i.e. an object in *ref R-set*; with *cfile* denoting the dynamic contents at that reference, i.e. value of the file variable.

2.5 Further Examples

To sharpen your understanding of set manipulations we now bring further examples.

Example 9:

The problem which we wish to abstractly define is that of recording equivalence classes. A set (e.g. *sas*) consisting of disjoint sets (e.g. *as1, as2,..., asn*) of (e.g. *A*) objects is said to define (or, which is the same, to 'realize') an equivalence relation. In fact we call the member sets (*as1,...*) for equivalence classes. The set of equivalence classes thus is a partitioning of the union of all the *(A)* objects of the equivalence classes. Given one partitioning and an *(A)* object, supposed to be a member of an equivalence class, we wish to inquire whether it is indeed in some equivalence class of that partitioning. (We call the predicate which tests for this *isRecorded.*) As another subproblem we wish, given a partitioning and two 'recorded' *(A-)* objects, *(a1,a2)* to generate a new partitioning as follows: If the two *A* objects are recorded in the same equivalence class, then the result partitioning is the same as the argument partitioning. If the two *A* objects are recorded in distinct equivalence classes, then the result partitioning is as the argument partitioning except for the collapse (union) into one memberset of the two sets of which *a1* and *a2* are respective members. (We call the new equivalence class generator function for *enter*).

```
1          EQ  =  B-set
           B   =  A-set
```

i.e.:

```
           EQ  =  (A-set)-set
```

```
2          is-wf-EQ(sas)=
  .1          (∀as1,as2∈sas)(as1≠as2 ⊃ as1∩as2={})
```

```
3          isRecorded(a,sas)=              isRecorded(a,sas)=
  .1          (a ∈ union sas)                 (∃ as ∈ sas)(a ∈ as)
```

where we gave two versions of *isRecorded*.

```
4          enter((a1,a2),sas)=
  .1          {as | as∈sas ∧ {a1,a2}∩as={}}
  .2        ∪ { as1∪as2 | (as1∈sas)∧(a1∈as1)∧(as2∈sas)∧(a2∈as2)}
```

```
5          equiv((a1,a2),sas)=
  .1          (∃as∈sas)(a1∈as ∧ a2∈as)
```

```
type: is-wf-EQ:    EQ          →  BOOL
      isRecorded:  A    EQ     →  BOOL
      enter:       (A A) EQ    ~→ EQ
      equiv:       (A A) EQ    →  BOOL
```

-- annotations:

1. *EQ* names a domain of objects. These are finite sets of *B* objects, with the latter being finite sets of *A* objects -- hence an *EQ* object is a set of sets of *A* objects.

2. For such a set to be a partitioning, i.e. to be well-formed as per the intentions of defining the domain *EQ*, no two distinct members (of *sas*) may have any *A* objects in common; or, which is the same, all membersets must be disjoint.

3. For an *A* object, *a*, to be recorded, shall mean that the object (*a*) is in some memberset, i.e. (3.3) that there is a memberset, *as*, of which it is an element.

4. The two lines (4.1 & 4.2) express what was stipulated above: the result partitioning is as the argument partitionings (4.1), except (4.2) -- etc.

The *type* clauses of the functions define the set of the input arguments, to the left of the arrows, and those of the results, to the right of the arrows. The arrows express that the defined functions are (indeed) (possibly partial ($\tilde{\rightarrow}$)) functions -- from the argument domains into the result domains.

Example 10:

As in the previous example, we deal with objects:

$$S = (A\text{-}set)\text{-}set$$

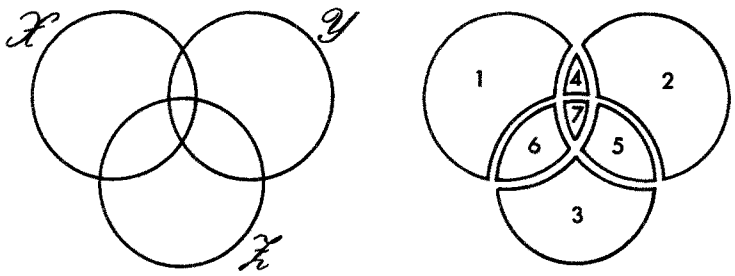
-- but now not subject to any restrictions. To check whether an S object, *sas*, is a partitioning we apply:

$$isDecomposed(sas) =$$
$$(\forall as1, as2 \in sas)(as1 \neq as2 \supset as1 \cap as2 = \{\})$$
$$type: S \rightarrow BOOL$$

To decompose an S object, *sas*, into the coarsest, i.e. smallest, having fewest elements, partioning of *sas*, we apply *decompose(sas)*.

A possibly incomplete definition of a coarsest partitioning, given a set, *sas*, of potentially overlapping sets, goes as follows: Let *sas'* be the result of *decompose(sas)*. For all *a1* and *a2* that are members of member-sets of *sas*, *a1* and *a2* are in the same member-set of *sas'* if-and-only-if (iff) for any subset, $\{as1, as2, \ldots, asn\}$ of *sas*, *a1* is in all *asi*, for $i \in \{1:n\}$, iff *a2* is in all *asi* (for $i \in \{1:n\}$); and only such *a*'s are in member-sets of *sas'* which are similarly in *sas*, i.e.: *union sas* = *union sas'*.

Sometimes a picture is worth quite a few words:



To the left is some *sas* ($\{X,Y,Z\}$), to the right is its corresponding *sas'*: the digits 1-7 index the corresponding forms in the right-hand side expression below:

$$decompose(\{X,Y,Z\}) = \{X\setminus(Y\cup Z), Y\setminus(X\cup Z), Z\setminus(Y\cup X),$$
$$(X\cap Y)\setminus Z, (Y\cap Z)\setminus X, (X\cap Z)\setminus Y,$$
$$X\cap Y\cap Z\}$$

Here is a formal definition of the function:

```
decompose(sas)=
   if (∃as1,as2∈sas)((as1≠as2)∧((as1∩as2)≠{}))
     then (let as1,as2 ∈ sas be s.t. as1∩as2≠{};
           decompose({as1\as2,as2\as1,as1∩as2}∪sas\{as1,as2})
     else sas

type: S → S
```

3. *TUPLE*s

Example 11:

The problem with which we wish to illustrate, before going into a more systematic coverage, the meta-language abstract data type of *TUPLE*s, is perhaps not a very abstract one. We are to compute the first k rows of the PASCAL triangle; informally:

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
```

etc.. So we define two functions: one computing the i'th row; another putting the first k rows together. The second function invokes the first.

type: *row*: $N_1 \rightarrow N_1^{+}$
type: *tri*: $N_1 \rightarrow N_1^{++}$

row takes a positive, non-zero integer, i.e. a natural number larger then *1*, and produces a tuple of such numbers; *tri* takes a non-zero natural number and produces a tuple of as many tuples of natural numbers. N_1^{+} stands for the domain of tuples of natural numbers; N_1^{++} for tuples of tuples of these numbers. An implicit way of specifying *row* and *tri* would e.g. define their *pre-* and *post*-conditions, i.e. the conditions that input (alone) must satisfy, respectively the conditions that input and output (together) must satisfy:

type: *pre-row*: $N_1 \rightarrow BOOL$
type: *post-row*: $N_1\ N_1^{+} \rightarrow BOOL$
type: *pre-tri*: $N_1 \rightarrow BOOL$
type: *post-tri*: $N_1\ N_1^{++} \rightarrow BOOL$

In particular:

```
pre-row(i) = true

post-row(i,r)=
     cases i: (1  →  (r = <1>),
               2  →  (r = <1,1>),
               T  →  (let rim1 = row(i-1) in
                      (l r = i) ∧ (r[1]= 1 = r[l r]) ∧
                      (∀j ∈ {2:l r-1})
                          (r[j] = rim1[j-1]+rim1[j])))

pre-tri(k) = true

post-tri(k,rr)=
     (l rr = k)
     ∧(∀i ∈ {1:k})(rr[i] = row(i))
```

In the notation of the meta-language, *tri(6)* would be presented as:

<<1>,<1,1>,<1,2,1>,<1,3,3,1>,<1,4,6,4,1>,<1,5,10,10,5,1>>

Giving the above implicit definitions of *row* and *tri* almost amounts to giving their explicit counterpart. Instead of doing this, let us "read" the *pre*- and *post*-'s:

The *pre* of *row* is always *true* (for $i>0$). The *post* of *row* says: if $i=1$ then the resulting row r is just the tuple of length *1* whose only member is a *1*. If $i=2$ then r is a *2*-tuple both of whose elements are *1*'s. In general, the length of r is i, its first and last elements are both *1*'s, and, for $i>2$, the elements of r, i.e. of *row(i)*, are related to elements of *row(i-1)* as follows: let j be any index into r exclusive of the first and last, then the j'th element of r is the sum of the $j-1$'st and j'th elements of *row(i-1)*.

The computation specified below computes *tri(k)* by imperative means:

```
(dcl Tri  := <<1>,<1,1>> type N1++,
     Rowi := <1>         type N1+;
 for i=2 to k do
    (let rim1 : (c Tri)[i-1];
     for j=2 to len rim1 do
         Rowi := (c Rowi)~<rim1[j-1]+rim1[j]>;
     Rowi := (c Rowi)~<1>;
     Tri := (c Tri)~<c Rowi>);
 return(c Tri))
```

3.1 Defining Domains of *TUPLE* Objects

Let *A* be the name of a class of objects. To define the class of objects all of whose members are finite length tuples whose elements are *A* objects we use either of the domain expression operators: * or +.

A^* denotes the infinite domain of finite length tuples all of whose elements are in *A*. The *0*-length tuple, <> , is in A^* -- for any *A!*

A^+ denotes the infinite domain of finite non-zero length *TUPLE*s, all of whose elements are in *A*. Thus <> is not in A^+.

Example 12:

Let *G* denote a class of objects, i.e. a domain, abstracting the fields of a record, with records consisting now of a finite number of such ordered fields, then:

$$G^*$$

is a domain expression abstracting the class of records. Call this *R*, i.e.:

$$R = G^*$$

Now let files be ordered, finite length sequences of one or more records. Then:

$$R^+ \text{ or } (G^*)^+$$

are equivalent domain expressions abstracting the class of files. In the latter expression, the parentheses are solely used for grouping -- you could, of course, omit parentheses here: G^{*+}. If you so wish you can likewise name the file domain:

$$Ft = R^+$$

In this example we did not retain the view that files consisted of collections of distinct, unordered records. Had we done so:

$$Fs = R\text{-}set$$
$$R = G^*$$

or:

$$Fs = (G^*)\text{-}set, \text{ or: } Fs = G^*\text{-}set$$

would have been applicable abstractions.

3.2 Representing Instances of *TUPLE* Objects

3.2.1 Explicit Enumeration

Given not necessarily distinct objects a_1, a_2, ..., a_n which are all in A, the expression:

$$<a_1, a_2, \ldots, a_n>$$

denotes an n-tuple, i.e. an object of, or in: A^*, A^+ (for $n>0$).

Empty Tuple

-- is denoted by:

<>

3.2.2. Implicit Enumeration - Part 1

Given a function F: *INTG* → *D* the expression:

< *F(i)* | *1≤i≤n* >

denotes an *n* tuple -- as above! So does:

< *F(i)* | *i∈{1:n}* >, and

< *G(d)* | *i∈{m:m+n-1}* >, etcetera,

where *G*: *D* → *A*. Thus we shall not be too particular about the form of the "type-building" predicate as long as it is clear, to the readers of your formulae, what the size, i.e. length, of your tuple will be. Since *G(d)* is independent of *i* the *n* tuple consists of identical *G(d)* elements.

Examples 13:

Let g_1, g_2, ..., g_m, and g_{ij} for varying i,j denote fields of a record, i.e. all objects in *G*, then:

$$\langle g_1, g_2, \ldots, g_m \rangle,$$

$$\langle\langle g_{11}, g_{12}, \ldots, g_{1x}\rangle, \langle g_{21}, g_{22}, \ldots, g_{2y}\rangle, \ldots, \langle g_{n1}, g_{n2}, \ldots, g_{nz}\rangle\rangle$$

are constructor expressions denoting objects in G^*, respectively R^+ That is: abstractions of (instances of) records, respectively files!

Let F_g, F_h be total functions from fields to fields, respectively records to records, i.e.: *type:* F_g: $G \rightarrow G$, *type:* F_h: $R \rightarrow R$; and let f,r denote a file, respectively a record; then:

$$< F_g(r[i]) \mid 1 \leq i \leq \underline{l}\, r >$$
$$< F_h(f[j]) \mid 1 \leq j \leq \underline{l}\, f >$$

denotes a record, respectively a file, derived from r, respectively f, having all its fields, respectively records, processed -- in any order -- by F_g, respectively F_h. Although to be dealt with more systematically below $r[i]$ denotes the i'th field of record r, with $\underline{l}\, r$ denoting the length, in terms of not necessarily distinct, fields, of record r.

3.2.3 Element Ordering

Sets are unordered collections of objects. Tuple elements are ordered. In:

$$<a_1, a_2, \ldots, a_n>$$

a_1 is the *1*st element, a_2 the *2*nd,..., and a_n the nth element. In:

$$< F(i) \mid i \in \{m:m+n-1\}>$$

the *1*st element is $F(m)$, the *2*nd element is $F(m+1)$,..., and $F(m+k)$ for $0 \leq k \leq n-1$ is the $k+1$-st element. In:

$$< G(d) \mid i \in \{m:m+n-1\}>$$

all elements are alike, and are $G(d)$ -- with n of them!

By:

$$< obj \mid obj \in set >$$

we mean a *card set* tuple of all distinct members of the assumed finite set set -- occurring in arbitrary order.

In general the form of an implicit tuple building constructor expression is:

$$< F(i) \mid O(i) \wedge P(i) >$$

where $O(i)$ is an ordering predicate indicating, mainly through the natural ordering of integers i, the order of those elements $F(i)$, for i, for which the more general predicate $P(i)$ holds:

3.2.4 Implicit Enumeration - Part 2:

Let P: $INTG \rightarrow BOOL$ be a (total function) predicate then the expression:

$$< F(i) \mid i \in intgset \wedge P(i) >$$

denotes an l tuple :

$$< F(i_1), F(i_2), \ldots, F(i_l) >$$

where :

$$\begin{aligned} & \{i_1, i_2, \ldots, i_l\} \subseteq intgset \\ \wedge\ & i_1 < i_2 < \ldots < i_l \\ \wedge\ & P(i_1) \wedge P(i_2) \wedge \ldots \wedge P(i_l) \\ \wedge\ & (\sim \exists i \in intgset \setminus \{i_1, i_2, \ldots, i_l\})(P(i)) \end{aligned}$$

In words: The ordering of the resulting tuple elements follow the natural ordering of the *INTeGers*. The *length* of the tuple will be the *cardinality* of that subset of the finite integer set, *intgset*, for which P is satisfied.

Example 14-15:

Although not illustrative of abstraction techniques, as we intend them, the following examples might help drive home the idea of implicit tuple constructions:

(1) Let *fib* denote the so-called fibonacci function, *type: fib:* $N_0 \rightarrow N_1$ where N_0, N_1 denote the sets of natural numbers larger than or equal to *0*, respectively *1*. In particular think of: *fib(0)=1*, *fib(1)=1*, with *fib(i)* for $i \geq 1$ being *fib(i-2)+fib(i-1)*. Then:

$$< fib(i) \mid (m \leq i \leq n) \wedge \underline{odd}\ fib(i) >$$

denotes a tuple of those of the *m*'th, *m+1*'st,..., up to, and including the *n*'th fibonacci number provided these are not even numbers, and $m, n \geq 0$! For *m=3* and *n=11* you would get:

$$<3,5,13,21,55,89>$$

i.e. the tuple of the 3rd, 4th, 6th, 7th, 9th and 10'th fibonacci number.

The type of the above expressions is: N_1^*

(2) The following pure expression constructs the pascal triangle between rows *1* and *k*:

```
< row(i) | 1 ≤ i ≤ k
         ∧ cases i
              (1  → row(i) = <1>,
               2  → row(i) = <1,1>,
               T  → row(i) = <1>
                             ~< (row(i-1))[j]+
                                (row(i-1))[j+1]
                              | 1 ≤ j ≤ len row(i-1))>
                             ~<1>) >
```

You are invited to compare this formulation with the impure expression-block given as the last item of example 11 above.

3.3 *TUPLE* Operations

The following special *TUPLE* operations are provided:

	[pairwise] concatenation,
h, *hd*	head,
t, *tl*	tail, (two forms provided)
l, *len*	length,
elems	elements,
ind	indices,
[i], *(i)*	index, selection, (two forms provided)
conc	distributive concatenation (of tuple of tuples),
+	replace.

Each of these operators will now be individually, and quite informally explained. It is assumed below that *tuple*, *tuple1*, *tuple2* denoted *TUPLE*s and *tuplelist* a *TUPLE* of *TUPLE*s.

tuple1~*tuple2* denotes the *TUPLE* whose first *l tuple1* elements are those of *tuple1* in the given order; whose last *l tuple2* elements are those of *tuple2*, and in that order; and whose length is exactly the sum of the lengths of *tuple1* and *tuple2*.

h tuple denotes the 1st element of *tuple*. Taking the head of an empty tuple is *undefined*.

t tuple denotes the *tuple* of all but the first element of *tuple* and otherwise in the same order as the elements of *tuple*. Taking the tail of an empty tuple is *undefined*.

l tuple denotes the length of the tuple, i.e. the Number of not necessarily distinct elements of *tuple*.

elems tuple denotes the *SET* of elements contained in the *tuple*.

ind tuple	denotes the *SET* of *N*atural numbers which are the indices of *tuple*.
tuple[*i*]	denotes the *i*th element of *tuple* provided *i* is a *N*atural number larger than *0* and less than or equal to the length of *tuple*.
conc tuplelist	denotes the *TUPLE* of elements of the tuples which are the immediate elements of *tuplelist* and in the order otherwise given in the unravelled or de-bracketed, element tuples.
tuple+[*i*→*o*]	denotes the tuple which is as *tuple* -- only the *i*th object is not *tuple*[*i*] but *o*. If $1>i>\underline{l}$ *tuple* the expression is *undefined.*

This last operation is (presently) only defined for a right operand singleton map (in $\acute{N}_1 \vec{m} OBJ$).

The tuple operations applied to objects other than tuples are *undefined*.

Examples 16-17:

Let *VAL* denote the class of values obtained during, i.e. as a result of partial, evaluations of expressions of some language, and let *STACK* be an abstraction of stacks of these values.

$$STACK = VAL^*$$

can then be considered a not too abstract, yet sufficiently implementation-independent, model of the domain of stacks. Let $stk \in STACK$, $v \in VAL$ then:

(1) $\langle v \rangle \frown st$

(2) $\underline{h}\ st$, $\underline{hd}\ st$

(3) $\underline{t}\ st$, $\underline{tl}\ st$

(4) $\langle\rangle$

are correspondingly reasonable abstractions of the following typical operations on or facts about stacks: (1) pushing a new object "on top" of, or into the stack; (2) reading the top of the stack; (3) the resulting stack after a pop operation -- with (4) not abstracting any operation but only brought here to exemplify the empty, unused stack .

...

Reverting now to our examples around file systems, let possibly suitably decorated f, r and g denote files, records and fields respectively, i.e. objects in F, R and G, then the expressions:

(5)	$f \frown \langle r \rangle$	$r \frown \langle g \rangle$
(6)	$\underline{h}\, f$, $\underline{hd}\, f$	$\underline{h}\, r$, $\underline{hd}\, r$
(7)	$f[i]$	$r[j]$
(8)	$f1 \frown f2$	$r1 \frown r2$
(9)	$f+[i \rightarrow r]$	$r+[j \rightarrow g]$
(10)	$\underline{conc}\, f$	
(11)	$\underline{l}\, f$, $\underline{len}\, f$	$\underline{l}\, r$, $\underline{len}\, r$

are reasonable abstractions of the following typical operations on files (left column) [and records (right column)]: (5) writing a record [field] to a sequential, even serial file [record]; (6) & (7) reading the first, respectively the i'th record [j'th field] of a file [record]; (8) chaining two files [records] together to form a new file [record]; (9) updating the i'th record [j'th field] of a file [record]; (10) chaining all the records of a file together to form a record (!); and (11) inquiring about the current size of a file [record] in terms of its number of records [fields].

3.4 *TUPLE*-oriented Combinators

The following combinator:

$$\underline{for}\ i{=}m\ \underline{to}\ n\ \underline{do}\ S(i)$$

is provided for the imperative handling of primarily lists of objects. In particular, let t denote a tuple, then:

for i=1 *to* *len* t *do* $S(t[i])$

is a typical statement. The integer valued bounds m and n must be statically determined. S is any statement. See the doubly nested example of a use of the above iterative statement given as the last item of example 11.

The meaning is given by the transcription:

$S(m);\ S(m+1);\ldots;\ S(n)$

where $S(j)$ arises from $S(i)$ by substituting all free occurrences of i in $S(i)$ by j. Since j is a constant no collision with other free variables in $S(i)$ will occur.

Example 18:

The constructor expression:

$$< F_h(f[i]) \mid 1 \leq i \leq \mathit{len}\ f >$$

is an applicative expression -- provided F_h does not rely on any state component. An imperative analog can be achieved by means of the *for*-*to*-*do* linearly iterative loop statement:

```
dcl file := <> type R*
(for i=1 to len f do
    file := (c file)^< F_h(f[i])>;
 c file)
```

The above can be seen to be a process-oriented abstraction of parts of a file handling system specification. The global variable *file* is initialized to the null tuple, <> , of no records; and fixed to contain only finite length sequences of not necessarily distinct records (*type* R*). *file* now denotes a constant reference to an R* object, i.e. an object of *ref* R*; with *c file* denoting the dynamic contents at that reference.

4 *MAPs*

Example 19:

As a prelude to a systematic treatment of the subject of maps we are in this introductory example going to illustrate the abstraction of what could be an operating system directory in terms of recursively nested maps. In this, perhaps hypothetical operating system users are partially ordered, i.e. hierarchically structured -- and this is reflected in the design of the directory. With each user is associated a qualified name consisting of an ordered sequence of zero one or more resource identifiers. The overall system directory is a directory and a directory consists of a finite non-empty set of uniquely identified resources. A resource is e.g. either a line printer, a card reader, a display terminal, a file identifier or it is a directory.

Abstractly modeling we write:

1 $DIR = Rid \xrightarrow[m]{} RES$
2 $RES = LP \mid CR \mid DT \mid Fid \mid DIR \mid \ldots$

The resource- and file-identifiers are considered to be further unanalyzed, e.g. elementary, objects:

$$Rid = \ldots, \quad Fid = \ldots$$

We also do not have here bother to specify what line printers, card readers and display terminals are:

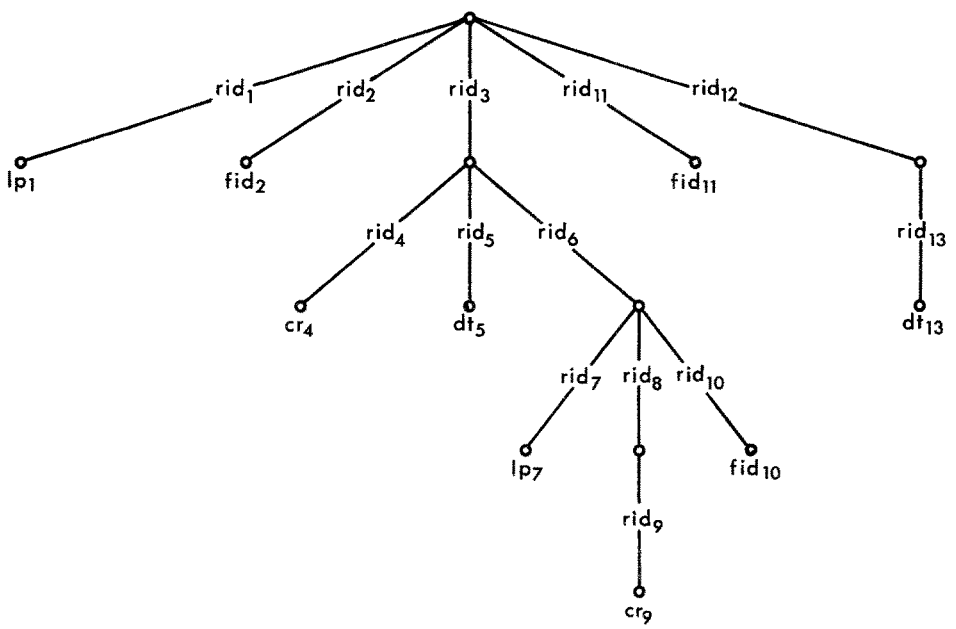
$$LP = \ldots, \quad CR = \ldots, \quad DT = \ldots$$

The above two equations (1,2) are examples of abstract syntax rules. The first defines the domain of finite domain maps from resource identifiers to resources, and names it *DIR*; the latter defines the domain of resources to be the union (|) domain of the domains listed to the right, i.e. in the definiens. Thus a particular resource, i.e. an object of the domain *RES*, is either an object in *LP* (i.e. a line printer), or -- etc.

If by suitably decorated lower case letter sequences analogous to the above domain names we mean to identify objects of respective classes, then the following is an expression of the meta-language denoting an object in *DIR*, i.e. an abstract "snapshot" of a directory:

$$
\begin{aligned}
&[rid_1 \rightarrow lp_1,\\
&\ rid_2 \rightarrow fid_2,\\
&\ rid_3 \rightarrow [rid_4 \rightarrow cr_4,\\
&\qquad rid_5 \rightarrow dt_5,\\
&\qquad rid_6 \rightarrow [rid_7 \rightarrow lp_7,\\
&\qquad\quad rid_8 \rightarrow [rid_9 \rightarrow cr_9],\\
&\qquad\quad rid_{10} \rightarrow fid_{10}]],\\
&\ rid_{11} \rightarrow fid_{11},\\
&\ rid_{12} \rightarrow [rid_{13} \rightarrow dt_{13}]]
\end{aligned}
$$

which we could informally picture:



In the above hierarchical structure the labeling of edges (by *rid*s) is usually understood to void any 'meaning' you may think that the left-to-right ordering of edges have.

Observe how, from the system-directory the qualified name $<rid_3, rid_6, rid_7>$ designates lp_7. Qualified names designating directories are: <>, $<rid_3>$, $<rid_{12}>$, $<rid_3, rid_6>$, $<rid_3, rid_6, rid_8>$, and these are the names associated with users.

Resources associated with existing users may be allocated or freed: Let *sys-dir* stand for the system directory, *qn* for a name known to designate a direcotry in *sys-dir*, and let *rid* and *res* denote respectively the name by which the resource *res* is to be known in the directory designated by *qn*. The following function, when invoked by: *catalog(sys-dir,qn,rid,res)* accomplishes this:

```
3.0    catalog(dir,q,id,r)=
 .1       if q=<>
 .2          then dir ∪ [id→r]
 .3          else dir + [h q→catalog(dir(h q),t q,id,r)]
          type: DIR Rid* Rid RES ~→ DIR
```

The reader is encouraged to apply this function to the above example.

The line: "*dir* ∪ [*id*→*r*]" reads: "join, or merge, as a new entry, the association of *id* to *r*, to the map *dir*". The last line reads: "*h q*, which is an *Rid*, designates a directory, *dir(h q)* , embedded in *dir*. Replace this directory, *dir(h q)* , with the one -- *catalog(dir(h q),t q,id,r)* -- obtained by cataloging the resource *r* named *id* in the subdirectory of *dir(h q)* designated by *t q* !".

Now it may be that the original *qn* did not designate a directory in *sys-dir*. Then the above function would go awry. So here is an improved version:

```
4.0  catalog(dir,q,id,r)=
 .1     if q=<>
 .2        then if id ∈ dom dir
 .3                then error
 .4                else dir ∪ [id→r]
 .5        else if h q ~∈ dom dir
 .6                then error
 .7                else (let res = dir(h q) in
 .8                      if ~is-DIR(res)
 .9                         then error
.10                         else dir + [h q→catalog(res,t q,id,r)])
```

This function also makes sure that is is not already used in the directory designated by q . Thus *dom dir* denotes the domain, i.e. the set of resource identifiers used in *dir*. Observe finally that since a qualified name might designate either nonsense or a resource, *res*, other than a directory, i.e. a *DIR*, we test, *is-DIR*, that *res* is indeed a directory.

Further experiments with the above abstraction will be exhibited in section 4.5.

4.1 Defining Domains of *MAP* Objects

Let A, B, A_i and B_j (for $i = 1,2,\ldots,n$ and $j = 1,2,\ldots,l$) be class names. To define the class of objects which are finite domain maps from (subsets of) A, respectively A_i, into B, respectively B_j, we use the domain expression operator $\vec{m}$:

$$A \vec{m} B, \qquad A_i \vec{m} B_j$$

Examples 20:

Let our file system now come of age! Let the file system consist of a set of uniquely named files; and let each file consist of a set of uniquely keyed records, i.e. the file may have its records randomly accessed by key. Let file names, keys and records presently be further unspecified. Then:

$$Key \vec{m} REC$$

is a domain expression suitably abstracting our notion of files, provided *Key* & *REC* denote the domains of keys and records. If you wish to name this file domain, by e.g. *FILE*, then:

$$FILE = Key \vec{m} REC$$

is an example of an abstract syntax rule which lets *FILE* denote what $Key \vec{m} REC$ denotes! Now:

$$Fid \vec{m} FILE$$

or equally well:

$$Fid \vec{m} (Key \vec{m} REC)$$

are domain expressions both suitable abstracting our ideas about the file system and files, again provided *Fid* denotes the domain of file names.

4.1.1 Defining *MAP* Domains -- Continued

Then the logical type expression:

$$(A_1|A_2|\ldots|A_n) \vec{m} (B_1|B_2|\ldots|B_l) \qquad l \gtreqless n$$

denotes the class of objects whose members are finite domain maps from (subset of) the union class of A_1, A_2, ..., and A_n objects into the union class of B_1, B_2, ..., and B_l objects.

Example 21:

Now define keys and records to be either integers or characterstrings, respectively finite, non-zero length sequences of either integers or characterstrings -- presently with no constraints. Then, if *INTG* and $QUOT^+$ are suitable abstractions of integers and characterstrings respectively, we have that:

$$(INTG|QUOT^+) \vec{m} (INTG^+|QUOT^{++})$$

is a domain expression which defines any one file to contain both integers and characterstrings in its domain, i.e. as its keys; and sequences of both of these in its co-domain or range, i.e. as its records. Thus e.g. a given file may map *3* into <*A*,*X*,*H*>, <*P*> into <*2*,*4*,*9*,*11*>, *5* into <*3*,*5*,*7*>, and <*Y*,*Z*> into <*I*,*J*>.

4.1.2 Defining *MAP* Domains -- Continued

The logical type expression:

$$(A_1 \vec{m} B_1) \cup (A_2 \vec{m} B_2) \cup \ldots\ldots \cup (A_k \vec{m} B_k)$$

denotes the class of objects whose members are finite domain maps as above but with the restriction that A_1 objects, if at all mapped, i.e. in the domain, are mapped into B_1 objects, A_2 objects (...) into B_2 objects, ..., and A_k objects into B_k objects. We read the $\cup$ *MAP* domain operator as a 'merge'-, rather than (as we did for |) an 'either'-, union operation.

Example 22:

If we wish to constrain integer keys to map into integer field records, and characterstring keys into records whose fields are characterstrings, then:

$$(INTG \xrightarrow[m]{} INTG^{+}) \ \underline{\cup} \ (QUOT^{+} \xrightarrow[m]{} QUOT^{++})$$

is a domain expression of the right kind for specifying the above!

Defining *MAP* Domains -- Terminated

The logical type expression:

$$(A_1 \xrightarrow[m]{} B_1) \ | \ (A_2 \xrightarrow[m]{} B_2) \ | \ \ldots \ | \ (A_k \xrightarrow[m]{} B_k)$$

denotes a class of objects whose members are finite domain, partial maps from A_1 into B_1, or A_2 into B_2,..., or A_k into B_k. Thus a given member is either an $A_1 \xrightarrow[m]{} B_1$ object, or an $A_2 \xrightarrow[m]{} B_2$ object,..., or an $A_k \xrightarrow[m]{} B_k$ object.

Example 23:

If finally any one file should not mix integers and characterstrings in its key domain or record co-domain, then:

$$(INTG \xrightarrow[m]{} INTG^{+}) \ | \ (INTG \xrightarrow[m]{} QUOT^{++}) \ |$$
$$(QUOT^{+} \xrightarrow[m]{} QUOT^{++}) \ | \ (QUOT^{+} \rightarrow INTG^{+})$$

is a domain expression guaranteeing this! Note that the domain expression:

$$(INTG|QUOT^{+}) \xrightarrow[m]{} (INTG|QUOT^{+})^{+}$$

permits any one record to have some fields being integers, others being characterstrings. If that is your ideas about records, this is a way in which you can so express it!

Note:

A map or a function from subsets of A into B is said to be a partial map, respectively partial function. It is quite common to deal with partial maps.

Programming Note:

Defining a class of functions using the $\vec{m}$ operator shall express that such objects have their graph, i.e. argument-value associations, computed when defined. This is in contrast to using the $\rightarrow$ and $\tilde{\rightarrow}$ operators whose use in defining total and partial functions shall express that the objects are implicitly defined through some λ-expressions-like device, about which it can be said that the graph of the denoted function is not computed when defined -- in fact: is never computed -- and where, in addition, the function domain(s) may be infinite. For the story on such functions see section 5.

4.2 Representing Instances of *MAP* Objects

4.2.1 Explicit Enumeration

Let a_1, a_2, ..., a_d be distinct A objects, a_{i1}, a_{i2}, ..., a_{id} be distinct A_i objects, b_1, b_2, ..., b_d be not necessarily distinct B objects, and b_{j1}, b_{j2}, ..., b_{jd} be not necessarily distinct B_j objects, then the constructor expressions:

$$[a_1 \rightarrow b_1, a_2 \rightarrow b_2, \ldots, a_d \rightarrow b_d]$$
$$[a_{i1} \rightarrow b_{j1}, a_{i2} \rightarrow b_{j2}, \ldots, a_{id} \rightarrow b_{jd}]$$

and:

denote maps from (subsets of) A into B, respectively (subsets of) A_i into B_j, i.e. in: $A \vec{m} B$, respectively $A_i \vec{m} B_j$. The constructor expression:

$$[a_{11}\rightarrow b_{11}, a_{12}\rightarrow b_{12}, \ldots, a_{1m_2}\rightarrow b_{1m_1},$$
$$a_{21}\rightarrow b_{21}, a_{22}\rightarrow b_{22}, \ldots, a_{2m_2}\rightarrow b_{2m_2},$$
$$\ldots$$
$$a_{k1}\rightarrow b_{k1}, a_{k2}\rightarrow b_{k2}, \ldots, a_{km_k}\rightarrow b_{km_k}]$$

denotes a map in:

$$(A_1 \vec{m} B_1) \ \underline{\cup} \ (A_2 \vec{m} B_2) \ \underline{\cup} \ \ldots \ \underline{\cup} \ (A_k \vec{m} B_k)$$

assuming of course that $a_{ij}\epsilon A_i$, $b_{ij}\epsilon B_i$ that is: that first subscripting index, i, binds to domain A_i respectively B_i. And:

$$[a_{x1}\rightarrow b_{y1}, a_{x2}\rightarrow b_{y2}, \ldots, a_{xm}\rightarrow b_{ym}]$$

denotes a map in:

$$(A_1|A_2|....A_n) \ \vec{m} \ (B_1|B_2|...|B_l)$$

provided:

$$a_{xi}\epsilon(A_1|A_2|...|A_n) \text{ and } b_{yj}\epsilon(B_1|B_2|...|B_l).$$

Empty Map:

-- is denoted by:

[]

Example 24:

Let fid_1, fid_2,..., fid_l denote, i.e. be abstractions of, distinct file names; let k_{ij} etc. denote, i.e. be abstractions of keys distinct for fixed i's; and let r_{ij} etc. denote, i.e. be abstractions of not necessarily distinct records. Then:

$$[fid_1 \rightarrow [k_{11} \rightarrow r_{11}, k_{12} \rightarrow r_{12}, \ldots, k_{1n_1} \rightarrow r_{1n_1}],$$
$$fid_2 \rightarrow [k_{21} \rightarrow r_{21}, k_{22} \rightarrow r_{22}, \ldots, k_{2n_2} \rightarrow r_{2n_2}],$$
$$\ldots$$
$$fid_l \rightarrow [k_{l1} \rightarrow r_{l1}\ k_{l2} \rightarrow r_{l2}, \ldots, k_{ln_m} \rightarrow r_{ln_l}]]$$

is a schematized (...) snapshot of an object in $Fid \xrightarrow[m]{} (Key \xrightarrow[m]{} REC)$, provided $fid_i \in Fid$, $k_{ij} \in Key$ and $r_{ij} \in REC$.

If $Key = INTG$ and $REC = QUOT^{++}$ then:

$$[3 \rightarrow \langle\langle \underline{A}, \underline{Z}, \underline{B} \rangle, \langle \underline{Y}, \underline{C} \rangle, \langle \underline{X}, \underline{D}, \underline{W}, \underline{E} \rangle\rangle,$$
$$5 \rightarrow \langle\langle \underline{V} \rangle\rangle,$$
$$7 \rightarrow \langle\langle \underline{F}, \underline{U}, \underline{G}, \underline{T}, \underline{H} \rangle, \langle \underline{S} \rangle\rangle]$$

represents a file of three records, with keys 3,5 and 7; etc. Whereas keys of any one file need be distinct, records or any two keys of respectively distinctly names files need not be distinct:

$$[fid_\alpha \rightarrow [2 \rightarrow \langle 3,4,5 \rangle, 3 \rightarrow \langle 3,4,5 \rangle],$$
$$fid_\beta \rightarrow [3 \rightarrow \langle 3,4,5 \rangle, 2 \rightarrow \langle 3,4,5 \rangle]];$$

in fact, as the above example of a snapshot of a small file system shows, distinctly named files need themselves not be distinct.

4.2.2 Implicit Enumeration

Let F: $D_d \xrightarrow{\sim} A$ and G: $D_r \xrightarrow{\sim} B$ denote partial functions, then:

$$[\ F(d) \rightarrow G(r)\ |\ P(d,r)\ ]$$

is a map constructor expression denoting a map in $A \xrightarrow[m]{} B$. P is assumed to be a predicate: P: $D_d\ D_r \rightarrow BOOL$.

Let F_i, F_{1n}, G_j, $G_{1\ell}$ denote partial functions:

$$\begin{array}{lll} F_i: & D_d \xrightarrow{\sim} A_i, & \\ F_{1n}: & D_d \xrightarrow{\sim} (A_1|A_2|\ldots|A_n), & \\ G_j: & D_r \xrightarrow{\sim} B_j & \text{and:} \\ G_{1\ell}: & D_r \xrightarrow{\sim} (B_1|B_2|\ldots|B_l) & \end{array}$$

for $i=1,2,\ldots,n$ and $j=1,2,\ldots,l$. The expressions:

$$\begin{array}{lll} [F_i(d) & \rightarrow G_j(r) \mid P(d,r)], & \text{and:} \\ [F_{1n} & \rightarrow G_{1\ell}(r) \mid P(d,r)] & \end{array}$$

denote maps from (subsets of) respectively A_i into B_j, and $(A_1|A_2|\ldots|A_n)$ into $(B_1|B_2|\ldots|B_l)$.

The expression:

$$\begin{array}{l} [F_1(d) \rightarrow G_1(r) \mid P_1(d,r)] \\ \cup[F_2(d) \rightarrow G_2(r) \mid P_2(d,r)] \\ \ldots \\ \ldots \\ \cup[F_k(d) \rightarrow G_k(r) \mid P_k(d,r)] \end{array}$$

where P_i are predicates, as above, denotes a map in:

$$(A_1 \xrightarrow[m]{} B_1) \ \underline{\cup} \ (A_2 \xrightarrow[m]{} B_2) \ \underline{\cup} \ \ldots \ \underline{\cup} \ (A_k \xrightarrow[m]{} B_k) .$$

Example 25:

Let F_r, F_f denote total (for simplicity: pure, applicative) functions from records into records, respectively files into files. Let s denote a file system, f some file, r some record; and, foregoing the next subsection (2.3.3) on operations, let $\underline{dom}\, s$ and $\underline{dom}\, f$ denote the set of filenames, respectively the set of keys, of the file system -- respectively of a file. Then:

$$\begin{array}{l} [\ fid \rightarrow F_f(s(fid)) \mid fid \in \underline{dom}\, s] \\ [\ k \rightarrow F_r(f(k)) \mid k \in \underline{dom}\, f] \end{array}$$

and:

$$[\ fid \rightarrow [\ k \rightarrow F_r((s(fid))(k)) \mid k \in \underline{dom}\, s(fid)] \mid fid \in \underline{dom}\, s\ ]$$

denotes a file system, a file, and a file system. The former file system is derived from a given file system, s, by transforming each of its files; the latter by transforming each of its records individually. The file is derived from a given file, f, by transforming each of its records. Thus the above three implicit map constructor expressions may be reasonable abstractions of various file 'processing' programs. Observe how randomness of record and file 'positions' or 'accessing' is preserved by the unorderedness of 'fetching' files (fid) respectively records (k, $f(k)$,....), for transformation.

4.2.3 A Note on Scopes

We have now seen three distinct meta-language constructs for implicit object constructions:

$$\{\ F_h(r)\ \mid\ r \in f\ \}$$
$$<\ F_h(f[i])\ \mid\ 1 \leq i \leq \underline{l}\ f\ >$$
$$[\ k \rightarrow F_h(f(k))\ \mid\ k \in \underline{dom}\ f\ ]$$

In all three, F_h and f were quantities defined outside these expressions; and r, i and k respectively, were bound variables, being defined and bound to the right of the builder combinator: |, and with a scope extending to the closest embracing pair of { & }, < & >, respectively [&]. The scope of these bound variables follow rules analog to those in ordinary programming languages: thus they extend to inner, i.e. nested expressions, including { }, < > and [ ] based constructor expressions -- unless redefined in such inner forms, a practice which certainly can and should be avoided! The last file system building expression of the previous example section illustrates the doubly nested use of bound variables. So does:

Example 26:

Let:

$$A \xrightarrow[m]{} (B \xrightarrow[m]{} C)$$
$$(A\ B) \xrightarrow[m]{} C$$

be domain expressions where we do not presently bother about A, B and C. Then: for suitably decorated, and occasionally distinct, $a{\in}A$, $b{\in}B$ and $c{\in}C$:

$$[\ a1 \rightarrow [b11 \rightarrow c11, b12 \rightarrow c12],\ a2 \rightarrow [b21 \rightarrow c21, b22 \rightarrow c22, b23 \rightarrow c23]\ ]$$

and:

$$[\ (a1,b11) \rightarrow c11, (a1,b12) \rightarrow c12,\ (a2,b21) \rightarrow c21, (a2,b22) \rightarrow c22, (a2,b23) \rightarrow c23]$$

are examples of somehow 'equivalently' denoting expressions in that functions can be defined which converts between their objects.

Let, for ease of reference, the first domain be named X, the second Y:

$$X = A \xrightarrow[m]{} (B \xrightarrow[m]{} C)$$
$$Y = (A\ B) \xrightarrow[m]{} C$$

then:

$$convxy(x) = [(a,b) \rightarrow c \mid a{\in}\underline{dom}\ x \wedge b{\in}\underline{dom}(x(a)) \wedge c = (x(a))(b)\ ]$$

$$convyx(y) = [\ a \rightarrow [b \rightarrow c \mid (a',b){\in}\underline{dom}\ y \wedge a' = a \wedge c = y(a,b)] \mid (a,\ ){\in}\underline{dom}\ y\ ]$$

where:

$$\underline{type}: convxy: X \rightarrow Y$$
$$\underline{type}: convyx: Y \rightarrow X$$

In fact the two functions are each others' inverses.

4.3 *MAP* Operations

The following special *MAP* operations are provided:

U	merge,
+	override,
(...)	apply,
⧵	restrict with,
|	restrict to,
dom	domain,
rng	range,
	composition

Each of these operators will now be individually, and quite informally explained:

*map1*U*map2* denotes a *MAP* provided the domains of *map1* and *map2* are disjoint, otherwise *undefined*. The denoted *map* maps domain elements of *map1* into the same range elements as does *map1*, and domain elements of *map2* into the same range elements as does *map2*, and maps nothing else.

map1+*map2* denotes the *MAP* which maps domain elements of *map2* into the same range elements as does *map2*, and maps those domain elements of *map1* which are not in the domain of *map2* into the same range elements as does *map1*, and maps nothing else.

map(a) denotes the range *OBJ*ect into which *map* maps *a*. If *a* is not in the domain of *map* the operation is *undefined*.

map⧵*wset* denotes the *MAP* which maps those domain elements of *map* which are not in the set *wset* into the same range elements as does *map*, and maps nothing else.

map|*tset* denotes the *MAP* which map those domain elements of *map* which are also in *tset* into the same range elements as does *map*, and maps nothing else.

dom map denotes the set of objects which are domain elements of *map*.

rng map denotes the *SET* of *OBJ*ects which are the range elements of *map*.

map1•*map2* denotes a *MAP* provided the range of *map2* is included, i.e. contained ($\subseteq$) in the domain of *map1*, otherwise the operation is *undefined*. The denoted map maps into those range elements of *map1* which are mapped to by the domain elements of *map1*, mapped into, as range elements of *map2* from domain elements of *map2*. More formally, and in this case certainly more concisely:

(map1•map2)(d) = *if* d∈*dom map2*
then if map2(d) ∈ *dom map1*
then map1(map2(d))
else undefined
else undefined

Example 27:

Let *s*, *id*, *f*, *r* and *k* stand for abstractions of file systems, file names, files, records and keys, then:

(1)	$s \cup [id \rightarrow f]$	$f \cup [k \rightarrow r]$
(2)	$s + [id \rightarrow f]$	$f + [k \rightarrow r]$
(3)	$s \setminus \{id\}$	$f \setminus \{k\}$
(4)	$s(id)$	$f(k)$
(5)	$id \in dom\ s$	$k \in dom\ f$
(6)	$f \in rng\ s$	$r \in rng\ f$

may be reasonable abstractions of the following typical operations on file systems (left column) [and files (right column)]: (1) writing an entirely new file [record] into the file system [a file] -- in the sense of its name [key] not hitherto being one of a file [record] in the system [file]; (2) updating an entire file [a single record] of the system [a file]; (3) deleting an entire file [record] named *id* [keyed with *k*] from the system [a file]; (4) reading an entire file [a record]; (5) asking whether a file of a given name [a record with a given key] is in the system [a file]; and (6) inquiring whether a given file (i.e. its 'value', not name) [record ('value', not key)] is in the system [a file].

Let suitably decorated *id*'s and *k*'s denote file names, respectively keys. Then:

(7) $s \mid \{id_1, id_2, \ldots, id_n\}$ $\qquad$ $f \mid \{k_1, k_2, \ldots, k_m\}$

seems to be an acceptable abstraction for the operation of restricting the files of a system to those of the indicated names -- i.e. deleting all other! [respectively deleting all those file records whose keys are different from k_i $(1 \leq i \leq m)$].

Combining the above operations:

(1') $s + [\ id \rightarrow s(id) \cup [k \rightarrow r]\ ]$

(2') $s + [\ id \rightarrow s(id) + [k \rightarrow r]\ ]$

(3') $s + [\ id \rightarrow (s(id) \setminus \{r\}\ ]$

(4') $(s(id))(k)$

(5') $k \in \underline{dom}(s(id))$

(6') $r \in \underline{rng}(s(id))$

(7') $s + [\ id \rightarrow ((s(id)) \mid \{k_1, k_2, \ldots, k_n\})\ ]$

we get the right column equivalents, on a file system.

4.4 *MAP*-oriented Combinators

The same combinator: the *for-all-do* statement, as was defined for processing sets, is available for processes on maps, e.g.:

for all d ∈ dom map do S(map(d))

Example 28:

We give the imperative variant of the last of the three implicit, applicative map constructions shown in the example 25.

```
del system := [] type (Fid ⇀m (Key ⇀m REC)),
    file   := [] type (Key ⇀m REC);
(for all fid ∈ dom s do
    file := [];
    for all k ∈ dom s(fid) do
        file := (c file) ∪ [ k → F_h ((s(fid))(k)) ];
    system := (c system) ∪ [ fid → (c file) ])
```

4.5 Further Examples

Example 29:

The introductory example of this section is now continued. We are now interested in defining a number of auxiliary functions applicable to the operating system directory. A function for catalogueing new entries, i.e. new resources, has already been shown. The dual function of uncatalogueing e.g. takes a directory and a non-null resource name and deletes the designated resource -- i.e. returns a directory in which the resource name is no longer a valid name:

```
1.0     uncatalog(dir,q)=
 .1        cases q:
 .2           (<id> → dir\{id}
 .3            T    → dir + [ h q → uncatalog(dir(h q),t q) ] )
           type:  DIR Rid+  →  DIR
```

Provided q indeed does designate some resource in dir! When such is the case we can show that:

$$uncatalog(catalog(dir,q,id,r),q^{\frown}\langle id\rangle) = dir$$

A function for testing whether a non-null resource name, q, does indeed designate something meaningfull in a directory, dir, is e.g. the following:

```
2       isvalidrn(dir,q) = q ∈ resnms(dir)
           type:  DIR Rid+ → BOOL
```

where:

```
3.0     resnms(dir)=
 .1        { <rid>^rn | rid ∈ dom dir
 .2                   ∧ (is-DIR(dir(rid)) → rn∈resnms(dir(rid)),
 .3                      T                → rn = <>) }
           type:  DIR → Rid+-set
```

another function would be:

```
4.0     isrnok(dir,q)=
 .1        (h q ∈ dom dir)
 .2        ∧(cases t q:
 .3             (<>  →  true,
 .4              T   →  isrnok(dir(h q),t q)))
           type:  DIR Rid+  →  BOOL
```

where we rely on the non-commutativeness, in the meta-language, of the boolean ∧ and ∨ operators. A function for retrieving a resource given its name could e.g. be expressed:

```
5.0     retrieve(dir,q)=
 .1        cases q:
 .2           (<id> → dir(id),
 .3             T   → retrieve(dir(h q),t q))
           type: DIR Rid+ → RES
```

The domain expression $Rid \xrightarrow[m]{} RES$ also defines the void map, [], from no resource identifiers to no resources, as part of its domain. In some of the above function definitions we have tacitly assumed that no (embedded) directory was empty. A predicate function for testing this well-formedness criteria not expressed by the domain expressions might e.g. look like:

```
6.0     is-wf-DIR_o(dir)=
 .1        (dir≠[])
 .2       ∧(∀rid ∈ dom dir)
 .3           (is-DIR(dir(rid)) ⊃ is-wf-DIR_o(dir(rid)))
           type: DIR → BOOL
```

File identifiers are part of the directory. If we e.g. impose that no two distinct resource names, via designated file identifiers, "point" to identical files, then the above $is\text{-}wf\text{-}DIR_o$ must be augmented:

```
7.0     is-wf-DIR(dir)=
 .1        is-wf-DIR_o(ir)
 .2       ∧(∀rn_1,rn_2 ∈ resnms(dir))
 .3          ((rn_1≠rn_2)
 .4           ⊃(((is-Fid(retrieve(dir,rn_1))∧is-Fid(rétrieve(dir,rn_2)
 .5             ⊃(retrieve(dir,rn_1) ≠ retrieve(dir,rn_2))))
```

Another Example, 30:

In this example we shall motivate the modeling of the scope-binding and variable concepts of block- & procedure-oriented programming languages by means of so-called environments, respectively abstract stores. In subsequent examples, (31,40,42,61), we shall then illustrate the semantics modeling of procedures and blocks of such languages. The introduction, into our realm of abstract modeling concepts, of environments and stores will now be argued in a few, short, illustrative, but not really complete steps. The notation used in the following for some arbitrary, realistic source languages notions of begin-end blocks and variable declarations and use has been deliberately chosen to coincide with the meta-languages notation for the same constructs. These will, however, not be formally introduced till section 10.

The form:

$$(\underline{dcl}\ v := expr\ \underline{type}\ D;\ C(\ldots))$$

is taken to represent some source languages block construct (with "(" equalling *begin*, and ")" the *end*). In the semantics of this source language v, which itself is an identifier, is to denote something constant. In fact: in this source-, as well as in our meta-, language, the denotation of identifiers is constant over their scope, i.e. does not change! The constant denotation of v over the above block, i.e. its defining scope, is, rather concretely speaking, taken to be the location (or: address) of the storage cell in which are to be held values, $\underline{cv}$, of the variable v. Thus, if by LOC, we denote the abstract domain of storage locations, and by OBJ, the values which can be held in those locations, then the domain of storages can be abstracted as:

$$LOC \xrightarrow[m]{} OBJ$$

We choose $\xrightarrow[m]{}$ since there will, at any point of execution of our source programs, only be a finite number of active variables, i.e. of allocated storage cells. Thus we abstract storages as unique associations, maps from locations to their values.

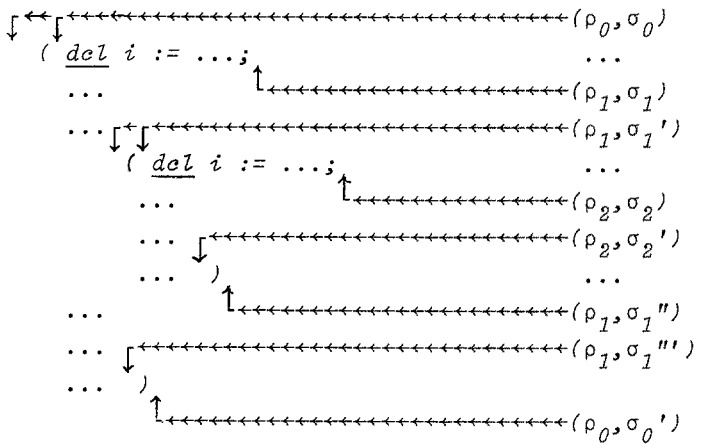
To keep track of the location that variable identifiers denote we introduce, speaking concrete again, a table, henceforth called an environment, in which is recorded the associations of variable names to their locations.

$$Id \xrightarrow[m]{} LOC$$

where e.g. $v \in Id$. Again the association is functional, and again the functional recordings are finite. Before argueing why we do not directly model storages as:

$$Id \xrightarrow[m]{} OBJ$$

let us first see that we do indeed need the full use of the meta-language *MAP* concept, i.e. of the *MAP* operations. Simultaneously we exemplify 'snapshots' of environments, ρ, and stores, σ. The example is that of one simple block, containing the declaration of the variable named i, nested in an outer block, also containing the declaration of a identically named variable.



The environment immediately prior to outer block elaboration is ρ_0 and the state is σ_0. These are not detailed. Obeying the 'first' declaration results in the elaborator claiming a new, fresh storage cell for i, let us call its location li_1. Thus:

$$\sigma_1 = \sigma_0 \cup [li_1 \rightarrow o_1]$$

with:

$$\rho_1 = \rho_0 + [i \rightarrow li_1]$$

Thus we use U as an abstraction of the storage allocation operation -- U since li_1 is new, i.e. not in the *dom*ain of σ_0: $li_1 \sim\in dom\ \sigma_0$! But we use + as an abstraction of the environment 'update' action -- i may namely be already recorded in ρ_0. Observe that we do "fall back" to ρ_0 when "falling through" this, the outer, block. Similarly:

$$\sigma_1' = \sigma_0' \cup [li_1 \rightarrow o_1']$$

where the ' on σ_0 shall indicate that variables outside this block may have had their values updated. Also i may have been updated (o_1')! And:

$$\rho_2 = \rho_1 + [i \rightarrow li_2] = \rho_0 + [i \rightarrow li_2]$$
$$\sigma_2 = \sigma_1' \cup [li_2 \rightarrow o_2] = \sigma_0' \cup [li_1 \rightarrow o_1', li_2 \rightarrow o_2]$$

That is: the storage claimed for the inner blocks' i is also fresh, not clashing with the storage for the outer blocks' i.

Observe how environments, e.g. ρ_1, remain constant in the outer block.

$$\sigma_2' = \sigma_0'' \cup [li_1 \rightarrow o_1'', li_2 \rightarrow o_2']$$
$$\sigma_1'' = \sigma_2' \setminus \{li_2\} = \sigma_0'' \cup [li_1 \rightarrow o_1'']$$
$$\sigma_1''' = \sigma_0''' \cup [li_1 \rightarrow o_1''']$$
$$\sigma_0' = \sigma_1''' \setminus \{li_1\} = \sigma_0'''$$

Thus freeing, or deallocation of storage for declared variables follow the block structure, i.e.: is performed as part of the block epilogue. The *MAP* operator abstracting deallocation is: $\setminus$.

We now turn to a brief justification for the use of <u>both</u> environments and stores. The above example of nested blocks redefining variable names is not a sufficiently convincing example. Static renaming of e.g. the inner i to e.g. j would permit modeling stores as $Id \xrightarrow[m]{} OBJ$. If, however, any block was [part of] a recursively defined procedure static renaming would not solve the problem of nested activations' i denoting distinct locations -- which are the semantics we normally expect!

In example 31 and onwards we use *ENV* and sometimes Σ as follows:

$$ENV = Id \xrightarrow[m]{} LOC$$
$$\Sigma = LOC \xrightarrow[m]{} OBJ$$

Finally, as a last example, we show the modeling of variable referencing, value access and of assignment.

For the elaborator 'state' (ρ,σ) elaboration of some $v \in \underline{dom}\ \rho$ is modeled $(\blacktriangle)$ as:

$$v \quad \blacktriangle \quad \rho(v)$$

Accessing the contents of a variable v:

$$\underline{c}\ v \quad \blacktriangle \quad \sigma(\rho(v))$$

and assigning the object (value) o to variable v changes the state σ to:

$$v := o \quad \blacktriangle \quad \sigma + [\rho(v) \rightarrow o].$$

A Last Example, 31:

The variables of some source language are either of scalar or array type with all array objects being scalars. Variable references, however, are to scalars only, i.e. not to arrays or array-slices thereof.

The syntactic domains of variable declarations of a block can be modeled as a map from variable name identifiers to their type:

1 $Block :: (Id \xrightarrow[m]{} Type) \ldots Stmt^*$

with:

2 $Type :: Scalar\text{-}type\ [Expr^+]$

i.e. with types having an optional [...] array upper bounds expression list, $Expr^+$. If the optional part is absent, i.e. $\underline{nil}$, the variable is a scalar.

The corresponding semantic domain of storages can be modeled as a map from scalar locations to scalar values.

3 $STG = (Bool\text{-}loc \xrightarrow[m]{} Bool) \cup (Int\text{-}loc \xrightarrow[m]{} Intg)$

where:

4 $Bool = BOOL \mid \underline{undefined}$
5 $Intg = INTG \mid \underline{undefined}$

Environments keep track of variable name associations:

6 $ENV = Id \xrightarrow[m]{} LOC$

with :

7 $LOC = Scalar\text{-}Loc \mid Array\text{-}Loc$
8 $Array\text{-}Loc = (N_1^+ \xleftrightarrow[m]{} Int\text{-}Loc) \cup (N_1^+ \xleftrightarrow[m]{} Bool\text{-}Loc)$
9 $Scalar\text{-}Loc = Int\text{-}Loc \mid Bool\text{-}Loc$

with the constraint that the integer index lists of any given array location forms a 'rectangle':

10.0 $(\forall l \in Array\text{-}Loc)$
.1 $(\exists il \in N_1^+)(\underline{dom}\ l = rectangle(il))$

where we informally explain rectangle:

11.0 $rectangle(\langle ub_1, ub_2, \ldots, ub_n \rangle) =$
.1 $\{ \langle i_1, i_2, \ldots, i_n \rangle \mid i_k \in \{1:ub_k\} \wedge 1 \leq k \leq n \}.$

The il 'picked' is (to be) the tuple, whose length corresponds to the array-dimension, and whose k'th element (ub_k) is the *u*pper-*b*ound index for the k'th dimension, all of whose lower-bounds are *1*!

A particular storage is modeled (here) as a global meta-variable, STG:

12 $\underline{dcl}$ STG := [] $\underline{type}$ STG

In the block prologue (.1-.2) locations of block declared variables are (automatically) allocated, and these are likewise freed (.4-.8) in the block epilogue:

```
13   int-Block(mk-Block(dcls,...,stl))(ρ)=
.1        (let ρ' : ρ + [id → get-loc(dcls(id))(ρ)
.2                       | id ∈ dom dcls ];
.3         int-Stmt-list(stl)(ρ');
.4         let locs  = { ρ'(id) | id ∈ dom dcls } in
.5         let slocs = { l | ((l∈locs) ∧ is-Scalar-Loc(l))
.6                          ∨((l ∈ rng al) ∧ (al ∈ locs) ∧
.7                                          is-Array-Loc(al))} in
.8         STG := c STG \ slocs)
```

with:

```
14.0     get-loc(mk-Type(sctp,bdl)) (ρ)=
  .1        if bdl=nil
  .2           then
  .3              (let l ∈ Scalar-Loc be s.t. (l~∈ dom cSTG)∧
  .4                                          l-tp-match(sctp,l);
  .5               STG := c STG ∪ [l→undefined];
  .6               return(l))
  .7           else
  .8              (let ebdl : < eval-Expr(bdl[i])(ρ) | 1 ≤ i ≤ len bdl>;
  .9               if (∃i ∈ {1:len ebdl})(ebdl[i]<1)
 .10                  then error
 .11                  else
 .12                     (let l ∈ Array-Loc be s.t.
 .13                              ((scl ∈ rng l)⊃ l-tp-match(sctp,scl))
 .14                              ∧(dom l = rectangle(ebdl))
 .15                             ∧((rng l ∩ dom c STG) = {});
 .16                      STG := c STG ∪ [ scl→undefined | scl ∈ rng l];
 .17                      return(l)))
```

and

```
15.0     l-tp-match(tp,l) = ((tp=BOOL) → is-Bool-Loc(l),
  .1                         (tp=INT)  → is-Int-Loc(l))
```

Assuming that the state, Σ, is describable as:

```
16       Σ   =   STG →m STG
```

(i.e. Σ is a one-point (STG) map), the type of the above functions are:

type: *int-Block*: $Block \xrightarrow{\sim} (ENV \xrightarrow{\sim} (\Sigma \xrightarrow{\sim} \Sigma))$
get-Loc: $Type \xrightarrow{\sim} (ENV \xrightarrow{\sim} (\Sigma \xrightarrow{\sim} (\Sigma\ LOC)))$
rectangle: $N_1^+ \rightarrow (N_1^+)\text{-}set$

5. *TREE*s

Examples 32-33:

*Block*s of some source language consists of three distinct parts: an unordered set of *var*iable declarations, an unordered set of distinctly named *proc*edures and an ordered sequence of one or more *stat*ements.

$$(Var\text{-}set \quad Id \xrightarrow[m]{} Proc \quad Stmt^+)$$

The above is a domain expressions. If *Var* denotes the domain of variable identifiers, *Id* that of procedure identifiers, *Proc* the domain of procedures and *Stmt* the domain of statements; then the above domain expression abstracts a nameless domain of blocks. These are trees, whose first subcomponents are sets of variable names, second subcomponents are maps from identifiers to procedures, and whose last subcomponents are tuples of statements.

If suitably decorated *v*'s, *id*'s, *p*'s and *s*'s stand for *v*ariables, procedure names, *p*rocedures, and *s*tatements, then:

$$mk(\{v_1,v_2,\ldots,v_v\},[id_1\rightarrow p_1,id_2\rightarrow p_2,\ldots,id_p\rightarrow p_p],\langle s_1,s_2,\ldots,s_s\rangle)$$

and:

$$(\{v_1,v_2,\ldots,v_v\},[id_1\rightarrow p_1,id_2\rightarrow p_2,\ldots,id_p\rightarrow p_p],\langle s_1,s_2,\ldots,s_s\rangle)$$

(the latter constructor expression having dropped the nameless constructor function, *mk*) represent the way we write down such composite objects. If *b* is a block, or more precisely, if *b* is an object in the domain specified above, then the following meta-language combinators:

let *mk(vs,pm,sl)* = *b*;
let *(vs,pm,sl)* = *b*;

provide means of decomposing nameless tree structured objects, here modelling blocks, into their proper constituents -- here the set of variables, *vs*; the map from procedure names to procedures, *pm*; and the statement list, *sl*.

The domain expression:

$$(s\text{-}vars{:}Var\text{-}set \quad s\text{-}procm{:}(Id \xrightarrow[m]{} Proc) \quad s\text{-}stmtl{:}Stmt^{+})$$

(with inner (,)'s, around $Id \xrightarrow[m]{} Proc$, used only as syntactic delimiter) defines the same domain as above, but in addition defines three selector functions, here arbitrarily named: *s-vars*, *s-procm*, *s-stmtl*. Now:

$$\begin{aligned} &\underline{let}\ vs = s\text{-}vars(b), \\ &\quad pm = s\text{-}procm(b), \\ &\quad sl = s\text{-}stmtl(b); \end{aligned}$$

has the same effect as the previous decomposition combinators. In other words, if $vs \in Var\text{-}set$, $pm \in Id \xrightarrow[m]{} Proc$, $sl \in Stmt^{+}$, then:

$$\begin{aligned} s\text{-}vars(mk(vs,pm,sl)) &= vs, \\ s\text{-}procm(mk(vs,pm,sl)) &= pm, \\ s\text{-}stmtl(mk(vs,pm,sl)) &= sl. \end{aligned}$$

...

We can give a name to the above block domain, e.g. *Block*:

$$Block :: Var\text{-}set \quad Id \xrightarrow[m]{} Proc \quad Stmt^{+}$$

-- note the dropping of (,)'s! With *vs*, *pm* and *sl* as before:

$$mk\text{-}Block(vs,pm,sl)$$

would now have to be our way of writing down instances of blocks. The trees denoted by *Block* are now named (*Block*). Also here we could introduce selectors:

Block :: s-vars:Var-set s-procm:(Id $\vec{m}$ Proc) s-stmtl:Stmt$^+$

(where the *(,)*'s, around *Id* $\vec{m}$ *Proc*, again are used as syntactic delimiters). And again:

s-vars(mk-Block(vs,...,...)) = vs

etc.. A benefit derived from naming the class of constructed tree objects, here *Block*, is that any object can be tested for membership of the named domain:

is-Block(b)

is *true* provided *b* is the result of some *mk-Block(vs,pm,sl)* -- for any applicable *vs*, *pm* and *sl*.

5.1 Defining Domains of *TREE* Objects

Two means of defining domains of tree objects exist in the meta-language:

(1) Named: Using the abstract syntax rule definition symbol *::* to separate definiens from definiendum:

$D :: D_1\ D_2\ \ldots\ D_n$

(2) Anonymous: Using the domain expressions operator *(...)*,

$(D_1\ D_2\ \ldots\ D_n)$

The former defines named or root labelled trees:

(1) $\{\ mk\text{-}D(d_1,d_2,\ldots,d_n)\ |\ is\text{-}D_i(d_i)$ for all $i\ \}$

The latter defines unnamed trees:

(2) $\{\ mk(d_1,d_2,\ldots,d_n)\ |\ is\text{-}D_i(d_i)$ for all $i\ \}$

or, dropping the *mk:*

$$\{ (d_1,d_2,\ldots,d_n) \mid is\text{-}D_i(d_i) \text{ for all } i \}.$$

Example 34:

Statements of a source language are either assignment statements, while-do statements or if-then statements:

$$\begin{aligned} Stmt &= Asgn \mid While \mid IfThen \\ Asgn &:: Id\ Expr \\ While &:: Expr\ Stmt \\ IfThen &:: Expr\ Stmt \end{aligned}$$

Observe how the two last domains:

$$\{ mk\text{-}While(e,s) \mid is\text{-}Expr(e) \wedge is\text{-}Stmt(s) \}$$

$$\{ mk\text{-}IfThen(e,s) \mid is\text{-}Expr(e) \wedge is\text{-}Stmt(s) \}$$

are disjoint -- simply because the names of their object make functions, *mk-While* and *mk-IfThen*, are distinct:

5.1.1 Tree Constructor Axioms

For any two D' and D'' identifiers being definienses of tree constructing abstract syntax rules:

$$\begin{aligned} D' &:: A_1\ A_2\ \ldots\ A_m \\ D'' &:: B_1\ B_2\ \ldots\ B_n \end{aligned}$$

objects:

$$mk\text{-}D'(a_1,a_2,\ldots,a_m),$$
$$mk\text{-}D''(b_1,b_2,\ldots,b_n)$$

are identical if and only if (iff):

D' is the same identifier as D'';
$m = n$;
$a_i = b_i$ for all i,
i.e. if A_i is the same identifier as B_i

that is, iff the two rules above are the same.

...

For any two implicit tree domain denoting expressions:

$(A_1\ A_2\ \ldots\ A_m)$
$(B_1\ B_2\ \ldots\ B_n)$

objects:

$mk(a_1,a_2,\ldots,a_m)$
$mk(b_1,b_2,\ldots,b_n)$

or, which is the same:

$(a_1,a_2,\ldots,a_m)$
$(b_1,b_2,\ldots,b_n)$

are identical iff:

$m = n$, $a_i = b_i$ and
A_i is the same identifier as B_i for all i.

Example 35-36:

The abstract syntax rules:

$CTLG = Fid \xrightarrow[m]{} (Ktp\ Dtp)$
$Define :: Fid\ (Ktp\ Dtp)$

are intended to define the semantic domain of file description *CaTaLoGues*, respectively the syntactic domain of file definition

commands. To each file, identified by its name (in *Fid*) is associated, in the catalogue, its description -- namely a 'pair' consisting of a *Key type*-, and a *Data type*- indication, i.e. an object in *(Ktp Dtp)*. A file definition command names, say *fid*, the file to be defined *(fid ∈ Fid)*, and gives the *Key type*-, and *Data type* indication, i.e. an object likewise in *(Ktp Dtp)*. In particular a catalog, *ctlg*, may look like:

$$[\ fid_1 \rightarrow mk(ktp_1, dtp_1),\ \ldots\ fid_n \rightarrow mk(ktp_n, dtp_n)\ ],$$

with a define command looking like:

$$mk\text{-}Define(fid, mk(ktp, dtp)).$$

Dropping the somewhat superfluous mentioning of the otherwise nameless *mk* (prefixing *(...)*), we get that *ctlg*, upon execution of the *define* command, is augmented to:

$$ctlg \cup [fid \rightarrow (ktp, dtp)]$$

provided, of course $fid \sim \in \underline{dom}\ ctlg$!

...

As another example of nameless tree constructions let us abstract the syntactic domain of procedures of some source language :

$$Proc :: (Id\ Tp)^* Block$$

The form $(Id\ Tp)^*$ could be written $Parm^*$, i.e.:

$$Proc :: Parm^* Block$$

provided:

$$Parm :: Id\ Tp$$

In the former case the parameter list is abstracted as a possibly

zero-length tuple of parameters, these being abstracted as nameless trees, themselves being 'pairs' of formal parameter *Id*entifiers and their *Type*s (not being further specified). In the latter case these parameters are abstracted as named (*Parm*) trees. Given that suitably decorated *id*'s, *tp*'s and *b*'s denote objects in *Id*, *Tp* and *Block*, respectively, we get, in the two cases, that:

$$mk\text{-}Proc(\langle(id_1,tp_1),(id_2,tp_2),\ldots,(id_n,tp_n)\rangle,b)$$

respectively:

$$mk\text{-}Proc(\langle mk\text{-}Parm(id_1,tp_1),\ldots,mk\text{-}Parm(id_n,tp_n)\rangle,b)$$

display instances of procedures according to either abstraction. Thus observe that the *(,)*'s in $(Id\ Tp)^*$ serve the double function of constructing a domain of anonymous trees, as well as indicating the scope of the tuple domain building * operator.

5.2 Representing Instances of TREE Objects

Given objects o_1, o_2, ..., o_n of not necessarily distinct domains, e.g.: D_1, D_2, ..., D_n, the constructor expressions:

$$(o_1,o_2,\ldots,o_n),\quad mk(o_1,o_2,\ldots o_n)$$

denote (identical) tree objects of domain:

$$(D_1\ D_2\ \ldots\ D_n)$$

i.e. an anonymous tree. Given that there exists an abstract syntax rule of the form:

$$D\ ::\ D_1\ D_2\ \ldots\ D_n$$

the constructor expression:

$$mk\text{-}D(o_1,o_2,\ldots,o_n)$$

denotes a tree object of domain *D*, i.e. a root labelled tree.

5.3 *TREE* Operations: Selector Functions

The only special operation defined on *TREE*s is the selector function. Names of selector functions are either explicitly, or implicitly, definable.

Explicitly Defined Selector Function Names

In the named tree constructing abstract syntax rule:

$$D \quad :: \quad s\text{-}nm_1{:}D_1 \; s\text{-}nm_2{:}D_2 \; \ldots \; s\text{-}nm_l{:}D_l$$

as well as in the anonymous tree constructing domain expression:

$$(s\text{-}nm_1{:}D_1 \; s\text{-}nm_2{:}D_2 \; \ldots \; s\text{-}nm_l{:}D_l)$$

the identifiers:

$$s\text{-}nm_1, \; s\text{-}nm_2, \; \ldots, \; s\text{-}nm_l$$

denote selector functions obeying:

$$\begin{aligned}
&s\text{-}nm_1(mk\text{-}D(o_1,o_2,\ldots,o_l)) = o_1\\
&s\text{-}nm_2(mk\text{-}D(o_1,o_2,\ldots,o_l)) = o_2\\
&\ldots\\
&s\text{-}nm_l(mk\text{-}D(o_1,o_2,\ldots,o_l)) = o_l
\end{aligned}$$

respectively

$$\begin{aligned}
&s\text{-}nm_1((o_1,o_2,\ldots,o_l)) = o_1\\
&s\text{-}nm_2((o_1,o_2,\ldots,o_l)) = o_2\\
&\ldots\\
&s\text{-}nm_l((o_1,o_2,\ldots,o_l)) = o_l
\end{aligned}$$

for all $o_1, o_2, \ldots, o_l$.

In the above, explicit, selector function name defining forms, it is assumed that all identifiers $s\text{-}nm_i$ and $s\text{-}nm_j$ are distinct. You may wish to omit any subset of these, including all, as we have done in the past.

Implicitly Defined Selector Function Names

Assuming uniqueness of D_i , for some or all i in $\{1:l\}$ in:

$$D :: D_1\ D_2\ \ldots\ D_l$$

respectively:

$$(D_1\ D_2\ \ldots\ D_l),$$

and in particular, assuming that D_i is an identifier, the above two tree domain constructing forms define the selector function:

$$s\text{-}D_i\ .$$

Example 37:

In the source language *Statement* syntax rules:

```
Stmt   =   Asgn | While
Asgn   ::  Id Expr
While  ::  Expr Stmt
```

s-Id applies to *Asgn* objects and yields the *Id* component; *s-Expr* applies to *Asgn* and *While* objects and yield the *Expr* component; and *s-Stmt* applies to *While* objects and yield the proper *Stmt* component. We emphasize: 'proper component' since any *While* object, by the first abstract syntax rule, is itself a *Stmt* object.

Non-Unique Selector Function Name Convention

For the common case where two or more identifiers, D_i and D_j, of either:

$$D :: D_1\ D_2\ \ldots\ D_l$$

or:

$$(D_1\ D_2\ \ldots\ D_l)$$

are the same, the following convention is adopted:

In the form $D_1\ D_2\ \ldots\ D_l$ let there be k distinct occurrences of the same identifier, say C, then:

s-C-1, s-C-2, ..., s-C-k

are the selector functions which select the *1*st, the *2*nd,..., respectively the *k*th proper *C* component -- from left-to-right.

Example 38:

In the source language *Statement* syntax rules:

Stmt = ... | For | If
*For :: Id Expr Expr Expr Stmt**
If :: Expr Stmt Stmt

the selector function *s-Expr-1*, *s-Expr-2* and *s-Expr-3* select the same *For* object components as would *s-init*, *s-step* and *s-limit* in:

For :: Id s-init:Expr s-step:Expr s-limit:Expr Stmt;*

and *s-Stmt-1* and *s-Stmt-2* selects the same *If* object components as would *s-then* and *s-else* in:

If :: Expr s-then:Stmt s-else:Stmt.

Programming Notes

The meta-language leaves unexplained the names of the implicitly defined selector functions in such forms as:

*Block :: Id-set (Id$\underset{m}{\rightarrow}$Prc) Stmt**
and: *For :: s-cv:Id (s-i:Expr s-b:Expr s-t:Expr)+ Stmt**

In the latter form there are only two implicitly defined selector functions, namely the two selecting the *(Expr Expr Expr)+* and *Stmt** tuple objects of *For* objects.

The reason for introducing explicit selector function names is mostly pragmatic. That is: there is, in most cases, no technical need for in-

venting names. To see this, recall that the technical purpose of selector functions is to select proper (sub-)components of trees. But this could as well be done by a 'reverse' use of the *mk* function! Let e.g. t be an object of

$$(D_1\ D_2\ \ldots\ D_n),$$

then: the *let* clause:

$$\underline{let}\ (o_1,o_2,\ldots,o_n) = t$$

so-to-speak decomposes t into its n subcomponents. So would:

$$\begin{aligned}\underline{let}\ o_1 &= s\text{-}D_1(t),\\ o_2 &= s\text{-}D_2(t),\\ &\ldots\\ o_n &= s\text{-}D_n(t)\end{aligned}$$

provided, of course, all D_i were distinct (and) identifiers! Similar for D objects t, where:

$$D\ ::\ D_1\ D_2\ \ldots\ D_n$$

leading to:

$$\underline{let}\ mk\text{-}D(o_1,o_2,\ldots,o_n) = t$$

etc..

6 *FUNCTIONs*

By a function, we shall -- in a somewhat circular fashion -- understand an operation which, when applied to something, which we shall call its argument yields a certain thing as the value of the function for that argument.

The object to which the function is applicable, i.e. for which it is guaranteed to yield a value (defined result), constitutes the domain of the function. The yielded values constitute the range (or: co-domain) of the function.

Two functions are identical iff they (1) have the same domain, (2) the same range, and (3) for each argument in the domain the same value. *FunCTion* equality is, however, not a defined operation.

To denote the value of a *f*unction for a given *a*rgument we write a name of the function, say *f*, followed by a name of the argument, say *a*; the latter, the former or both possibly enclosed between parentheses:

$$fa,\ f(a),\ (f)(a),\ (fa),\ (f)a$$

Many functions can more easily be described algorithmically, i.e. by a recipe for how to compute the result value given an argument value, than by explicit or implicit enumeration. Moreover, some, if not most such, functions, which we wish to manipulate (create, pass and apply), have an infinite domain -- and by the pragmatics of *MAP*s could not be constructed as such. Finally: many functions can best be described in an implicit, or even recursive, way, which certainly does not conjure the image, or thought, of its graph being computed at the time of definitions. (By the graph of a function we understand the set of all argument result 'pairs'.)

For maps this graph is indeed being computed at "time" of definition, whereas we may think of this graph never being computed in connection with the kind of function definitions we are interested in in this section.

Examples 39-40:

The following is a block-expression of the meta-language; its value is that of the denotation of *f*, which is the factorial function:

```
(let f(n) = if n=0 then 1 else n×f(n-1) in
 f)
```

...

Let, as another example, the procedure header -- exclusive of procedure name -- and procedure body, of some source language be abstract syntactically describable as:

```
Prc  ::  Id* Block
```

Here *Prc* is the name of the domain of procedure definienses (header + body, - name), *Id* of the domain of formal parameter names, and *Block* of blocks -- as e.g. statement blocks.

The meaning of a procedure, i.e. the denotation of a procedure name, may, according to the school of mathematical semantics, be taken as a function from argument lists to the denotation of blocks. If the denotation of these latter are functions from states (Σ) to states then:

```
Prc: ARG* ~→ (Σ ~→ Σ)
```

Given a *Prc* object, *prc*, i.e. one of the form *mk-Prc(idl,bl)*, we now give the function which itself yields the denotation of *prc*. Let, as a last preparation, procedure denotations be functions of the defining rather than calling environment.

```
V-Prc(mk-Prc(idl,bl))(ρ)(σ)=
   (let fct(al)(ξ)=
          (let ρ' = [ idl[i]→al[i] | 1≤i≤l al ] in
           I-Blk(bl)(ρ+ρ')(ξ))                    in
    fct)
```

Observe the following: eValuation of a *Prc*edure denotation takes place in a defining environment, ρ, and state, σ. The state is ignored, and need thus not have been shown. The result of *Prc*edure

eValuation is a function, *fct*. This function is completely described in the three-line *let* clause. The definition given there can be read as follows: *fct* is that function which when applied to some argument list, *al*, and in some state, ξ, will yield a new state. This new state results from Interpreting the *Block bl* in the defining environment ('slightly') extended -- by the bindings of formal parameter identifiers, *idl*[*i*], to actual, call time, arguments, *al*[*i*] -- and in the calling state, ξ. We do not here display the *I-Block* elaboration function, but see section 6.3.

6.1 Defining Domains of FunCTion Objects

Let *A* and *B* denote Domains. To define the class of implicitly λ-defined objects which are partial, respectively total functions from *A* into *B*, we use the domain expression operators $\xrightarrow{\sim}$, respectively →:

$$A \xrightarrow{\sim} B, \qquad A \to B \quad .$$

Example 41:

Applying the ability to describe function spaces to the built-in operations of the meta-language itself, we can now list the logical type of these:

Sets:	*type:*	∪:	*SET*	*SET*	→	*SET*
		∩:	*SET*	*SET*	→	*SET*
		∖:	*SET*	*SET*	→	*SET*
		⊆:	*SET*	*SET*	→	*BOOL*
		⊂:	*SET*	*SET*	→	*BOOL*
		power:		*SET*	→	*SET*
		union:		*SET*	$\xrightarrow{\sim}$	*SET*
		∈	*OBJ*	*SET*	→	*BOOL*
		card		*SET*	→	N_0

Tuples:	*type:*	$\frown$:	*TUPLE*	*TUPLE*	$\rightarrow$ *TUPLE*
		h, *hd*:		*TUPLE*	$\xrightarrow{\sim}$ *OBJ*
		t, *tl*:		*TUPLE*	$\xrightarrow{\sim}$ *TUPLE*
		l, *len*:		*TUPLE*	$\rightarrow N_0$
		[.]:	*TUPLE*	N_1	$\xrightarrow{\sim}$ *OBJ*
		elems:		*TUPLE*	$\rightarrow$ *SET*
		ind:		*TUPLE*	$\rightarrow N_1$-*set*
		conc:		*TUPLE*	$\xrightarrow{\sim}$ *TUPLE*
		+:	*TUPLE*	*MAP*	$\xrightarrow{\sim}$ *TUPLE*
		$\leftarrow$:	*TUPLE*	N_1-*set*	$\xrightarrow{\sim}$ *TUPLE*
Maps:	*type:*	$\cup$:	*MAP*	*MAP*	$\xrightarrow{\sim}$ *MAP*
		+:	*MAP*	*MAP*	$\rightarrow$ *MAP*
		$\leftarrow$:	*MAP*	*SET*	$\rightarrow$ *MAP*
		|:	*MAP*	*SET*	$\rightarrow$ *MAP*
		(.):	*MAP*	*OBJ*	$\xrightarrow{\sim}$ *OBJ*
		dom:		*MAP*	$\rightarrow$ *SET*
		rng:		*MAP*	$\rightarrow$ *SET*
		•:	*MAP*	*MAP*	$\xrightarrow{\sim}$ *MAP*

The reason why certain of these operations are partial functions will now be explained. The *union* operation applies to sets of sets, *SET* just expresses: sets of objects. We could get around this by writing *SET-set* whereby: *type*: *union*: *SET-set* $\rightarrow$ *SET*, i.e. *union* is total over *SET-set*. *hd* and *tl* applies to non-zero length tuples, i.e.: *type*: *hd*: $OBJ^+ \rightarrow OBJ$, *tl*: $OBJ^+ \rightarrow OBJ^*$ -- note totality. Indexing, [*i*], must have *i* lie in the *ind*ex set of the tuple, otherwise *undefined*. *conc* is to tuples what *union* is to sets, i.e. *type*: *conc*: $TUPLE^* \rightarrow TUPLE$. Updating (+) tuple elements must have *MAP* in $(N_1 \xrightarrow[m]{} OBJ)$ with domain elements be in *ind*ex set of tuple. So not even: *type*: +: $TUPLE\ (N_1 \xrightarrow[m]{} OBJ) \xrightarrow{\sim} TUPLE$ would make + total over defined domain.

6.2 Representing Instances of FunCTion Objects

This section is somewhat differently organized than were otherwise comparable earlier sections (i.2 for i=2,3,4,5). In a number of subsec tions we recount basic aspects of so-called λ-expressions.

We shall shortly, in a sequence of eight small steps, arrive at an understanding of functional abstraction. The idea being to write forms which denote functions. One such class of forms were the map expressions covered in section 4. Another such form is that of λ-expressions. Map expressions are used to define functions whose argument-value association is known (i.e. being fixed) at the instance of definition, and whose domain-range sets are individually known and finite.

Functional Abstraction

(0) There are simple and composite proper names.

(1) Simple names are either arbitrarily assigned to denote something, or their denotation has been assigned an arbitrary name. Composite names express through their structure some analysis of the way in which they denote.

(2) A constant is a proper name having a fixed, or single denotation.

(3) A variable is a proper name whose denotation may range over a set of values.

(4) An expression is a name containing other names as proper constituents.

Composite names are expressions.

(5) A form is either a variable or an expression in which one or more proper names have been replaced by variables.

(6) In order to speak about the function of a free variable that a form is we abstract the form by prefixing it with a sequence of three symbols: a λ, the free *variable* and a *dot*.

$\lambda y.3+y$ is the function of y that $3+y$ is; i.e. which increments any number by *three*.

(7) The passage from a form to an associated function is called functional abstraction.

Functionally abstracting the forms: $3+y$, $x+y$ and $\lambda y.x$ into e.g. $\lambda y.3+y$, $\lambda x.\lambda y.x+y$ respectively $\lambda x.\lambda y.x$ results in the increment-by-3 function, the addition (of two numbers) function and the function from (say) integers (x) to constant functions from (say) booleans (y) to that integer x.

In the above we have employed a rather restricted and not completely detailed use of the concepts: names, expressions and forms. The term expression shall imply the inclusion of simple constants and variable (identifier)s. Forms are henceforth expressions.

λ-expressions

Any clause (expression or statement), *clause*, of the meta-language can be functionally abstracted, whether containing free variables or not (for free/bound variables, see below).

Functional abstraction in zero variables is written:

$\lambda().clause$

Writing, in the meta-language, the definition:

let $f = \lambda().clause$

or:

let $f() = clause$

-- which is equivalent -- thus identifies f as a name for the function of no variables that *clause* is. Any subsequent occurrence in the scope of the definition, i.e. *use* in contrast to *definition*, of f without $()$ shall then denote the function, whereas any subsequent occurrence in the scope of the definition of f of $f()$ shall denote the elaboration of *clause*.

If the valuation of *clause* is dependent on a state -- not shown -- then repeated occurrences of $f()$, elaborated in different states, may result in distinct $f()$, i.e. *clause*, 'values'. These remarks

also apply to functions of several variables

Functional abstraction in two or more variables is written either:

$$\lambda id_1 \cdot \lambda id_2 \cdot \ldots \cdot \lambda id_n \cdot clause$$

or :

$$\lambda(id_1, id_2, \ldots, id_n) \cdot clause$$

where $n \geq 2$, all id's distinct and ... is a shortening ellipsis extraneous to the meta-language. Let id_i range over D_i and the values of (for suitable substitutions of actual for formal parameters, see below) over D, then the denoted function is in the space:

$$D_1 \rightarrow (D_2 \rightarrow (\ldots \rightarrow (D_n \rightarrow D) \ldots))$$

respectively:

$$D_1\ D_2\ \ldots\ D_n \rightarrow D$$

The former form corresponds to a so-called "currying" or "schönfinckeling" of the more familiar, latter, form. The former form also permits the application of (strictly) less than, as here, n arguments, namely the first (from left-to-right, say) k $(k<n)$ arguments.

Such an application then denotes a function in the space:

$$D_{k+1} \rightarrow (D_{k+2} \rightarrow (\ldots \rightarrow (D_n \rightarrow D) \ldots))$$

where $k+1 \leq n$. [Of course, if D is a itself a functional space, say $D' \rightarrow D''$, then application of n arguments will also yield a function.]

m-ary functions, for $m \geq 0$, can be identified:

$$\underline{let}\ f(id_1, id_2, \ldots, id_m) = clause$$

or:

$$\underline{let}\ f = \lambda(id_1, id_2, \ldots, id_m) \cdot clause$$

where no two id_i and id_j are the same. If f occurs free (see below) in *clause* then the denoted function is the so-called minimal fix-point solution to the (recursive) equation:

$$f(id_1, id_2, \ldots, id_m) = clause$$

etc. The minimal fix-point finding operation, Y, will be explained subsequently. We could have indicated any such intended recursion by instead defining f by:

$$\underline{letrec}\ f(id_1, id_2, \ldots, id_m) = clause$$

respectively:

$$\underline{let}\ f = Y\lambda g.\lambda(id_1, id_2, \ldots, id_m) \cdot clause'$$

where *clause'* arises from *clause* by substituting all free occurrences in *clause* of f by g.

[Using g instead of f on the right-hand side is strictly speaking not necessary, since -- under the *letrec* versus *let* convention -- the left hand f is not bound to any of the right-hand f's.]

We shall, however, always force the user of the meta-language to mean recursion when using the defined function identifier in *clause*. Hence we choose to omit the *rec* suffix on *let*'s and never to use the Y recursion maker. Consequently the only style in which a function can be recursively defined (inside a conventional function definition, see section 5) is by means of a *let* definition which also gives name to the function, a name whose denotation is known within the scope of the meta-language block in which the *let* occurred. This is in contrast to the form :

$$Y\lambda f.\lambda(id_1, id_2, \ldots, id_m) \cdot clause$$

which otherwise defines the same function, but does not further identify it!

Free and Bound Variables, Scope

For the purpose of the definition of the concepts of *free* and *bound* variables we consider the meta-language programs (or: function definitions) as made up from the following three expression constructs:

(1) Variables
(2) Function Applications
(3) Function Abstractions

(1) x is *free* in x.

(2) x is *free* in $f(a)$ if it is *free* in either f, or a, or both.

(3) x is *free* in $\lambda y.clause$ if $x \neq y$ (i.e. if x is an identifier distinct from y) and x is *free* in $clause$.

(4) Since the non-recursive *let* definition block:

$$\begin{array}{l}(\underline{let}\ y = expr\ \ \underline{in} \\ \ \ se)\end{array}$$

where se is either a meta-language statement or expression, can be understood to be a syntactic sugaring of:

$$(\lambda y.se)(expr)$$

provided *call-by-value* is used, we see that *freeness* of x in a *let* definition block follows the rules for *freeness* of x in an application.

(5) Similar for statement composition:

$$\begin{array}{l}(stmt1; \\ \ stmt2)\end{array}$$

only makes sense if $stmt1$ and $stmt2$ both denote functions from states, σ, to states, σ', (stores to stores), hence:

$$(stmt1;\ stmt2)\sigma$$

is really to be understood as:

$$(\underline{let}\ \sigma' = stmt1(\sigma)\ \underline{in}\ stmt2(\sigma'))$$

which then reduces to:

$$(\lambda\sigma'.(stmt2(\sigma')))(stmt1(\sigma))$$

A variable is *bound* in some expression if it occurs in the expression but is not free.

It follows then that the formal parameter variables (identifiers) bind all of the free occurrences of such identifiers in the body, i.e. *clause*, of a λ-expression:

$$\lambda(id_1, id_2, \ldots, id_m).clause$$

The id_i's are called bound variables.

The scope of a bound variable is the entire body-part of the λ-expression to which it belongs, with the specific exception of each properly contained λ-expression of the body-part having that same bound variable identifier.

Application

Applying a λ-defined function:

$$\lambda var.clause$$

to an argument, *arg*:

$$(\lambda var.clause)(arg)$$

then means to evaluate the expression *clause'* which arises from *clause* by substituting all free occurrences of (the syntactic object) the identifier *var* in *clause* by the (semantic) object denoted by *arg*.

Hence the meta-language only has call-by-value.

The above application rule extends to functions of several arguments.

Definition versus Application

To write:

$$(\underline{let}\ f(a) = clause\ \ \underline{in}\ C(f))$$

means: let f denote the function which satisfies the above equation and evaluate or interpret $C(f)$ in such an environment where f is bound to that function. It does, e.g., not mean: evaluate *clause* before proceeding to elaboration of $C(f)$. *clause* is evaluated iff f is applied and then the suitable substituted *clause'* is evaluated everytime, afresh, whenever f is so applied!

The Y Operator

For the benefit of those who are not familiar with the Y operator of the λ-Calculus we now present a development leading up to this operator and its purpose.

Let :

$$(\underline{let}\ f(x) = F(x,f)\ \ \underline{in}\ \ldots)$$

define f recursively -- i.e.: the right-hand side occurrence(s) of f in $F(\ldots)$ denotes the same as does the left-hand side f.

Since, in the λ-Calculus:

$$G = (\lambda x.G)(x)$$

provided x does not occur (free) in G, and since $\underline{let}\ f(x) = F(x,f)$ is the same as $\underline{let}\ f = \lambda x.F(x,f)$ we get:

$$(\underline{let}\ f = (\lambda g.\lambda x.F(x,g))(f)\ \ \underline{in}\ \ldots)$$

where g does not occur (free) in $F(x,f)$. If we ascribe the name F to $\lambda g.\lambda x.F(x,g)$, i.e. if we let:

$$F = \lambda g.\lambda x.\mathsf{F}(x,g)$$

then the previous can be written:

$$(\underline{let}\ f = F(f)\ \ \underline{in}$$
$$\ldots)$$

Now: F is a function, in fact it is a functional. And objects, say w, satisfying:

$$w = Hw$$

where H is any function, are said to be fixed-points of H. The f we are looking for, i.e. defining by the *let* clause, thus is a fixed-point of F. It turns out that there may be many fixed-points of any given functional -- so, following good practice, we select one which is in some way unique; in particular, we choose to let such f's denote the so-called minimal fixed-point. This means: that solution, f, (to the equation) whose graph is included ($\subseteq$) in any other solution. It then turns out that, under suitable and always reasonable constraints (monotonicity, etc.), there exists a functional, let us name it Y, which when applied to objects, like F, produce their minimal fixed-point!

This is the Y we are referring to. It is the so-called minimal fixed-point finding operator. Observe the sequence, repeated from above:

$$\begin{array}{lll}
\underline{let} & f(x) & = \mathsf{F}(x,f) \\
\underline{let} & f & = \lambda x.\mathsf{F}(x,f) \\
\underline{let} & f & = (\lambda g.\lambda x.\mathsf{F}(x,g))(f) \\
\underline{let} & f & = F(f) \\
\underline{let} & f & = \mathsf{Y}F \\
\underline{let} & f & = \mathsf{Y}\lambda g.\lambda x.\mathsf{F}(x,g) \\
\underline{let} & f & = \mathsf{Y}\lambda f.\lambda x.\mathsf{F}(x,f)
\end{array}
\quad \left.\begin{array}{l}\\\\\\\\ \\ \\ \\ \end{array}\right\} \text{where : } F = \lambda g.\lambda x.\mathsf{F}(x,g)$$

That is: writing $\lambda g.\lambda x.\mathsf{F}(x,g)$ or writing $\lambda f.\lambda x.\mathsf{F}(x,f)$ produces identical meanings, since the λ's shield the right-hand side f from the left-hand side f -- they are now, formally speaking, not the same, although they denote the same object!

6.3 FunCTion Operations

There is only defined one operation on FunCTion objects:

(...) apply.

Example 42:

In the introductory example we illustrated the 'concoction' of procedure denotations. In the following example we show their application. Let:

1 *Block :: Var-set (Id $\overset{}{\underset{m}{\rightarrow}}$ Prc) Stmt**

be an abstract syntax (::-) rule. *Block* abstracts the domain of the blocks of some goto-free source language. They consist of three parts: *Var*iable declarations, uniquely named (*Id*) *Pro*cedures, and a *Statement* list. Their semantics is:

type: I-Block: Block $\tilde{\rightarrow}$ (ENV $\tilde{\rightarrow}$ (Σ $\tilde{\rightarrow}$ Σ))

where:

2 *Σ = LOC $\overset{}{\underset{m}{\rightarrow}}$ VAL*
3 *ENV = Id $\overset{}{\underset{m}{\rightarrow}}$ DEN*
4 *DEN = LOC | (ARG* $\tilde{\rightarrow}$ (Σ $\tilde{\rightarrow}$ Σ))*

and:

```
5.0  I-Block(mk-Block(vars,procm,stl))(ρ)(σ)=
 .1     (let (σ',map) = allocate(vars)(σ)                    in
 .2      let ρ' = ρ + map
 .3                 + [id → V-Prc(procm(id))(ρ')(σ')
 .4                       | id ∈ dom procm ]                 in
 .5      let σ'' = I-Stmt-list(stl)(ρ')(σ')                  in
 .6      σ''\rng map)
```

Lines .1-.4 constitute the block activation prologue, line .6 the matching epilogue. The function allocate is given variable names and states and produces a new state in which is allocated all the cells corresponding to the variables, their cell locations is recorded in map which

binds variable names to these locations:

```
6.0     allocate(vars)(σ)=
 .1       if vars={}
 .2          then (σ,[])
 .3          else (let id ∈ vars                                  in
 .4                    let (σ',map) = allocate(vars\{id})(σ)      in
 .5                    let l ∈ LOC be s.t. l ~∈ dom σ'            in
 .6                    (σ' ∪ [l→undefined],map ∪ [id→l]))

        type: allocate:  Var-set → (Σ ~→ (Σ (Var →m LOC)))
```

and:

```
        type: I-Stmt-list:  Stmt* ~→ (ENV ~→ (Σ ~→ Σ)).
```

Now a statement could be a *call* of a procedure:

```
7       Stmt  =  Call  |  ...
8       Call  ::  Id Expr*
```

and with:

```
9.0     I-Stmt-list(stl)(ρ)(σ)=
 .1       if stl=<>
 .2          then σ
 .3          else (let σ' = I-Stmt(h stl)(ρ)(σ)  in
 .4                   I-Stmt-list(t stl)(ρ)(σ'))
```

we end up having to explain:

```
10.0    I-Stmt(stmt)(ρ)(σ)=
  .1       cases stmt:
  .2          (mk-Call(id,el) →
  .3              (let argl = < V-Arg(el[i])(ρ)(σ) | i ∈ ind el >,
  .4                   fct  = ρ(id)                                  in
  .5               fct(argl)),
                ...)
```

where it is assumed that:

```
        type: V-Arg:  Expr ~→ (ENV ~→ (Σ ~→ ARG))
```

in other words, that evaluation of the *arg*ument *l*ist does not create side-effects by changing the state (σ).

Observe now, how the procedure denotation, which is a *FunCT*ion, $(ARG^* \xrightarrow{\sim} (\Sigma \xrightarrow{\sim} \Sigma))$, "packed" in the environment ρ' set up at prologue "time", is retrieved (10.4) and applied (10.5) at "calling" time.

We have completed our task of showing the construction of *FunCT*ion objects, the second of the introductory examples of this section; and the application of such objects to their arguments.

Observe that the apply operator of line .4 above, retrieving *fct* from the *MAP* object ρ, is a *MAP* operation; whereas the apply operator of line .5 above, applying the *FunCT*ion object, *fct*, to its *arg*ument *l*ist, is the *FunCT*ion operation of apply.

7. ABSTRACT SYNTAX

Example 43:

```
Program     =   Stmt
Stmt        =   Block | If | For | Call | Goto |
                Assign | In | Out | NULL
Block       ::  (Id -m-> Type) Proc-set Named-Stmt*
Type        ::  Scalar-type [(Expr | *)+]
Proc        ::  Id Parm* Stmt
Parm        ::  Id (Type | PROC)
Named-Stmt  ::  [Id] Stmt
If          ::  Expr Stmt Stmt
For         ::  Id Expr Expr Expr Stmt
Call        ::  Id (Var-ref | Id)*
Goto        ::  Id
Assign      ::  Var-ref Expr
In          ::  Var-ref
Out         ::  Expr
Expr        =   Infix-expr | Rhs-ref | Con-var-ref | Const
```

The above is an example of an incomplete(d) abstract syntax. It is intended to define fragments of the syntactic domains of some source language. Before commenting on its use of = and :: rules; and of domain operators: |, $\overset{}{\underset{m}{\rightarrow}}$, *, -set and [...], let us, for the sake of instruction, annotate the above syntax in the way we believe a formal model should render itself more open to inspection by casual readers:

-- annotation

A program is a statement .

A statement is either a block, or an if, or a for, or a call, or a goto, or an assign, or an in, or an out, or a null.

A block has a variable declaration part which to variable identifiers uniquely associates the type of the values that can be stored in the variable, a set of procedures, and a list of named statements. Thus a block has three parts. And either, some, or all, of these may be void.

A variable type is either a scalar or an array type. In this model the scalar type is modeled whenever the optional [*(Expr|*)⁺*] object is 'absent', i.e. *nil*. An array type has all array elements being of the same scalar type. The dimension of the array is given by the length of the present *(Expr | *)⁺* list.

A procedure has an identifying name, a parameter specification list, and a body which is a statement.

A parameter specification has an identifying formal parameter name and an argument type description. The type of an argument is either that of a scalar-, an array- or, a procedure.

A named statement has two parts: an optional label, which is an identifier, and a statement.

Etcetera.

Comments

From the above annotations it transpires that the domain expression "|" operator stands for "either", the paired operators "[...]" for optionality, and "*(...)*" delimits the scope of the infix $\underset{m}{\rightarrow}$ in rule 3, the suffix $^{+}$ in rule 4, the infix | in rule 6, and the suffix * in rule 10.

Next we observe that the above syntax defines e.g. the domain *Stmt* recursively. For example: a statement is (=) an if-then-else statement (2); and an if-then-else statement contains both a (then) consequence and an (else) alternative (8), both of which are statements.

Finally we conclude that the domains denoted by the rule left-hand sides must constitute some kind of solution to the mutually recursive set of equations, whether the definition symbol be "=" or "::".

7.1 Domains of Abstract Objects

In the following we shall rely on your returning yourself to the above example, as well as to some of the abstract syntax examples previously shown.

The aim of section 7 is to teach you how to decompose the more general task of defining recursive classes of abstract objects. Sections 2-3-4-5-6 already taught you the more isolated construction of set, tuple, map, tree and function domains.

Motivation:

The reason why we in general, wish to define arbitrarily compounded classes is the following.

The meta-language is used, primarily, to describe complex software architectures: programming languages, data bases, operating system command & control language interfaces; and their stepwise realization: language processors, etcetera. Just like we ordinarily describe the concrete text strings, making up e.g. programs, queries and commands, in terms of e.g. BNF grammars -- so we now desire to give abstractions of these classes of objects. Likewise: just as we, when coding a higher-level language program, a data base data definition, etcetera, define, normally as part of variable declarations, the structure of our internal, stored objects -- so we now desire to give storage layout-independent abstractions of the more intrinsic of these value classes.

Syntactic & Semantic Domains:

In the motivation above, two kinds of abstract object classes were singled out. The ones input to the software systems being defined, viz.: program texts, data base query & update commands, operating system job control language commands, etc. And the ones manipulated, and, as we shall take it in general, denoted by these inputs, viz.: internal data structures; catalogues, files & records; processes, resources, ercetera. We shall generally refer to the former object classes as being of syntactic nature, and the latter as being semantic. Syntactic and semantic domains will, in the meta-language be defined using the programming construct of abstract syntax.

Analogies:

The "programming construct" of BNF grammars or some extended variant thereof is usually applied when defining sets of concrete text strings

-- i.e. context-free "languages". And the type and mode, definition facility of PASCAL, respectively ALGOL 68, is the corresponding programming construct for the internal data structures. In the meta-language the same tool will be applied in the construction of potentially, and usually, infinite classes, or as we shall prefer to call them, domains of abstract objects.

Motivation:

The reason why we apply the same tool and why this tool is neither that of e.g. BNF, the PASCAL type definition nor the ALGOL 68 mode definition facility, is that we, on one hand, neither want to deal with text strings (or corresponding parse trees), nor, on the other hand, are concerned about storage space and access-efficient layout of objects. Instead we desire to provide what we believe to be appropriate & fitting representational abstractions.

Representational Abstractions:

By representational abstraction is meant the target system (e.g. input/output terminal-, or machine-storage device) independent specification of software function concepts, especially objects, emphasizing the choice of such abstractions whose, usually composite, (logical) type most directly expresses intrinsic, i.e. to our understanding relevant, properties of the notions being specified.

Domain Definitions & Compositions

In order:

(1) to decompose the task of constructing domains, i.e. of composing them from constituent domains, and in order

(2) to permit the constructions of reflexive domains, i.e. composite domains whose objects contain properly embedded objects of the same kind (-- or domain)

the meta-language provides the conventional notion of definitions.

Definitions / Rules

A domain definition consists, as does any definition, of two parts. A left- and a right hand side. Other words are lhs, respectively rhs; and definiendum ("that which is being defined"), respectively definiens ("that which defines it"). We shall use the words abstract syntax rule synonymously with the concept of a domain definition.

Abstract syntax rules all have their lhs's being simple names, i.e. identifiers. The rhs's are usually compound expressions henceforth referred to as logical type, or domain expressions.

The first point, above, about decomposing the task of defining domains is achieved by the ability to use, in the rhs's of some rules, the lhs identifier of some other rule. The second point, then, is achieved by permitting the use, in the rhs of any rule, of the lhs identifier of that (or those) rule(s).

Compositions / Domain Expressions

In order:

(3) to model notions of software systems representationally abstract, and in order

(4) to provide a reasonably fitting variety of choices for abstracting objects

the meta-language centers its abstract compound objects around the mathematically tractable concepts of sets, tuples, functions and trees together with associated, primitive operations.

Domain Operators:

As a consequence the meta-language provides a number of operations which apply to arbitrary (constituent, elementary &/or composite) domains to denote set-, tuple-, map-, function- and abstract tree domains.

These operators have already been introduced, and are:

$$-set;\ ^{*}\ ,\ ^{+}\ ;\ \overrightarrow{m}\ ,\ \tilde{\rightarrow}\ ,\ \rightarrow\ ;\ (\ldots)$$

the latter ((...)) denoting (implicit or anonymous) tree construction.

In the compounding, usually signalled by the implicit, or anonymous, tree constructor *(...)*, of more than one domain:

$$(D_1 \quad D_2 \quad \ldots \quad D_n) \qquad\qquad n \geq 2$$

there is an invisible "cartesian product-like" operator, ×. That is, you might read the above as:

$$(D_1 \times D_2 \times \ldots \times D_n).$$

The point about the use in this meta-language of the cartesian operation is that it is only used when domains of trees of two or more subcomponents are denoted. Not, e.g., when tuple domains are to be constructed. These can only be constructed, in this meta-language, using the * or + operations. Thus, where context permits, we may drop implicit tree constructors, as in:

$$A \rightarrow B\ C$$

which is taken to be identical to:

$$A \rightarrow (B\ C)$$

That is: the precedence of × is higher than any other infix operator, but lower than any suffix operator. We usually omit writing the × operator.

The following domain operators assemble not necessarily disjoint domains into 'unions' of these:

| ∪

-- the latter applicable only in connection with map domains, as explained in section 4.1. The | operator will be further defined below.

Finally the operators:

[...] -- optionality
(...) -- grouping/delimiting

enables shortness of descriptions. The former:

$[A] \equiv A \mid \underline{nil}.$

The Scope Delimiting (...) Operator

The latter can easily be confused with the tree construction operator. Therefore: when (...) surrounds two or more domain expressions with no infix domain operator, other than the cartesian operator, separating them, e.g. as in:

$$(A^* \ B\text{-}set \ C),$$

then (...) denotes tree domain construction; otherwise (...) as in :

$$(A^* \mid (B\text{-}set \underset{m}{\rightarrow} C))^*,$$

serves to indicate, i.e. delimit, the scope of infix or suffix domain operators. The corresponding scope rules are the conventional ones.

Infix Operator Commutativity and Associativity

With infix operators the question of their commutativity and associativity arises:

$\mid$, $\underset{\sim}{\cup}$	are commutative
$\underset{m}{\rightarrow}$, $\underset{\sim}{\rightarrow}$, $\rightarrow$	associates to the right.

The cartesian product operator, $\times$, does not associate!

Examples 44-45-46:

(1) The domains denoted by:

$$A \mid B \quad \text{and} \quad B \mid A$$

are identically the same, and so are:

$$(A \underset{m}{\rightarrow} B) \underset{\sim}{\cup} (C \underset{m}{\rightarrow} D) \ , \quad \text{and :}$$
$$(C \underset{m}{\rightarrow} D) \underset{\sim}{\cup} (A \underset{m}{\rightarrow} B).$$

(2) The domains:

$$A \rightarrow (B \rightarrow (C \rightarrow D)) \ , \quad \text{and :}$$
$$A \rightarrow B \rightarrow C \rightarrow D$$

are identically the same. Writing the former is a clarification of the

latter, should you have forgotten the rules of associativity. Thus the domain:

$$((A \to B) \to C) \to D$$

is distinct from the domain denoted by the former expressions.

(3) The tree domains:

$$(A \times B \times C),\ ((A \times B) \times C),\ \text{and}\ (A \times (B \times C))$$

are all distinct (in fact, disjoint). Only in this, the 3rd example, was (...) used for tree domain construction.

Semantics of the | Operator

Of the new domain expression operators ([...], (...), |) introduced in section 7 only the meaning of | remains to be explained.

Let A and B be domain expressions, denoting $\mathbb{A}$, respectively $\mathbb{B}$. Then:

$$A \mid B$$

denotes:

$$\mathbb{A} \cup \mathbb{B}$$

where $\cup$ is the (potentially infinite set) union operator; i.e.:

$$A \mid B \quad \triangleq \quad \{ o \mid o \in \mathbb{A} \vee o \in \mathbb{B} \}$$

Comment:

Thus | does not denote the discriminated or disjoint union operation.

In Scott 76 the disjoint union operator is represented by +, and:

$$A + B \quad \triangleq \quad \{ (\underline{A}, a) \mid a \in \mathbb{A} \} \cup \{ (\underline{B}, b) \mid \in \mathbb{B} \}$$

The "tagging" of the $A+B$ domain elements, with a "label" indicating the originating domain, is useful whenever (1) domains A and B overlap, and (2) the need arises for testing, say in some function, whether an $A+B$ object comes from A or from B.

Since the | operator does not separate the operand domains it is the task of the definer to see to it that | operand domains are disjoint whenever some functions applicable to the union domain need ascertain originating domain.

Constructing Disjoint Domains

Let us assume that a union domain, roughly of the type:

$$D_1 \mid D_2 \mid \ldots \mid D_n$$

is required. Let us further assume that for some, or all, $i \neq j$:

$$D_i \text{ is } \underline{\text{not}} \text{ disjoint from } D_j$$

Since D_i and D_j overlap there is no direct way of separating (some) D_i objects from D_j.

Introducing the abstract syntax tree domain constructing rules:

$$A_1 :: D_1$$
$$A_2 :: D_2$$
$$\ldots$$
$$A_n :: D_n$$

and :

$$A_1 \mid A_2 \mid \ldots \mid A_n$$

(in lieu of $D_1 \mid D_2 \mid \ldots \mid D_n$), effectively corresponds to a distinct marking ($mk\text{-}A_1$, $mk\text{-}A_2$, ..., $mk\text{-}A_n$) of respective D_1, D_2, ..., D_n objects in the union domain $A_1 \mid A_2 \mid \ldots \mid A_n$. Distinctness is solely achieved by distinctness of left-hand side (A_i, A_j) names.

Thus we introduce *::* abstract syntax rules whenever disjointness is desired.

7.2 Abstract Syntaxes & Rules

Abstract Syntaxes

```
Abstract-Syntax    ::=    Rule1
                          Rule2
                          ...
                          Ruler
```

is an informal "extended" BNF, or BNF-like, syntactical description defining an abstract syntax as consisting of a number of rules. Each rule:

```
Rule               ::=    Identifier = Domain-Expression
                   ::=    Identifier :: Domain-Expression
```

basically consists of two parts: a left-hand side definiendum, which is an identifier; and a right-hand side definiens, which is a domain expression. The two rule parts are separated either by an equality *(=)* operator or by the so-called tree domain constructor *(::)* operator. Both the $=$ and the $::$ operators are definition symbols (like: $\triangleq$ or $\stackrel{=}{\text{def}}$). The latter, in addition, 'constructs' trees. Its use in:

$$A :: B_1\ B_2\ \ldots\ B_n$$

could be taken as an abbreviation for the (intended, reflexive) use of $::$ in:

$$A = ::(B_1\ B_2\ \ldots\ B_n)\ .$$

Context Constraint

In a complete meta-language program there must be no two rules of any one or pair of abstract syntaxes with identical definienda, i.e. lhs identifiers.

is-Function:

Any abstract syntax rule implicitly introduces, i.e. defines, a predicate function *is*-Identifier:

type: *is*-Identifier: *OBJ* → *BOOL*

which applies to any object and yields *true* if it is an object in the domain denoted by Identifier, otherwise *false*.

Note:

We could have chosen to write:

obj ∈ Identifier

instead of:

is-Identifier(*obj*)

but, except for a few cases, we reserve the former form for set membership tests where the set is not one defined as a Domain as defined by some abstract syntax. The exceptions are those illustrated by the following descriptor and quantified expressions (see sections 1.5 & 1.4.1):

(Δ*x* ∈ Identifier)(*P*(*x*))
(∀*x* ∈ Identifier)(*P*(*x*))
(∃*x* ∈ Identifier)(*P*(*x*))
(∃!*x* ∈ Identifier)(*P*(*x*))

Example 47:

Given the introductory abstract syntax example, and given: *is-Stmt(s)*, *is-ID(id)*, *is-Scalar Type(st)*, *is-Named-Stmt(ns1)*, *is-Named-Stmt(ns2)*, *is-Stmt(s1)*, *is-Stmt(s2)*, and *is-Expr(e)* we are able to construct:

mk-Program(s)	*pr*
mk-Block([id→mk-Type(st,nil)],{},<ns1,ns2>)	*bl*
mk-Named-Stmt(nil,s)	*ns*
mk-If(e,s1,s2)	*if*

etc.. Let us call these four objects *pr*, *bl*, *ns* and *if*. Now:

is-Program(pr)	~ *is-Program(bl)*
~ *is-Stmt(pr)*	*is-Stmt(bl)*
~ *is-Block(pr)*	*is-Block(bl)*
~ *is-Named-Stmt(pr)*	~ *is-Named-Stmt(bl)*
~ *is-If(pr)*	~ *is-If(bl)*
~ *is-Program(ns)*	~ *is-Program(if)*
~ *is-Stmt(ns)*	*is-Stmt(if)*
~ *is-Block(ns)*	~ *is-Block(if)*
is-Named-Stmt(ns)	~ *is-Named-Stmt(if)*
~ *is-If(ns)*	*is-If(if)*

etc..

Further Context Constraint

A =-rule, when expressible in the form:

$$Identifier \quad = \quad D_1 \; D_2 \; \ldots \; D_n \qquad (n \geq 2)$$

is identical to the rule:

$$Identifier \quad :: \quad D_1 \; D_2 \; \ldots \; D_n \qquad (n \geq 2)$$

Semantics of Abstract Syntaxes

The prescription for computing most of the domains designated by compound domain expressions has already been given. Allowing now for domain definitions, i.e. rules, in fact for several such, and, in addition, for potentially mutually recursive rules, we require a more comprehensive prescription for computing the domains given an arbitrary abstract syntax. We shall only do that informally here, relying in general on the formal foundations laid by Scott. [Scott 1976].

An abstract syntax can, speaking rather formally, be viewed syntactically as an equation set. Its meaning is a family of named sets of mathematical objects being the minimal, fix-point solution to the equation set. That is: to each lhs there corresponds a potentially infinite class of finite objects, the domain. These domains are the smallest such classes which, when substituted in lieu of their identifying names

in the equation lhs and rhs's will, satisfy the equations. The bit about the fix-point comes into the picture since our equations may be mutually recursive.

Example 48:

The equation:

$$D \;=\; D \underset{m}{\rightarrow} D$$

has the following solution:

$$D: \quad \{[],[[]\rightarrow[]],[[]\rightarrow[[]\rightarrow[]]],[[[]\rightarrow[]]\rightarrow[]],\ldots\};$$

with :

$$D \;=\; (D \underset{m}{\rightarrow} D) \mid A$$

"adding":

$$\begin{array}{l}\{[a_1 \rightarrow []],\ldots \\ \;[[] \rightarrow a_1],\ldots \\ \;[a_1 \rightarrow a_1],\ldots \\ \;[a_1 \rightarrow a_1, a_2 \rightarrow a_2,\ldots,a_n \rightarrow a_n],\ldots \\ \;\ldots\}\end{array}$$

to the above solution.

7.3 Abstract Syntax oriented Combinators

The meta-language (structured-) expression and statement combinators specifically complementing the (abstract syntax) domain expression alternative operator, |, are:

The McCarthy Conditional Clause:

$$
\begin{array}{l}
(pe_1 \rightarrow c_1, \\
\ pe_2 \rightarrow c_2, \\
\ \ \ldots \\
\ pe_n \rightarrow c_n)
\end{array}
$$

The Cases Conditional Clause:

$$
\begin{array}{l}
\underline{cases}\ e_0: \\
\quad (e_1 \rightarrow c_1, \\
\ \ e_1 \rightarrow c_2, \\
\ \ \ldots \\
\ \ e_n \rightarrow c_n)
\end{array}
$$

Here pe_i stand for predicate expressions, c_j for either expressions or statement, and e_k for expressions. A clause is either a statement or an expression. If either of the above conditional clauses is (intended to be) of the expression type, then all c_j are (to be) expressions. Similarly they are all to be statements if the above clauses are to be conditional statements. We refer to the e_k expressions as follows: e_0 as the root expression and e_k, for $1 \leq k \leq n$, as the branch expressions.

Example 49:

Given the abstract syntax for expressions of some source language:

```
1    Expr          =   Infix | Rhs-ref | Con-var-ref | Const
2    Infix         ::  Expr Op Expr
3    Rhs-ref       ::  Var-ref
4    Con-var-ref   ::  Id
5    Const         =   INTG | BOOL
6    Var-ref       ::  Id [Expr+]
7    Op            =   Int-Op | Bool-Op | Rel-Op
```

We can define a function, *ex-tp*, which given a dictionary:

```
8    DICT          =   Id →m (Type | PROC | ...)
9    Type          ::  Scalar-type [(Expr | *)+]
10   Scalar-type   =   INT | BOOL
```

computes the *Scalar-type* of an expression, *e*:

```
11.0  ex-tp(e,dict)=
  .1    cases e
  .2      (mk-Infix( ,op, ) →
  .3            (is-Int-Op(op)   → INT,
  .4             is-Bool-Op(op)  → BOOL,
  .5             is-Rel-Op(op)   → BOOL),
  .6       mk-Rhs-ref(mk-Var-ref(id, )) →
  .7             s-Scalar-type(dict(id)),
  .8       mk-Con-var-ref( ) →
  .9             INT,
 .10       T  → (is-INTG(e)      → INT,
 .11             is-BOOL(e)      → BOOL))

        type:  Expr DICT → Scalar-type
```

Observe how, in this case, we nested the Cases and the McCarthy constructs. Relate the structures of the abstract syntax and the *ex-tp* function, and observe how they "match". Observe next the use of *mk-* constructs in the cases branch expressions. Observe finally that we only name those arguments of the *mk-* functions which we explicitly require.

-- annotations:

An expression is either an infix expression, a right-hand-side reference, a controlled variable reference or a constant.

An infix expression has three parts: two operand expressions and an operator.

A right-hand-side variable reference is a variable reference.

A controlled variable reference is an identifier.

A constant is either an integer or a boolean.

A variable reference has an identifier and, if this identifier denotes an array location, then an index list, which is a non-zero, finite length expression list, else nil.

An operator is either an integer (arithmetic) operator, a boolean operator, or a relational operator.

-- comments:

An expression either denotes an integer or a boolean. Hence the type of an expression is (said) either (to be) *INT* or (to be) *BOOL*. The type of an expression -- which is otherwise well-formed -- can be statically ascertained. Given that the type with which a variable or formal parameter is declared:

12 *Block :: (Id* $\underset{m}{\rightarrow}$ *Type) Proc-set Named-Stmt**

13 *Proc :: Id Parm** *Stmt*

14 *Parm :: Id (Type | PROC)*

is recorded in some (compile-time) dictionary, also called static environment, the function ex-tp computes (at compile-time) the type of any expression.

-- annotations, continued:

ex-tp is given an expression, *e*, and a dictionary, *dict*.

If *e* (.1) is an infix expression (.2), i.e. if *e* can be expressed as some *mk-Infix(e1,op,e2)*, where *e1,op* and *e2* become the names of those *Expr, Op,* respectively *Expr* objects of which *e* is made up, then the type of *e* is ascertainable from the kind of operator that *op* is. If *op* is .3) an integer operator, then the type of *e* is *INT*; if *op* instead is either (.4) a boolean or (.5) a relational (or comparison) operator, then the type of *e* is *BOOL*.

If *e* (.1) is a right-hand side reference (.6), i.e. if *e* can be expressed as some *mk-Rhs-ref(vr)*, where *vr* is a variable reference (meta-)expressible as some *mk-Var-ref(id,el)*, where *id* and *el* become the names of the *Id*, respectively [*Expr*$^+$] objects of which *e* can be made up, then the type of *e* can be looked up in the dictionary as that

of the *Scalar-type* component of the *Type* object with which *id* is associated.

PART III COMBINATORS

In part II, i.e. in sections 2-7 inclusive, we dealt with all aspects of constructing domains of objects, constructing objects and performing primitive, i.e. language-defined operations on objects.

Certain desired transformations on objects are, however, of a complexity which cannot easily be described by some such operator/operand expression, regardless of its composition. To that end, and as is quite customary in programming, the meta-language provides a number of constructs which facilitate the gradual composition of transformations and processes on objects. We call these constructs for combinators. They are:

Variables: Declarations, Assignment & the State
Structured Clauses
Blocks: Let & Return
Exits

So far we have officially dealt only with the applicative aspects of the meta-language. Introducing variables implies introducing imperative constructs, i.e. statements. The applicative part of the language, however, includes the *let* constructs of, and hence also, blocks, as well as most of the structured clauses. We say, in general, that the applicative constructs of the language permit the expression of composite transformations on objects. The imperative constructs correspondingly enable the decomposed, stepwise statement of processes on objects.

8. VARIABLES

Example 50:

The following four function definitions all define f_i to denote the factorial function:

```
1    f0(n)  =  if n=0 then 1 else n*f0(n-1)

2    f1(n)  =  (dcl v := 1 type N1;
 .1             for i=1 to n do
 .2                 v := (c v)*i;
 .3             return(c v))

3    f2(n)  =  (dcl v := 1 type N1;
 .1             for all i ∈ {1:n} do
 .2                 v := (c v)*i;
 .3             return(c v))

4    f3(n)  =  (dcl v := 1 type N1,
 .1                 i := n type N0;
 .2             while (c i ≠ 0) do
 .3                  (v := (c v)*(c i);
 .4                   i := (c i) - 1);
 .5             return(c v))
```

The first defines f_i applicatively; the remaining three, imperatively.

We say that the parenthesized constructs: *(dcl ... return(...))* are blocks. v and, only in the last definition, i are (assignable) variables. They form part or all of the state current in these blocks. If no other (externally declared) variables 'exist', then they form all of the state. The logical type of the four f's are:

type: f_0: $N_0 \to N_1$

$$\underline{type}: \quad f_1, f_2, f_3: \quad N_0 \to (\Sigma \to (\Sigma\ N_1))$$

where Σ denotes the state space. The contributions of $\underset{\sim}{v}$ and $\underset{\sim}{i}$, inside the blocks, to this space is:

$$\Sigma \quad = \quad \ldots \underline{\cup} (\underset{\sim}{v} \overset{\rightarrow}{\underset{\sim}{m}} N_1) \underline{\cup} (\underset{\sim}{i} \overset{\rightarrow}{\underset{\sim}{m}} N_0)$$

where the ... ellipsis refers to possibly externally declared variables. Since none of the f_1, f_2 nor f_3 assign to any such global variables it is easy to see that they do not alter any global state, and hence that the potential state transformation indicated by:

$$\Sigma \to \Sigma$$

is in fact the identity change. Thus the four functions can indeed be shown 'equivalent' even when considering a global state.

8.1 Declarations & The State

Although the examples above featured block local declarations we shall normally not find a need for other than global variables in abstract models of higher-level software.

The meaning of a declaration:

$$\underline{dcl}\ \underset{\sim}{Var} := \ldots\ \underline{type}\ D$$

-- in which we may omit the $\underline{type}\ D$ clause -- is that of joining to our state, σ, a contribution:

$$\sigma \cup [\underset{\sim}{Var} \to obj]$$

It is understood that $\underset{\sim}{Var}$ is not already declared, i.e. that $\underset{\sim}{Var} \sim\in \underline{dom}\ \sigma$. In general, the state space, Σ, given a collection of (global) declarations:

$$\begin{array}{l} \underline{dcl}\ \underset{\sim}{V}_1 := \ldots\ \underline{type}\ D_1, \\ \quad\ \underset{\sim}{V}_2 := \ldots\ \underline{type}\ D_2, \\ \quad\ \ldots \\ \quad\ \underset{\sim}{V}_n := \ldots\ \underline{type}\ D_n; \end{array}$$

can be defined as:

$$\Sigma = (\underset{\sim}{V}_1 \xrightarrow[m]{} D_1) \cup (\underset{\sim}{V}_2 \xrightarrow[m]{} D_2) \cup \ldots \cup (\underset{\sim}{V}_n \xrightarrow[m]{} D_n)$$

The domain expressions:

$$(\underset{\sim}{V}_i \xrightarrow[m]{} D_i)$$

are degenerate in that the denoted map domains are singular, i.e. consist of just one element, the (quotation-like) object $\underset{\sim}{V}_i$ which can be regarded as denoting itself.

Example 51:

Our source language example features the ability to input data from an (one) external device, to output data to an (one) external device -- distinct from the input device, and to declare variables in any (nested) block. The relevant syntactic domains, continuing example 31, are:

```
Block  ::  (Id ->m Type) ... Stmt*
Stmt   =    In | Out | Assign | ...
In     ::   Var-ref
Out    ::   Expr
Assign ::   Var-ref Expr
```

The semantic functions ascribing meaning to these source constructs will be based on the following three (global) variables:

```
del Input  := <...> type (INTG|BOOL)*,
    Output := <>    type (INTG|BOOL)*,
    STG    := []    type STG;
```

where *STG* was defined in the last example of section 4.4.

The state space, Σ, of the elaboration functions, is:

$$\begin{aligned}\Sigma = {} & (\mathit{Input} \xrightarrow[m]{} (\mathit{INTG}|\mathit{BOOL})^*) \cup \\ & (\mathit{Output} \xrightarrow[m]{} (\mathit{INTG}|\mathit{BOOL})^*) \cup \\ & (\mathit{STG} \xrightarrow[m]{} \mathit{STG})\end{aligned}$$

8.2 Variable References

Prelude

Recall the introductory examples of section 8. Variable identifiers ($\underset{\sim}{V}$) occurred in two contexts:

$$\underset{\sim}{V} := \ \ldots \ \underline{c}\ \underset{\sim}{V} \ \ldots$$

The left-hand side occurrence denotes itself, i.e. the meta-storage (state) location. The right-hand side occurrence of $\underset{\sim}{V}$ likewise denotes itself ! But here we traditionally expect the value kept in the $\underset{\sim}{V}$ location. The meta-language breaks with this tradition. If you need the content, then you are required to perform the $\underline{c}$ operation to $\underset{\sim}{V}$.

end-of-Prelude

If $\underset{\sim}{V}$ is the name of a declared variable:

$$\underline{dcl}\ \underset{\sim}{V} := \ldots\ \underline{type}\ D$$

then $\underset{\sim}{V}$ is said to denote itself, or if need arises for more precision, to denote a $\underline{ref}\ D$ object, i.e. a reference to an object of type D.

To get at the value 'stored' in $\underset{\sim}{V}$ apply the contents-of operation denoted by $\underline{c}$:

$$\underline{c}\ \underset{\sim}{V}$$

$\underset{\sim}{V}$ and $\underline{c}\ \underset{\sim}{V}$ are meta-language expressions. Given the state, σ:

$$\sigma \ = \ [\underset{\sim}{V} \rightarrow obj, \ \ldots \ ]$$

$\underline{c}\ \underset{\sim}{V}$ is explained as:

$$\sigma(\underset{\sim}{V}).$$

If $\underset{\sim}{V}$ is of type $\underline{ref}\ D$, then $\underline{c}\ \underset{\sim}{V}$ is of type D. That is: $\underline{c}$ de-references $\underset{\sim}{V}$. Thus $\underline{c}$ denotes a function from state variables and states to objects; which we generalize to a function from state-variable and states to states 'paired' with objects:

$$\underline{c} \quad \sim \quad \lambda\upsilon.\lambda\sigma.(\sigma,\sigma(\upsilon))$$

$$\underline{c} \quad \in \quad (\mathbb{V} \to (\Sigma \xrightarrow{\sim} (\Sigma\ OBJ)))$$

8.3 Assignment

Assignment is a meta-language statement:

$$\underset{\sim}{V} := expr$$

If $\underset{\sim}{V}$ is of type $\underline{ref}\ D$ then *expr* must be of a type D', included in D, i.e. $D' \underline{\subseteq} D$, e.g. $D \equiv D'$

The meaning of assignment is as you think it. Formally, however:

$$:= \quad \sim \quad \lambda\upsilon.\lambda obj.\lambda\sigma.(\sigma\dagger[\upsilon \to obj])$$

i.e.: $:= \quad \in \quad (\mathbb{V} \to (OBJ \to (\Sigma \xrightarrow{\sim} \Sigma)))$

Example 52:

We now conclude the examples of sections 4.4, 6.3 and 8.1. Recall in particular the last example of section 4.4.

The elaboration functions ascribing meaning to the source language *In*, *Out* and *Assign* statements will themselves employ, but now, meta-language assignments. Since the basic source language means of referring to a variable is through:

$$Var\text{-}ref \ :: \ Id\ [Expr^{+}]$$

we define a set of auxiliary functions:

$$\begin{array}{lll} \underline{type}: & eval\text{-}Var\text{-}ref: & Var\text{-}ref \xrightarrow{\sim} (ENV \xrightarrow{\sim} (\Sigma \xrightarrow{\sim} (\Sigma\ Scalar\text{-}loc)) \\ & contents: & Scalar\text{-}loc \xrightarrow{\sim} (\Sigma \xrightarrow{\sim} (\Sigma\ (INTG|BOOL))) \\ & assign: & Scalar\text{-}loc\ (INTG|BOOL) \xrightarrow{\sim} (\Sigma \xrightarrow{\sim} \Sigma) \end{array}$$

eval-Var-ref takes a variable reference, an environment and a state, and produces the same state and the scalar location denoted by the variable reference. The need for accessing the state arises as a result

of potentially evaluating array variable subscripts. Whereas *eval-Var-ref* applies to a syntactic object (*Var-ref*), and hence needs the environment; *contents* and *assign* apply only to semantic objects, and hence do not require the environment. Finally: objects assigned to storage locations must have their type match the type of the location. Recall that the storage model of the last example of section 4.4 distinguished between integer- and boolean locations:

```
0        v-tp-match(v, sloc)=
 .1          (is-INTG(v) → is-Intg-loc(sloc),
 .2           is-BOOL(v) → is-Bool-loc(sloc))
         type:  (INTG|BOOL) (Intg-loc|Bool-loc) → BOOL
```

Now:

```
1. int-In(mk-In(vr))(ρ)=
   .1  if c Input = <>
   .2    then error
   .3    else (let sloc : eval-Var-ref(vr)(ρ),
   .4              iv   : hd c Input;
   .5          if v-tp-match(iv,sloc)
   .6             then (Input := tl c Input;
   .7                   assign(sloc,iv))
   .8             else error)
       type: In ~→ (ENV ~→ (Σ ~→ Σ))


2. int-Out(mk-Out(e))(ρ)=
   .1  (let ov : eval-Expr(e)(ρ);
   .2   Output := (c Output)⌢<ov>)
       type:  Out ~→ (ENV ~→ (Σ ~→ Σ))


3. int-Assign(mk-Assign(vr,e))(ρ)=
   .1  (let sloc : eval-Var-ref(vr)(ρ),
   .2       val  : eval-Expr(e)(ρ);
   .3   assign(sloc,val))
       type: Assign ~→ (ENV ~→ (Σ ~→ Σ))
```

The auxiliary functions:

```
4. eval-Var-ref(mk-Var-ref(id,ssl))(ρ)=
   .1  if ssl=nil
   .2     then return(ρ(id))
   .3     else (let arrloc = ρ(id)  in
   .4           let il      : < eval-Expr(ssl[i])(ρ) | 1 ≤ i ≤ len ssl >;
   .5           if  il  ~∈ dom arrloc
   .6               then error
   .7               else return(arrloc(il)))


5. contents(l)=
   .1  (let v : (c STG)(l);
   .2   if v=undefined
   .3      then error
   .4      else return(v))


6. assign(l,v)=
 .1     STG := c STG + [l→v]
```

-- annotations

1. *Int*erpreting an *In*put statement proceeds as follows: (.1) If there is no more *Input*, then (.2) interpretation halts, otherwise (.3-.8) we *eval*uate (.3) the scalar *loc*ation denoted by the variable reference constituting the *In*put statement and (.4) retrieve the front *in*put stream value. If (.5) the *type* of this value *match*es that of the scalar location then (.6-.7) we proceed to (.6) 'shorten' the *Input* stream by the element value just retrieved, and (.7) to *assign* this value to the scalar storage *l*ocation; otherwise interpretation halts.

2. *Int*erpreting an *Out*put statement consists, rather more simply, of (.1) *eval*uating the expression it is composed of and (.2) of appending the resulting output value to the *Output* stream.

Etc. for 3., *int-Assign*.

4. *Eval*uating a *var*iable *ref*erence which consists of a possibly void array *sub*script *list* and the variable *id*entifier proceeds as follows: (.1) If the *sub*script *list* is absent, then (.2) the *id*entifier is guaranteed, by well-formedness context conditions not shown, to denote a scalar variable, whose location can be obtained from the environment directly. (.3-.7) If instead the *sub*script *list* is present, then (.3) the *id*entifier denotes an *arr*ay variable whose *loc*ation is kept in the environment. The *sub*script *list* is guaranteed to consist of proper *expr*essions all yielding integer values. Well-formedness context conditions for this are not shown, but see [Jones 1978a]. We therefore (.4) *eval*uate all the subscript *expr*essions to obtain an *index list*. (.5) If this *index list* is not one of the array, then (.6) interpretation fails, otherwise the denoted scalar location is obtained by applying the *arr*ay *loc*ation, which is a map from index lists to scalar locations, to the computed *index list*.

Comments

In the above definitions certain 'pairs' of evaluations proceeded in "parallel": 1.3-1.4, 3.1-3.2. In [Jones 1978a] the same evaluations proceed sequentially, in the order listed. This gives rise to two distinct semantics!

8.4 Derived References

Section 8.2 dealt only with simple variable references. That is: references denoting the location of an 'entire' variable. In the design and use, since 1973, of the meta-language, a need, when abstracting software, has not been registered for easily denoting references to proper components of store composite objects.

When applying the meta-language to lower-level abstractions, i.e. to rather more implementation-biased specifications, such a need may arise. The following notation is therefore offered.

Sub-References to *TUPLE* Elements

Given:

(0) *dcl* Tuple := <...> *type* D^*

Retrieving the i'th component of *c* Tuple is basically expressible as:

(1) (*c* Tuple)[i]

with the selective update of that tuple position basically expressible as:

(2) Tuple := *c* Tuple + [$i \rightarrow d$];

It is now suggested to let:

(3) Tuple$^{\cdot}$[i]

denote the reference to the i'th position of *c* Tuple, permitting (1) to be rewritten:

(1') *c*(Tuple$^{\cdot}$[i])

and (2) as:

(2') Tuple$^{\cdot}$[i] := d

Sub-References to *MAP* Range Elements

The forms corresponding to (1,2,3,1',2') above, but for variables of type map:

(0) *dcl* Map := [...] *type* $A \overset{m}{\rightarrow} D$;

are:

(1) (*c* Map)(a)

(2) Map := *c* Map + [$a \rightarrow b$]

(3) Map$^{\cdot}$(a)

(1') *c*(Map$^{\cdot}$(a))

(2') Map$^{\cdot}$(a) := b

-- where $a \in \underline{dom}\ c\ \underline{Map}$ prior to (2).

Sub-References to Sub-"*TREE*"s.

Given:

$$D :: s\text{-}nm_1{:}D_1\ s\text{-}nm_2{:}D_2\ \ldots\ s\text{-}nm_l{:}D_l$$

and:

(0) $\underline{dcl}\ \underline{Tree} := mk\text{-}D(\ldots)\ \underline{type}\ D.$

The forms corresponding to (1,2,3,1',2') above are now:

(1) $s\text{-}nm_i(\underline{c}\ \underline{Tree})$ $\qquad 1 \leq i \leq l$

(2) $(\underline{let}\ mk\text{-}D(d_1,\ldots,d_i,\ldots,d_l) := \underline{c}\ \underline{Tree};$
$\underline{Tree} := mk\text{-}D(d_1,\ldots,d_i',\ldots,d_l))$

where all d_j for $j \neq i$ are unchanged, i.e. where (2) only 'changes' the $s\text{-}nm_i$ sub-"tree". Now:

(3) $s\text{-}nm_i{}^{\bullet}\underline{Tree}$

(1') $\underline{c}(s\text{-}nm_i{}^{\bullet}\underline{Tree})$

(2') $s\text{-}nm_i{}^{\bullet}\underline{Tree} := d_i'$

Discussion

The reader may verify that a similar need for variables containing sets does not arise.

Also: the benefit of the derived reference operation, denoted by •, is higher in forms (2') than (1') -- cf. forms (2) and (1), respectively.

Finally: the semantics of the derived reference operator, •, is fully explained by the equivalence of forms (1) and (1'), (2) and (2'), respectively.

9 STRUCTURED CLAUSES

As used in this section a clause is either a statement or an expression.

It turns out that to each structured statement, and block, see next section, there corresponds a structured expression. The conditional form of such clauses syntactically looks very much alike, whereas this is not the case for the iterative clauses.

9.1 Overview

Let in the following s and $S(...)$ stand for statements, $E(...)$ and suitably decorated e's for expressions, suitably decorated pe's for predicate expressions, and suitably decorated c's for clauses. It is further assumed that the c's of a given conditional structure are either all statements, or all expressions, leading to this conditional in turn being a statement, respectively an expression.

The following presents the various structured clauses in a schematic way:

Conditional Clauses

(1) *if* pe *then* c_1 *else* c_2

(2)
$(pe_1 \rightarrow c_1,$
$pe_2 \rightarrow c_2,$
$\ldots$
$pe_n \rightarrow c_n)$

and (2')
$(pe_1 \rightarrow c_1,$
$pe_2 \rightarrow c_2,$
$\ldots$
$T \rightarrow c_n)$

(3) *cases* e_0:
$(e_1 \rightarrow c_1,$
$e_2 \rightarrow c_2,$
$..$
$e_n \rightarrow c_n)$

and (3') *cases* e_0:
$(e_1 \rightarrow c_1,$
$e_2 \rightarrow c_2,$
$..$
$T \rightarrow c_n)$

Iterative Clauses:

(4) *for i = m to n do S(i)* (4') *<E(i) | m≤i≤n>*

(5) *for all id∈set do S(id)*
- (5') *{E(id) | id ∈ set}*
- (5") *[id → E(map(id)) | id∈dom map]*

(6) *while pe do s* (6')

```
f(obj)=
   if P(obj)
      then F(obj)
      else G(obj,f(H(obj)))
```

where in the last schema ((6')) *P*, *F*, *G* and *H* are appropriate functions, *P* being a predicate. (4, 5, 6) are statements; (4', 5', 5") are expressions, and so is an application of *f*.

Examples -- 53:

Illustrations of the duality between clauses (4) and (4'), respectively (5) and (5'), (5"), have already been given -- see sections 3.4, respectively 2.4, 4.4.

The duality of the conditional statements and conditional expressions need not be further discussed.

The following examples illustrate the duality of forms (6) and (6').

To sum the elements of a tuple, *t*, of integers:

```
(del  q   := t type INTG*,
      sum := 0 type INTG;
 while (c q) ≠ <> do
     (sum := (c sum) + hd c q;
      q   := tl c q)
 return(c sum))
```

```
sum(t)=
  if t = <>
     then
     else hd t + sum(tl t)
```

Subsequent elaboration functions will illustrate the power and neatness of recursive definitions centered around a simple *if*-*then*-*else* clause.

9.2 Detailed Syntax & Semantics

9.2.1 If-then-else Conditional

Schema:

if expr	predicate-expression
then clause	consequent
else clause	alternative

Meaning:

Elaboration of an if-then-else clause proceeds as follows: first the premiss- (or test-) predicate expression is evaluated. If it does not yield a truth value an *error* has occurred -- and the meta-language program is in error! If the premiss yields truth then the consequent clause is elaborated. Otherwise the alternative clause. Any elaboration of either the consequent- or the alternative clause strictly succeeds the evaluation of the premiss.

Programming Notes:

1. Observe that the form :

   if expr then stmt

   is not provided. Instead the programmer, if so forced, is adviced to use:

   if expr then stmt else I

   for *I* see section 1.2. Note that the converse is possible:

   if ~expr then I else stmt

   and that the 'similar' problem is not relevant for if-then-... expressions!

2. The premiss sometimes relates as follows to either the consequent or alternative clause:

```
if (∃x...)(P(x))
   then (let x be s.t. P(x);
         eC(x))
   else ...
```

or, e.g.:

```
if~(∃!y...)(P(y))
   then ...
   else (let z = (Δx)(P(x));
         aC(z))
```

Whereever the 'connection' is so obvious, then we informally, but strictly speaking erroneously, write:

```
if (∃x...)(P(x))
   then eC(x)
   else ...
```

respectively:

```
if~(∃!x...)(P(x))
   then ...
   else aC(x)
```

etcetera. The 'shifts' in identifier naming is, of course, immaterial. Thus the x of the latter two premisses (informally) bind as far as the entire *e*consequent and *a*lternative clauses.

9.2.2 McCarthy Conditional

Schema:

$$(e_1 \rightarrow es_1,\ e_2 \rightarrow es_2,\ \ldots\ e_n \rightarrow es_n) \qquad n \leq 2$$

and:

$(e_1 \rightarrow es_1,$
$e_2 \rightarrow es_2,$
$\ldots$
$T \rightarrow es_n)$ $\qquad n \geq 2$

where the T of the latter form is a meta-language keyword only used in McCarthy and Cases constructs in the e_n 'position', i.e. in lieu of e_n. All e_i $(1 \leq i \leq n)$ are predicate (or propositional) expressions.

Meaning:

We give the semantics of the above in terms of their transcription into the if-then-else form:

$(\underline{if}\ e_1\ \underline{then}\ es_1$
$\underline{else}$
$(\underline{if}\ e_2\ \underline{then}\ es_2$
$\underline{else}$
$($ $\ldots$
$\underline{else}$
$(\underline{if}\ e_n\ \underline{then}\ es_n$
$\underline{else}\ \underline{error})\ldots)))$

respectively:

$(\underline{if}\ e_1\ \underline{then}\ es_1$
$\underline{else}$
$(\underline{if}\ e_2\ \underline{then}$
$($ $\ldots$
$\ldots$
$\underline{else}\ es_n\ldots)))$

Programming Note

The same remarks concerning informal name binding and scope as for the *if-then-else* construct, apply with the constraint of the scope of an identifier defined in some e_i $(1 \leq i \leq n)$ restricted to the corresponding es_i:

```
(...                          ...
 ...
(∃...x...)(P(x))    →    ES(x),
 ...
 ...                          ...)
```

9.2.3 The Cases Conditional

Schema:

```
cases e0:
   (e1  →  es1,
    e2  →  es2,
    ...
    en  →  esn)
```

and:

```
cases e0:
   (e1  →  es1,
    e2  →  es2,
    ...
    T   →  esn)
```

with T as defined above. The form of e_i $(1 \leq i \leq n)$ is either an ordinary expression, i.e. one all of whose free identifiers are bound by/in the surrounding context, or it is one of the so-called defining expression forms:

$\{d_1, d_2, \ldots, d_k\}$
$\langle d_1, d_2, \ldots, d_k \rangle$
$mk\text{-}Nm(d_1, d_2, \ldots, d_k)$
id, cst

where the d_j's $(1 \leq j \leq k)$ are of either of these five forms. id represents identifiers, cst constants.

Programming Note:

Usually the d_j's of an immediate defining expression are identifiers.

Some of these may be bound in the containing scope. If all are bound or constants, then we have an ordinary expression.

Meaning:

We first assume ordinary, i.e. bound expressions, and explicate through reduction to earlier understood forms:

```
(let v0 : e0;
 if v0=e1 then es1
          else
(if v0=e2 then es2
          else
(         ...
          else
(if v0=en then esn
          else error)...)))
```

respectively:

```
(let v0 : e0;
 if v0=e1 then es1
          else
(if v0=e2 then es2
          else
(         ...
          else esn...)))
```

Name Binding & Scope:

We explain the free identifier, defining expression variants by specifically presenting an atypical combination and then transcribing it to an if-then-else form combined with *let* blocks. The example assumes all branch defining expressions identifiers free and rather casually mixes sets, tuples and trees!

```
cases e0:
   ({id1,id2,...,idk}    → es1,
    <id1,id2,...,idk>    → es2,
    mk-Nm(id1,...,ifk)   → es3,
    id                   → es4,
    ...                    ...)
```

e.g. transcribes into:

```
(let v0: e0;
 if is-SET(v0) ∧ card v0=k
    then (let {id1,id2,...,idk} = v0 in
          es1)
    else
(if is-TUPLE(v0) ∧ l v0=k
    then (let <id1,id2,...,idk> = v0 in
          es2)
    else
(if is-Nm(v0)
    then (let mk-Nm(id1,id2,...,idk) = v0 in
          es3)
    else (let id = v0 in
          es4))))
```

This latter form is now annotated (i.e. commented) referring alternatively to the former form:

First the base expression, e_0, is evaluated. The name v_0 identifies the evaluated object. Following the listing given in the cases form we now elaborate, in turn, successive branches until a match is found. Specifically: we first ask whether v_0 is a set and, if so, of cardinality k. If a fit is thus found we dissolve the set v_0 into its k elements naming these id_1, id_2, ..., id_k, whereupon the expression- or statement clause, es_1, is elaborated. This terminates elaboration of the cases clause. If v_0 is not a set it is then asked whether it is a tuple, and if so, of length k. If a fit is thus found..., etc. If v_0 is not a set, tuple, nor a tree of type N_m then v_0 is renamed id and es_4 is elaborated. Thus a free name, as here: id, 'corresponds' to a T-clause -- leading to no elaboration ever of succeeding branches.

The scope of the free identifiers of the branch-expressions is that of the corresponding expression, or statement, clause.

...

And so on: many variations, combinations and permutations (e.g. orderings of free- & bound variables). The above 'schematic' examples have attempted to convey the general idea of defining expressions, their possible mixture of free- & bound variables and even constants. The reader should, from this, be able to extrapolate. The basic point is this: since it is an abstraction, i.e. meta-language and since there is generally not to be an interpreter for it, anything sensible and context-wise obvious is to be allowed.

Example 54:

We shall use the meta-language conditional expressions and recursion to exemplify simpler versions of the conditional statements. That is: we applicatively define imperative constructs!

Abstract Syntax:

The syntactic domains:

$$
\begin{array}{lll}
Cond & = & If \mid MeC \mid Case \\
If & :: & Expr\ \ Stmt\ Stmt \\
MeC & :: & ExSt^{+}\ [Stmt] \\
Case & :: & Expr\ ExSt^{+}\ [Stmt] \\
ExSt & :: & Expr\ Stmt
\end{array}
$$

where *Expr* and *Stmt* are the meta-language expression and statement domains:

The semantic domains; first the textual, then the temporal:

$$
\begin{array}{lll}
\rho \in ENV & = & Id \xrightarrow[m]{} (LOC\ \ldots) \\
\sigma, \xi \in STG & = & LOC \xrightarrow[m]{} OBJ
\end{array}
$$

where *Id* is the syntactic domain of identifiers, *LOC* the further unanalyzed domain of locations.

Semantic Functions:

Let *I* and *V* be names of the generic functions elaborating meta-language statements, respectively expressions, i.e.:

```
type: I:    Stmt ~→ (ENV ~→ (Σ ~→ Σ))
type: V:    Expr ~→ (ENV ~→ (Σ ~→ Σ OBJ))
```

```
I(stmt)(ρ)(σ)=
  cases stmt:
    (mk-If(p,c,a)
     → (let (σ',b) = V(p)(ρ)(σ) in
        if b
          then I(c)(ρ)(σ')
          else I(a)(ρ)(σ')),
     mk-McC(esl,t)
     → (let mcc(les)(ξ)=
              ((lse≠<>)
                 → (let (ξ',b) = V(s-Expr(h lse))(ρ)(ξ) in
                    if b
                      then I(s-Stmt(h lse))(ρ)(ξ')
                      else mcc(t lse)(ξ')),
               (t≠nil)
                 → I(t)(ρ)(ξ),
               T → error) in
        mcc(esl)(σ)),
     mk-Cases(e,esl,t)
     → (let (σ',v) = V(e)(ρ)(σ) in
        let case(lse)(ξ)=
              ((lse≠<>)
                 → (let (ξ',v') = V(s-Expr(h lse))(ρ)(ξ) in
                    if v=v'
                      then I(s-Stmt(h lse))(ρ)(ξ')
                      else case(t lse)(ξ')),
               (t≠nil)
                 → I(t)(ξ),
               T → error) in
        case(esl)(σ')),
     ...)
```

9.2.4 The Ordered, Iterative For-To-Do Statement

Schema:

for id = $expr_m$ to $expr_n$ do stmt

id must be free in the surrounding context. $expr_m$ and $expr_n$ must be integer valued expressions. All occurrences of *id* in *stmt* are free in *stmt*.

Programming Note

We usually constrain $expr_m$ and $expr_n$ to be so-called static expressions. These are expressions whose value can be ascertained statically, without reference to any computation (e.g. state). As such, static expressions are usually either constants (viz.: $expr_m = 1$) or simple operator/operand expressions involving only textual, i.e. syntactic domain objects.

The Controlled Variable

id is called a, or the,controlled variable. It is not a variable in the sense of denoting a *REF*erence, and thus it cannot be changed in *stmt*. *id* denotes an integer.

Name Binding & Scope:

The first occurrence of *id* in "*for id =...*" is the defining occurrence. It binds all (free) occurrences of *id* in *stmt*. The scope of this controlled variable is the for statement in which it is defined.

Meaning:

We first explicate the primarily intended usages of the for statement: namely the ones which have $expr_m$ and $expr_n$ being static expressions evaluating to *m*, respectively *n*. Let *S(id)* be another way of alluding to *stmt*. Then:

for id = m to n do S(id)

≡

S(m);S(m+1);...;S(n-1);S(n)

Thus if $m<n$ then: "*for* $id = m$ *to* n *do* $S(id) =$ "*I*". Given $S(id)$, $S(k)$ means the substitution of all id in S by k.

We then explain the meaning of for statement whose from and to expressions, $expr_m$ & $expr_n$ are expressions of any kind. To this end we apply the meta-language itself!

Example 55:

Let the for statement syntactic domain specification be:

For :: Id Expr Expr Stmt

where $Id \subset TOKEN$, and *Expr & Stmt* denote the class of respectively source-language expressions and statements.

Let the semantic domains pertaining to the interpretation of *For* Statements be the temporal store:

$$\sigma \in STG = LOC \xrightarrow[m]{} OBJ$$

which maps *LOC*ations to *OBJ*ects, and the simplified textual environment:

$$\rho \in ENV = Id \xrightarrow[m]{} (LOC|INTG|...)$$

which maps source-language text *Id*entifiers into either *LOC*ations, if they are names of declared variables, or to *INTeG*ers, if they are e.g. controlled variable names, or... .

Let *V* denote the generic expression evaluation function which applies to source -language expressions, environments & states/stores and yields state changes & objects -- in this case *INTeG*ers. *I* denotes the generic statement interpretation which applies to source-language statements, environments & states/stores and yields state changes.

```
    I(mk-For(id,fe,te,s))(ρ)(σ)=
.1      (let (σ',f) = V(fe)(ρ)(σ),
.2           (σ",t) = V(te)(ρ)(σ) in
.3       let σ"'∈{σ',σ"} in
.4       let for (ρ')(ξ)=
.5               if ρ'(id)>t
.6                  then ξ
.7                  else (let ξ' = I(s)(ρ')(ξ) in
.8                            for(ρ'+[id→(ρ'(id))+1])(ξ'));
.9       for(ρU[id→i])(σ"' ))
```

Annotations:

.1-.2 Both specification expressions, the *f*rom *e*xpression and the *t*o *e*xpression, are e*V*aluated in parallel "," (or in any order), and both may lead to implicit state changes (σ', respectively σ") and to the explicitly desired from, *f*, and to, *t*, values.

.3 This line illustrates the problem: which new (or next) state, σ' or σ", to choose. In this case the problem has been brought upon us by our insistance on parallel evaluation of all specification expressions. Had our pragmatics' lead us to choose sequentiality of *f*rom- and *t*o *e*xpressions evaluation there would have been no need for a non-deterministic choice.

.4 A local function, *for*, is recursively (.4 vs. .8) defined which if applied:

.5 will test whether the controlled variable value (ρ'(*id*)) has gone beyond the limit (*t*),

.6 if so, the final state is the one current when the last invocation (.8 or .9) of *for* was made, i.e. at the end of last iteration, respectively when *for* was first called (initialized);

.7 otherwise the value of the controlled variable is still within bounds, and the 'body', *s*, of the for statement shall be Interpreted, leading to a new state, ξ'.

.8 With this state, and the 'updated' environment which binds the (textual) name of the controlled variable to its incremented value the *for* loop is invoked (recursively).

This completes the definition of the auxiliary *for* function,

.9 which is first activated with the 'initial' state, σ'", and an environment which binds the controlled variable to its *i*nitial value, *i*.

10 Note that the environments are simply extended versions of the environment in which the for statement is interpreted.

In summary: the meta-language for loop was conceived of as a means for iterating through syntactic tuple structures, *tl*. Hence the standard requirement that $expr_m$ and $expr_n$ be static expressions, the former usually *1*, the latter usually $\underline{l}\ tl$.

Further Examples 56 & 57:

We now illustrate imperative and applicative definitions of the *Compound Statement* source language construct:

*Cmpd :: Stmt**

The *i*mperative version is basically based on a state:

$\Sigma_i = (\underline{STG} \rightarrow STG)\ \underline{\cup}\ \ldots$

Int-Cmpd(mk-Cmpd(stl))(ρ)=
  for i=1 to $\underline{l}$ *stl do int-Stmt(stl[i])(ρ)*
type: Cmpd $\tilde{\rightarrow}$ $(ENV_i \tilde{\rightarrow} (\Sigma_i \tilde{\rightarrow} \Sigma_i))$

...

The *a*pplicative version is similarly based on a state:

Σ_a *:: STG ...*

I-Cmpd(mk-Cmpd(stl))(ρ)(σ)=
  if stl=<>
    then
    else (let σ' = I-Stmt($\underline{h}$ *stl)(ρ)(σ) in*
      I-Cmpd(mk-Cmpd($\underline{t}$ *stl))(ρ)(σ'))*
  type: Cmpd $\tilde{\rightarrow}$ $(ENV_a \tilde{\rightarrow} (\Sigma_a \tilde{\rightarrow} \Sigma_a))$

where correspondingly:

type: *int-Stmt:* $Stmt \xrightarrow{\sim} (ENV_i \xrightarrow{\sim} (\Sigma_i \xrightarrow{\sim} \Sigma_i))$

type: *I-Stmt:* $Stmt \xrightarrow{\sim} (ENV_a \xrightarrow{\sim} (\Sigma_a \xrightarrow{\sim} \Sigma_a))$

9.2.5 The unordered For-All & Parallel Statements

The For-All Statement

Schema:

for all def ∈ set do stmt

where def and stmt pairwise are of the forms:

$$\left.\begin{array}{l} id \\ \{cd_1, cd_2, \ldots, cd_n\} \\ \langle cd_1, cd_2, \ldots, cd_n \rangle \\ mk\text{-}Nm(cd_1, \ldots, cd_n) \end{array}\right| \begin{array}{l} \sim S(id) \\ \\ \sim S(id_1, id_2, \ldots, id_m) \end{array}$$

where a proper subset of cd_1, cd_2, ..., cd_n may be constants, otherwise of either of the above four listed forms, usually identifiers, *id*! These must all be distinct. And where id_1, id_2, ..., id_m are the identifiers of *def* free in the containing context. All such must be free in S. It must be statically decidable that *set* denotes a set. And if either of the latter three *def* forms are used, then it must be statically decidable that *set* is of a logical type, *Y*, such that either some *X-set*, X^* or *Nm* is contained in *Y*; viz.: *X-set* $\subseteq$ *Y*, $X^* \subseteq Y$ or: *Nm* $\subseteq$ *Y*!

The Controlled Variables

The identifiers of *def* free in the containing context are the controlled variables. Again, as in section 9.2.4 they are not assignable but directly denotes non-*REF*erence *OBJ*ects.

Name Binding & Scope

The identifier occurrences in *def* free in the containing context are defining occurrences. They bind all their (free) occurrences in *Stmt (S)*. The scope of these controlled variables is the for statement in which they are defined.

Meaning:

In this section we only explicate the semantics of the for-all statement for the simple case of static *set* expressions and simple identifier *def*'s -- this is also the most common case. Let *set* denote a set of k objects, and name these $id_1, id_2, \ldots, id_k$. That is $set \sim \{id_1, id_2, \ldots, id_k\}$:

for all $id \in \{id_1, id_2, \ldots, id_k\}$ *do* $S(id)$

$\sim$

$// \{S(id_1), S(id_2), \ldots, S(id_k)\}$

The Parallel Statement

Schema:

$//\{stmt1, stmt2, \ldots, stmtn\}$ $\quad n \geq 2$

Meaning:

Elaboration of the statements *stmt1*, *stmt2*, ..., $stmt_n$ proceeds in parallel, "independently" of each other. Elaboration of this collateral clause terminates as a result of all statements having been elaborated.

Examples 58-59:

We now illustrate the semantics of a source language statement similar to the parallel statement. Instead of elaborating the statements in parallel, they are just executed in any, arbitrary order:

All :: *Stmt-set*

is the syntactic domain specification. The definition is kept applicative

Σ :: ...

```
I-All(mk-All(stmts))(ρ)(σ)=
   if stmts={}
      then σ
      else (let s ∈ stmts in
            let σ' = I-Stmt(s)(ρ)(σ) in
            I-All(mk-All(stmts\{s}))(ρ)(σ'))
type: All ≃→ (ENV ≃→ (Σ ≃→ Σ))
```

...

A for-all source language construct could be given the following definition:

```
ForAll  ::  Id Expr Stmt

Σ       ::  ...

I-ForAll(mk-ForAll(id,e,s))(ρ)(σ)=
  (let (σ',set) = V-Expr(e)(ρ)(σ) in
   let all(coll)(ξ)=
          if coll={}
             then ξ
             else (let obj ∈ coll in
                   let ξ' = I-Stmt(s)(ρ+[id→obj])(ξ) in
                   all(coll\{obj})(ξ'))                    in
   all(set)(σ'))
type: ForAll ≃→ (ENV ≃→ (Σ ≃→ Σ))
```

where it is assumed that the eValuation function, which elaborates *id*, finds the meaning of *id* just by looking up in the environment!

```
type: I-Stmt:  Stmt ≃→ (ENV ≃→ (Σ ≃→ Σ))
type: V-Expr:  Expr ≃→ (ENV ≃→ (Σ ≃→ (Σ OBJ)))
```

9.2.6 The While-Do Statement

Schema:

```
while expr do stmt
```

where it is statically decidable that *expr* is a truth-valued expression.

Programming Note:

expr is usually an impure expression, i.e. one whose value depends on the current state. The state is, of course, potentially being changed by *stmt*..

Meaning:

As you might expect it -- but here is an applicatively expressed formalization of the imperative while construct.

Example 60:

Abstract Syntax:

$Wh :: Expr\ Stmt$

for the syntactic domain. And:

$\sigma \in STG = LOC \xrightarrow[m]{} OBJ$
$\rho \in ENV = Id \xrightarrow[m]{} (LOC|...)$

for the semantic domains.

Semantic Functions:

$$
\begin{array}{l}
\mathcal{I}(mk\text{-}Wh(e,s))(\rho)(\sigma) = \\
\quad (\underline{let}\ wh(\xi) = \\
\qquad (\underline{let}\ (\xi', b) = \mathcal{V}(e)(\rho)(\xi)\ \underline{in} \\
\qquad\ \underline{if}\ b \\
\qquad\quad \underline{then}\ (\underline{let}\ \xi'' = \mathcal{I}(s)(\rho)(\xi')\ \underline{in} \\
\qquad\qquad wh(\xi'')) \\
\qquad\quad \underline{else}\ \xi') \qquad\qquad\qquad\qquad\qquad \underline{in} \\
\quad wh(\sigma))
\end{array}
$$

$\underline{type}: \mathcal{I} : Wh \xrightarrow{\sim} (ENV \xrightarrow{\sim} (\Sigma \xrightarrow{\sim} \Sigma))$

$\underline{type}: wh: \Sigma \xrightarrow{\sim} \Sigma$

Example 61:

Dijkstra's Guarded, Repetitive Construct [Dijkstra 75]:

rg ∈ *RepGuard* :: *(Expr Stmt)-set*

with *STG* and *ENV* as above, has the following 'informal' semantics: Execute *rg* as long as evaluation of at least one expression in the un=ordered set of pairs of *Expression-Statements* yields *true*:

```
   I(mk-RepGuard(ess))(ρ)(σ)=
.1     (let gwh(sès)(ξ) =
.2           if ses={}
.3              then ξ
.4              else (let (e,s) ∈ ses          in
.5                    let (ξ',b) = V(e)(ρ)(ξ) in
.6                    if b
.7                       then (let ξ" = I(s)(ρ)(ξ') in
.8                             gwh(ess)(ξ"))
.9                       else gwh(ses\{(e,s)})(ξ))      in
.10    gwh(ess)(σ))
       type: RepGuard ~→ (ENV ~→ (Σ ~→ Σ))
```

In this definition, evaluation of guards (*e*) may cause side-effects, i.e.: ξ goes to ξ'. Thus repeated evaluations of some (one) guard, in=itially yielding falsity, might eventually produce truth -- had we not passed the initial state (σ, subsequently ξ) onto alternative evalua=tions (.9). Doing so also secures Dijkstra's requirement that if some guard is chosen, then it was *true* in the 'initial' state (σ, subse=quently ξ).

The non-deterministic aspect of this construct is exhibited by the ar=bitrary set element selection of line .4.

The reader may verify that this command is a (sledge-hammer ?) general=ization of the *while do* construct above!

10. BLOCKS

A compound statement:

$(stmt_1; stmt_2; \ldots ; stmt_n)$

is a sequence of two or more statements. The semicolon, ";", acts as the combinator here.

A block:

$(\underline{let}\ id = expr_d\ \underline{in}\ expr_b)$	$(\underline{let}\ id = expr\ \underline{in}\ stmt)$
$(\underline{let}\ id : expr_d; expr_b)$	$(\underline{let}\ id : expr; stmt)$
$(\underline{dcl}\ vr := expr_d; expr_b)$	$(\underline{dcl}\ vr := expr; stmt)$

consists of a (*let*) definition of a constant, non-reference object, or a (*dcl*) declaration (of a variable) -- on one hand --, compounded with an expression, or a statement (-sequence) -- on the other hand. The com= binators here are either the semicolon, ";", or the "*in*" symbol.

This section will not bring very many examples. Previous sections abound with examples of blocks. We leave it to the reader to look these up for re-confirmation:

Reference to block-examples:

Parenthesized numerals refer to example numbers, the i.j-k trailing these refer to formulae (i) and line number (j-k) sequences.

Applicative expression blocks:

(0)11.3-8, (0)11.6-8, (0)16.1-6, (0)17.3-7, (0)18.3-21, (0)18.8-21, (0)20.3-11, ...

Imperative expression blocks:

(31)14.3-6, (31)14.12-17, (50)2.0-3, (50)3.0-3, (52)4.3-7, (52)5.1-4, ...

Statement blocks:

(31)13.1-8, (52)1.3-8, (52)2.1-2, (52)3.1-3, ...

The Block Concept

The block concept of the meta-language essentially deals with the notion of scope. Blocks are of the forms:

(let id = expr in
clause)

and:

(let id : expr;
clause).

We have, however, in past examples, also permitted ourselves to introduce local variables:

(dcl id := expr type D;
clause).

In all three cases *id* stands for a locally, i.e. a block, defined, constant quantity. We say that the scope of *id* is the block: *(...)* in which it is defined. Only in this textual part, may it be referred to. In the first two blocks *id* denotes the value of *expr* upon block entry. In the last case *id* denotes the location of a meta-storage cell, i.e. an object in the domain of Σ:

$$\Sigma = \ldots \underset{\sim}{\cup} (Id \xrightarrow[m]{} D)$$

In particular:

$$\sigma = \ldots \cup [\underset{\sim\sim}{id} \rightarrow obj]$$

where $\underset{\sim\sim}{id}$ is the denotation of *id*, and *obj* is the value of *expr* upon block entry.

The *let* (or *dcl*) definitions of *id* are said to bind any free occurrences of *id* in *clause*. Thus the scope of *id* extends to inner, nested blocks in which *id* is not redefined.

clause may be a meta-language expression or a sequence of one or more statements.

10.1 Let Constructs

Observe the two distinct cases of *let* constructs:

The syntactic, or applicative *let* construct:

```
(let id = expr in
 clause)
```

If *expr* is a pure expression, i.e. an expression whose evaluation does not require access to any state components, then we use "=" and "*in*" as delimiters.

The semantic, or imperative *let* construct:

```
(let id : expr;
 clause)
```

If, on the other hand, evaluation of *expr* requires access to the state, i.e. if *expr* is an imperative, or impure, expression , then we use ":" and ";" as delimiters.

Assume *id* to occur free in *clause*, i.e. think of *clause* as *C(id)*. Then the meaning of the syntactic *let* clause block is *C(expr)*, i.e. can be obtained by substituting *expr* for all free occurrences of *id* in *clause*. Note that the substitution could be done on the syntactic level, i.e. by replacing texts. The meaning, however, of the semantic *let* clause block is *C(val)*, where *val* is the value of *expr* upon block entry. That is: the meaning can be obtained by substituting values of *expr* for all free occurrences of *id* in *clause*.

Let Construct Variants

A very useful *let* variant is:

```
(let obj ∈ D be s.t. P(obj);
 clause)
```

which we understand as: let *obj* be an object in the domain or set *D* such that it satisfies the predicate P. Other variants, which are simple cases of the above are:

(let obj ∈ Set;
clause)

and:

(let obj be s.t. P(obj);
clause)

The above three variants were indicated, by the ";" separating the *let* construct from the block body *clause*, to be of the semantic type. Similar, but syntactic constructs would instead of the infix ";" use an infix "*in*".

Composite Object Let Decomposition

The following are useful, so-called composite *let* constructs:

(let mk-D(d_1,d_2,...,d_n) = $expr_d$ in
clause)

(let <d_1,d_2,...,d_n> = $expr_t$ in
clause)

and:

(let {d_1,d_2,...,d_n} = $expr_s$ in
clause)

Their respective meanings are:

$expr_d$ must evaluate to a *D* tree object, with:

$$D :: D_1\ D_2\ \ldots\ D_n$$

then the above is equivalent to:

```
(let tree = expr_d in
 (let d_1  = s-D_1(tree),
      d_2  = s-D_2(tree),
      ..
      d_n  = s-D_n(tree) in
  clause)).
```

$expr_t$ must evaluate to an n-tuple:

```
(let tuple = expr_t in
 (let d_1  = tuple[1],
      d_2  = tuple[2],
      ..
      d_n  = tuple[n] in
  clause)).
```

$expr_s$ must evaluate to a set of cardinality n. The 'assignment' of set objects to d_i is arbitrary:

```
(let set = expr_s in
 (let d_1 ∈ set in
  (let d_2 ∈ set\{d_1} in
   (let d_3 ∈ set\{d_1,d_2} in
    ...
    (let d_n ∈ set\{d_1,d_2,...,d_n-1} in
     clause)...))))
```

Simultaneous & Recursive Let Definitions

The form:

```
(let d_1 = e_1,
     d_2 = e_2,
     ...
     d_n = e_n in                    (n≥1)
 clause)
```

simultaneously defines the objects $d_1, d_2, \ldots, d_n$. They are, in general, to be the ("smallest" such) objects which satisfy the (above set of simultaneous) equations. This description permits d_j's to occur recursively in the equation set. The d_j's might be composite constructs,

introduced in the previous subsection, in which case we mean that the constituent (free) identifiers (ultimately occurring in d_j) may occur recursively in the equation set.

For the story on recursive definitions of functions we refer to section 6.2.

Notational Conventions

The form:

```
(let d1 = e1 in
 let d2 = e2 in
 ...
 let dn = en in
 clause)
```

is a short-hand for:

```
(let d1 = e1 in
 (let d2 = e2 in
  (...
   (let dn = en in
    clause)...)))
```

The form :

```
(let mk-A( ,c, ) = ea in
 clause)
```

where e.g.:

```
A :: B C D,
```

is a short-hand for:

```
(let mk-A(b,c,d) = ea in
 clause)
```

It is used to alert the reader to the non-use of the *B* and *D* components of the *A* tree denoted by e_a.

Similarly for anonymous trees; and:

$$(\underline{let}\ \langle x, , z, \rangle = e_t\ \underline{in}$$
$$clause)$$

which is the same as:

$$(\underline{let}\ t = e_t\ \underline{in}$$
$$\underline{let}\ x = t[1],$$
$$\quad z = t[3]\ \underline{in}$$
$$clause).$$

10.2 Pure & Impure Expressions

A pure (or applicative) expression is one whose evaluation never requires access to the state.

An impure (or imperative) expression is one whose evaluation potentially requires access to the state.

Given that Σ_i denotes the state space of an imperative (global variable only) model we can say that the type of the denotation of an impure expression is either of the form:

$$\Sigma_i \xrightarrow{\sim} (\Sigma_i\ OBJ)$$

or of the form:

$$\Sigma_i \xrightarrow{\sim} OBJ$$

The former type hints that evaluation of the meta-language expression potentially leads to a (side-effect) state-change, whereas the latter type only expresses that the state is accessed in order to compose the resulting *OBJ*ect value.

...

The choice of forming an expression either as a pure-, or as an impure, expression, is solely determined by the kind of object to be denoted. That is: even though some abstract model is primarily centered around global state variables, some objects may still be denotable by pure expressions.

If at least one target expressions of a conditional expression is impure, then all such target expressions are to be impure. This requirement is further motivated in [Jones 1978a]. To render an otherwise pure expression impure prefix it with the operator *return*. See section 10.6.

10.3 The ";" Combinator

The ";" is a combinator. Its use can be explained in two ways: syntactically, and semantically. Syntactically speaking, ";" separates imperative clauses: statements; semantic *let* clauses; a semantic *let* clause from a statement; and a statement from an impure expression:

$$(\ldots\ stmt_1;\ stmt_2\ \ldots)$$

$$(\ldots\ \underline{let}\ x_1 : expr_1;\ \underline{let}\ x_2 : expr_2;\ \ldots)$$

$$(\ldots\ \underline{let}\ x : expr;\ iexpr)$$

$$(\ldots\ stmt;\ iexpr)$$

Semantically speaking ";" denotes functional composition. Since the type of the denotation of a meta-language semantic *let* clause, let, or statement, stmt, is:

$$\text{let, stmt}: \Sigma_i \xrightarrow{\sim} \Sigma_i$$

The construct:

$$(c1;c2)$$

where $(c1,c2)$ are either (semantic lets, statements) or (statements, statements), means:

$$\lambda\sigma.(c2(c1(\sigma)))$$

where ci is the meaning of ci (i=1,2). Thus:

$$; \sim \lambda c1.\lambda c2.\lambda\sigma.(c2(c1(\sigma)))$$

i.e. $$; \in (\Sigma \tilde{\to} \Sigma) \to ((\Sigma \tilde{\to} \Sigma) \to (\Sigma \tilde{\to} \Sigma))$$

10.4 Compound Statements & Statement Sequences

The meta-language permits any statement to be a compound statement:

$$(stmt_1; stmt_2; \ldots ; stmt_n)$$

The body of a (statement-) block, i.e. the syntactic construct referred to as *clause* in e.g. section 10.1, may be a statement sequence:

$$stmt_1; stmt_2; \ldots; stmt_n$$

There is no semantic difference between these two constructs. We omit parenthesizing the latter since it is always superfluous.

The formal meaning is:

$$\lambda\sigma.\delta stmt_n(\delta stmt_{n-1}(\ldots(\delta stmt_2(\delta stmt_1(\sigma)))\ldots))$$

where $\delta stmt_i$ is the meaning of $stmt_i$.

We refer to [Jones 1978a] for a further λ-definition of the meanings of the basic statements.

The informal meaning is as you would expect it to be.

10.5 Statement- & Expression Blocks

A statement-block is a statement. The block, when interpreted, effects a state-change. An expression-block is an expression. The block, when evaluated, may effect a state-change, but always, in addition, delivers a value:

stmt-block: $\Sigma_i \tilde{\to} \Sigma_i$

expr-block: $\Sigma_i \tilde{\to} (\Sigma_i\ OBJ)$

A statement block consists of a sequence of one or more syntactic and/or

semantic *let* clauses followed by a sequence of one or more statements.

An expression block is either a pure- or an impure- (i.e. an applicative-, respectively an imperative-) expression. A pure expression block consists of one or more syntactic *let* clauses followed by a pure expression. An impure expression block consists of a non-zero length sequence of zero, one or more syntactic- or semantic *let* clauses followed either by an impure expression or a statement all of whose syntactically possible execution paths end with an impure expression. Since only well-structured statements are permitted the test for this latter is quite simple.

10.6 Return

From the explanation of ";" it follows that if a statement sequence, of an expression block, is to be followed by an expression, then the type of this expression must be:

$$\Sigma \rightarrow (\Sigma\ OBJ)$$

i.e. impure.

In general, if the context determines that an expression be impure, and the value to be yielded can be denoted by a pure expression, e, then we need to render e impure. This is the purpose of the monadic expression operator *return*.

$$\textit{return} \sim \lambda obj.\lambda\sigma.(\sigma, obj)$$

$$\textit{return} \in (OBJ \rightarrow (\Sigma \rightarrow (\Sigma\ OBJ)))$$

Examples

See examples: (31) 14.6, (31) 14.17, (50) 2.3, (50) 3.3, (50) 4.5.

11. *EXIT*s

Even though the meta-language has imperative constructs, it lacks the conventional GOTO construct. Hence it lacks labels.

The exit mechanisms, in many ways, replace the GOTO construct. You may say, without being grossly wrong, that *exit* provides a structured GOTO, albeit, in general, to a dynamically determined, and -- in any case -- unlabelled, program point.

Example 62:

We use the exit mechanism when modeling the GOTO concept of our running example source language! Therefore: in the following read carefully. Observe the distinction between *source*- and meta-language constructs, in particular the block constructs.

```
1    Block       ::  (Id -m-> Type) ... Named-Stmt*
2    Named-Stmt  ::  [Lbl] Stmt
3    Stmt        =    Block | Goto | ...
```

In this source language *Goto*s may not go into phrase structures. That is: *Goto*s may e.g. go from one statement of a *Named-Statement* list to another statement of the same list, or out of the containing *Block* to a statement of the *Named-Statement* list of the next embracing *Block*, in fact it may go out of any such number of nested levels. Each time a *Block* activation is left, whether through (normal) epilogue due to all statements having been executed, or whether due to a global *Goto*, the same 'clean-up' epilogue actions -- as were illustrated in e.g. example 31, section 4.5 -- must first take place:

```
4 int-Block(mk-Block(dcls,procs,nstl))(ρ,ca)=
  .1 (let aid ∈ AID be s.t. aid ~∈ ca in
  .2  let ρ' : ρ+[ lbl → mk-LAB(aid,lbl) | lbl ∈ Labels(nstl) ]
  .3             +[ v → get-loc(dcls(v))(ρ) | v ∈ dom dcls ]
  .4             +[ s-ID(p) → eval-Proc(p)(ρ') | p ∈ procs ];
  .5  always
  .6      (let locs  = {ρ'(v) | v ∈ dom dcls}        in
  .7       let slocs = { ... see example 31 ... }   in
  .8       STG := c STG\slocs)
  .9  in int-Named-Stmt-list(nstl)(ρ,ca∪{aid},aid))

      type:  Block ~→ ((ENV AID) ~→ (Σ ~→ Σ))
```

.4 *eval-Proc* is the imperative version of example 40's *V-Prc*, as used in example 42 (section 6, respectively 6.3).

.2 The denotation of a label is in the domain:

LAB :: AID Lbl

The *AID* component serves to keep track of active source language block activations.

.1 The prologue action of a *Block* selects an *a*ctivation *id*entifier not in the set of (identifiers selected by all) *c*urrent *a*ctivations.

.2 The block environment associates all *Labels* of its *n*amed *s*tatement *l*ist with this (unique) activation (identifier).

.4 Since procedures may be recursive, and -- independently thereof -- since *goto*s may occur from within a procedure (activation) to the *n*amed *s*tatement *l*ists of any of its embracing, including defining, blocks, the environment passed to *eval-Proc* is the environment being (thus recursively) defined.

We shall return to lines .5-,9, which -- on the whole -- resemble lines .4-.8 & .3 of *int-Block* of example 31, section 4.5.

```
5 int-Named-Stmt-list(nstl)(ρ,ca,aid)=
  .1 (tixe [ mk-LAB(aid,l) → cue-int-Named-Stmt-list(i,nstl)(ρ,ca)
  .2                       | 1 ≤ i ≤ len nstl ∧ s-Lbl(nstl[i] = l ≠ nil ]
  .3  in cue-int-Named-Stmt-list(1,nstl)(ρ,ca))

6 cue-int-Named-Stmt-list(i,nstl)(ρ,ca))=
  .1  for j=i to len nstl  do int-Stmt(s-Stmt(nstl[j]))(ρ,ca)

7 int-Stmt(stmt)(ρ,ca)=
  .1  cases stmt:
  .2     (mk-Goto(lbl)  → exit(ρ(lbl)),
  .3      mk-Block(...) → int-Block(stmt)(ρ,ca),
  ..      ...)
```

-- annotations:

5.0 *Int*erpreting a *n*amed *s*tatement *l*ist is the same as:

5.3 *int*erpreting that list as from its *1*st statement.

That is: the *tixe* (*exit* spelled in Polish) clause of lines 5.1-5.2 is not 'executed' by the meta-language elaborator when first entering the *int-Named-Stmt-list* functions. We shall subsequently 'discover' the purpose of the *tixe* clause.

6.0 Having *cue*d this function with *i*, i.e. having given it the start-word *i*, *int*erpretation of the *statement*s of the (*n*amed) *statement list*

6.1 proceeds linearly, as from the *cue*d, *i*th, statement until the last -- provided, of course, meta-elaboration is not re-directed.

When & how this occurs will presently be uncovered.

7.1 If the *int*erpreted *statement* is

7.2 a *Goto*, then an *exit* is performed. It is given an argument. This argument is the *LAB*el denotation, *mk-LAB(aid,lbl)*, by which this label is known in the environment.
Meta-elaboration of an *exit* dynamically 'unravels' the meta-language `block` invocations. That is: we retrace our steps, back to (here) the most recent meta-language `block` having a *tixe* clause. We say that the *tixe* clause stops the *exit*.

5.1 The *tixe* clause we backtrack to, is the one associated with the *n*amed *statement list* of which the 'offending' *Goto* was an immediate statement.

The *tixe* clause is to be understood as follows: if the argument passed back with the *exit* (7.2) is equal to some *mk-LAB(aid,l)* of the "`map`" component of the *tixe* clause, then the corresponding *cue-int-Named-Stmt-list*, with the appropriate *cue* position (*i*), is invoked; otherwise the despatching *exit* is not stopped here, but passed out to the next embracing *tixe* clause -- i.e. the *Goto* is to an embracing *Block*.

6-7 In now (re-)*int*erpreting the (same) *n*amed *statement list*, but, possibly, as from another, *cue*d, position, new *Goto*s may occur.

5.1 These are stopped, thus recursively, by the 'same' *tixe* clause.

7.1 If the *interpreted statement* is instead, in contrast to annotations 7.1-7.3 above,

7.3 a *Block*, then that, nested, *Block* is *interpreted*.

And so on. *exit*s are always stopped by an *always* clause before being passed on (4.5-4.8).

Comments

The "maps" of *tixe* clauses may, as we have done in the above annotations, be considered not to be "computed" when the meta-language elaborator enters a meta-language `block` having a *tixe* clause. The domains of such "maps" are constrained to be statically determinable, and finite. Since, in the *cue-int-...* elaboration function, *nstl* is a static (i.e. fixed) object, this constraint is satisfied.

You may in fact view the domain of the "map" of the *tixe* clause as being 'computed' upon entry to the meta-language `block` of which it is part. Now, for this source language, the test for whether the "returned" argument of an *exit* belongs to such a "map" is likewise 'statically' decidable since only *Goto*s with constant *Label* designators were permitted.

11.1 The Exit Mechanisms

In the following the language constructs, dealt with and mentioned, are those of the meta-language, and not constructs of a *source* language being modeled.

Variables presuppose declarations, and declarations define a state. Statements are requests for state changes, and denote state transformers, i.e. functions from states to states. The serial statement composition operator ";" (semicolon) thus denotes functional composition .

In compound expressions all component sub-expressions are (usually) completely evaluated before any result value is yielded.

In this section we shall describe the only statement- and expression construct available for changing the meta-language statement interpre-

tation-, respectively controlling the expression evaluation-, part.

The *exit* mechanism now to be explained offers -- in the context of statements -- in one sense a restricted form of, or alternative to conventional *goto*s. In particular: instead of providing, in the meta-language, *goto*s to (arbitrary or phrase-structure constrained) labelled statements *exit*s provide such *goto*s to statement-block or block-expression "*end*s".

In the context of an applicative language it is an altogether new construct.

The meta-language provides two kinds of *exit* designators:

```
1,2        exit,   exit(expr)
```

and three kinds of *exit* stopping constructs:

```
3.0        (always F(...)
 .1         in C(...))

4.0        (trap exit(def) with F(def)
 .1         in C(...))

5.0        (tixe [ G(def) → F(def) | P(def) ]
 .1         in C(...))
```

Lines 3.0, 4.0 and 5.0 schematize the stopping clauses. Any clause, *C(...)*, prefixed by a stopping construct becomes a block.

Clauses 1 or 2 can occur where meta-language statements or expressions may occur. Thus the *exit* mechanism is an imperative, as well as an applicative construct -- see [Bjørner 77e]. The use of the *always* stopping clause, as will be seen, is, however, restricted to imperative blocks.

The forms *def* can be of either of the forms:

$$\{d_1,d_2,\ldots,d_n\}$$
$$<d_1,d_2,\ldots,d_n>$$
$$mk\text{-}D(d_1,d_2,\ldots,d_n),\quad (d_1,d_2,\ldots,d_n),$$
$$id, \qquad \text{or}$$
$$cst$$

where *id* stand for (free or bound) *id*entifiers, *cst* for *cons*tants (literals), d_i for forms of the above kind, and where the *def* form need not contain any free identifiers.

11.2 Pragmatics & Semantics of the Exit Mechanism

The *exit* concept is based on the following four principles:

(I) The first basic principle of *exit* is to permit *goto*-like transfer of elaboration (statement interpretation, respectively expression evaluation) control to block boundaries -- i.e. to just outside their terminating part.

In particular: elaboration of any *exit* not definitively stopped in a block properly contained in *C(...)* will result in immediate termination of any further parts of *C(...)* , this to be followed by elaboration of some *F(...)*.

(II) The second basic principle of *exit* is to permit the user to (implicitly) specify which (dynamically enclosing) block end the *exit*s go to!

A block with no stopping clause is said to not definitively stop any *exit*. A *trap exit* unit whose elaboration may result in an *exit* is likewise said to not definitively stop an arbitrary *exit*. Finally: if an elaboration of *F(...)* is completed with no *exit* then the *exit* is said to be definitively trapped. In that case elaboration of the block consists of elaboration of the part of *C(...)* up to the *exit* followed by elaboration of the *F(...)*. The potential state transformation yielded by an imperative block is in this case the serial composition of the two net effects. For the case the block is a block-expression *F(...)* must in this case, namely that of definitive en*trap*ment, yield a value.

(III) The third basic principle of *exit* is to permit the meta-language programmer to specify that certain actions be taken at any stopping block end.

The stop clauses serves this purpose. *exit*s from multiply nested blocks may cause successive stopping actions, one per block (inside-out), and each being terminated by an *exit* to the next enclosing block.

An *exit* not definitively stopped by a stop clause of the (outermost) block of a function definition is dynamically passed to the immediately embracing block in which the actual reference to the function, i.e. "to this function definition", occurred.

The *always* stop clause unconditionally filters any *exit*, its *F(...)* is elaborated, and the *exit* passed on to outer blocks.

In consequence: *exit*s of one function definition may, depending on dynamic calling patterns, be trapped by a multitude of stop clauses of blocks contained in various (other) function definitions. An *exit* of an activation of a recursively defined function may thus be stopped by a prior, temporarily suspended activation of that "same" function.

(IV) The fourth basic principle of the *exit* mechanism, i.e. the joint use of *exit*s and stopping clauses is finally to communicate information from the (usually only dynamically determinable) point of *exit* to *trap exit* and *tixe* stop clauses' *F(...)*. The idea being to let the elaboration of *F(...)* depend on *exit* "returned" data.

In particular: the value, *v*, of the *expr* of *exit(expr)* is obtained; the proper stop clause is found; and *v* is substituted for all free occurrences of that unit's formal parameter *def* in that unit's *F(...)* resulting in *F'(...)*. Then *F'(...)* is elaborated.

Scope Rules

An alternative way of describing some of the linguistic properties (of the exit mechanism) is now presented.

Two scope aspects are important: A syntactic (or static), and a semantic (or dynamic).

The syntactic scope rule is concerned with the scope of identifiers occurring in the *def* form of the *trap exit* clause. Identifiers of *def*, free in the embracing context, bind free occurrences of these identifiers in the corresponding *F(def)*. The static scope rules of the *tixe* "map" are the same as those of any implicit map (set or tuple) construction.

The semantic scope rule is concerned with the scope of the *always*, *trap exit* and *tixe* clauses.

The dynamic scope rules of the *always* and *trap exit* clauses is the text *C(...)*; whereas the dynamic scope rule of the *tixe* clause includes both the *tixe* "map" and the text *C(...)*!

Thus: an *exit* 'occurring' as a result of elaborating *F(def)*, of 3.0 or 4.0, if not stopped in *F(...)*, is not stopped by this (3.0, respectively 4.0) instance of the *always*, respectively *trap exit* clause. Instead it is passed out to possibly embracing meta-blocks.

An *exit* 'occurring' as a result of elaborating some *F(def)* of 5.0, i.e. of a *tixe* "map", if not trapped in *F(...)*, is trapped by this *tixe* clause (5.0). Thus the *tixe* clause is said to apply recursively!

Some Equivalence Transformations

The *always* stop clause in:

```
(always F(...)
 in C(...))
```

is semantically equivalent to:

```
(trap exit(id) with (F(...); exit(id))
 in C(...))
```

In general a block with no stopping clause:

```
(let x : E(...);
 C(...))
```

is identical to a block with the simple gate:

```
(trap exit(id) with exit(id)
 in (let : E(...);
     C(...))).
```

When no block termination actions are needed in an imperative block, then we write:

```
(trap exit with I
 in C(...))
```

Correspondingly, when the value passed back (by the *exit*) unconditionally is to become the result of the block, we write:

```
(trap exit(id) with id
 in C(...))
```

or:

```
(trap exit(id) with return(id)
 in C(...))
```

dependent on whether the block (-expression) is pure or impure.

Finally: 'mixed', nested uses of *exit(e)* and *exit*, and thus e.g. *(trap exit with V(...) in C(...))* and *(trap exit(id) with F(id) in C(...))* do not make sense:

```
(trap exit(id) with  F1(id)
 in (...)
        (trap exit with F2(...)
         in (...
             exit(expr)
             ...))
    ...))
```

Etcetera.

Example 63:

Continuing our directory example, from examples 19 & 29, we recall:

$$DIR = Rid \xrightarrow[m]{} RES$$
$$RES = VAL \mid DIR$$

with:

```
retrieve-res(dir,ridl)=
   ((ridl = <>) → dir,
    T           → (let rid = hd ridl in
                   if rid ∈ dom dir
                      then retrieve-res(dir(rid),tl ridl)
                      else undefined))
```

type: $DIR\ Rid^* \xrightarrow{\sim} RES$

Users of the system directory each have their qualified name, *uid* in *Rid**, otherwise guaranteed to designate a *DIR* object:

is-DIR(retrieve-res(Dir,uid))

Users issue resource names, *vid* in Rid^+, designating values in the glo= bal directory, *Dir*, as follows: let *dir* be the directory designated by *uid* :

let dir = retrieve-res(Dir,uid)

If *vid* is some <*rid*> then either *rid* names an entry in *dir*, and we are through, the result is *dir(rid)*. Or *rid* does not name an entry in *dir*. We now chop off the last *Rid* element of *uid*-- to resume the search as from the directory designated by the resulting ('remaining') *uid'*. If *vid* is some $<rid_1,rid_2,\ldots,rid_n>$ for $n>1$, then rid_1 either names a *DIR* entry in *dir*, and we search as from the designated directory, with resource name: $<rid_2,\ldots,rid_n>$, or rid_1 does not name a *DIR* entry, and we 'back' up the directory hierarchy, to a level one higher, i.e. nearer the root, than that at which we originally started , or at which the search which just failed took place. We complete the above incomplete description by giving the formal definition of the proper search algo= rithm:

```
     search(uid,vid)(updown)(Dir)=
.1      (let dir = retrieve-res(Dir,uid) in
.2      (trap exit with
.3              if updown = DOWN
.4                 then exit
.5                 else if uid = <>
.6                         then undefined
.7                         else search(fst(uid),vid)(UP)(Dir) in
.8       if len vid = 1
.9          then if hd vid ∈ dom dir
.10                 then dir(hd vid)
.11                 else exit
.12         else if hd vid ~∈ dom dir
.13                 then exit
.14                 else search(uid~<hd vid>,tl vid)(DOWN)(Dir)))
        type: Rid* Rid+ ≅→ ((UP|DOWN) ≅→ (DIR ≅→ RES))
```

where:

$$fst(ridl) = \langle ridl[i] \mid 1 \leq i \leq (\underline{len}\ ridl) - 1 \rangle$$

Given the global directory, *Dir*, which we could omit as a parameter, and a 'relative' resource name, *vid*, we initially invoke the above search function with the users identification and the *updown* marker set to *UP*.

We may tie up any loose ends in your understanding of the search algo= rithm, by the following, alternative characterization: *uid* designates a node, *N*, in the directory hierarchy (or: "tree", see example 19). There are now three possibilities for *vid*. Either there is a downward path, towards the leaves, from *N* whose sequence of edge labels is *vid*. Then *vid* designates the object 'hanging' on to the other end of this path (as seen from *N*). If there is no such path, starting at *N*, then *vid* might still designate an object in the hierarchy, but now as from a node *N'*, between the root of the overall directory and *N*. Search starts with the node *N'* immediately above *N*.

The purpose of the updown marker is to guide the trap mechanism in backing up beyond already searched 'subtrees'.

The third possibility, for *vid*, is that there is no resource, relative to *uid*, that is named by it.

Example 64:

The Zahn Event Mechanism [Zahn 74, Knuth 74, Halås 75, Bjørner 77d] sche= matically looks like:

```
loop stmtl0
     until eid1 do stmtl1,
           eid2 do stmtl2,
           ...
           eidn do stmtln
pool
```

An abstract syntax is:

Zahn :: Stmt$^+$ (Eid $\vec{m}$ Stmt$^+$)

No statement in any $stmtl_i$ $(1 \leq i \leq n)$ may name any event:

event eid

on any eid_j $(1 \leq j \leq n)$. With:

Event :: Eid

as a statement, i.e. with:

Stmt = Event | Zahn | ...

the static well-formedness criteria can be completely specified:

```
is-wf-Stmt(s)(eids)=
.1    cases s:
.2       (mk-Event(eid)      → eid ∈ eids,
.3        mk-Zahn(stl0,esm) → (∀c ∈ elems stl0)
.4                                 is-wf-Stmt(c)(eids ∪ dom esm)
.5                            ∧(∀eid ∈ dom esm)
.6                                 (∀c ∈ elems(esm(eid)))
.7                                    is-wf-Stmt(c)(eids\dom esm),
.8        ...                 → ...)
```

The informal, incomplete semantics is: $stmtl0$ is repeatedly interpreted. If an event is interpreted in $stmtl0$ then interpretation of $stmtl0$ is terminated. If the event names some eid_k $(1 \leq k \leq n)$, then the corresponding $stmtl_k$ is interpreted. This terminates interpretation of the mechanism. If the event names some properly containing mechanisms' eid_ℓ, then its $stmtl_\ell$ is obeyed, ..., etc. Formally, we can capture things much more succinctly:

Assuming:

$\Sigma = (\underset{\sim}{STG} \rightarrow STG) \cup (\ldots\ \ldots)$

we get an imperative formulation:

type: int-Stmt: Stmt $\tilde{\rightarrow}$ *(ENV* $\tilde{\rightarrow}$ *(*Σ $\tilde{\rightarrow}$ Σ*))*

```
int-Stmt(s)(env)=
   cases s:
      (mk-Event(eid)
        →exit(eid),
       mk-Zahn(stl,esm)
        →(tixe  [eid → int-Stmt-list(esm(eid))(env) | eid ∈ dom esm]
          in
          while true do int-Stmt-list(stl)(env)),
       ..→..)

int-Stmt-list(stl)(env)=
   for i=1 to len stl do int-Stmt(stl[i])(env)
```

It is thus we observe that the *tixe* construct closely parallels the Zahn event mechanism. The reader is,however, well-adviced to study [Jones 78a,78b] in order to discover the neatness of the λ-calculus semantics given there for this meta-language combinator.

PART IV ABSTRACT MODELS

Part I exemplified a complete, abstract model. Its components were: (1) abstract syntaxes for both syntactic & semantic domains; (2) [static and dynamic] consistency constraints on objects of these domains -- called *is-wf-...* functions; (3) semantic elaboration- , and (4) auxiliary function definitions.

In part II, sections 2.1, 3.1, 4.1, 5.1, 6.1 and 7 taught the notation for, and techniques of, defining domains. Sections 2.2, 3.2, 4.2, 5.2 and 6.2 taught the notation for representing objects of these domains. Sections 2.3, 3.3, 4.3, 5.3 and 6.3 finally taught the notation for, and techniques of, expressing transformations on objects.

Part III generally taught the notation for, and techniques of, stating processes, respectively composite transformations, on objects.

Part IV finally closes the story on the meta-language. With the bits and pieces of the meta-language introduced formally, and also heavily exemplified in parts II & III, we can now formally introduce the notion of function definitions. Function definitions have, however, already been extensively exemplified. Section 12 will therefore wrap-up our tutorial on the meta-language. The section can best be understood if you frequently compare the formal information and schematized "examples" with examples 0-63! That is: we do not present comprehensive, abstract models involving function definitions, although those are its prime subjects!

12 Function Definitions & Abstract Models

In section 5 we introduced the notation for denoting domains of *FunCTion* objects, for representing *FunCTion* objects, and for operations on *FunCTion* objects. In a sense we are here continuing the story we started giving there.

Example

Recall the abstract models of e.g. examples 0,19,29,42,51-52,54-55,58-59-60,61. On one hand we gave abstract syntaxes for syntactic domains, on the other we gave abstract syntaxes for semantic domains. The meaning of a syntactic object was a semantic object. The meaning of a *Proce*-dure, i.e. an object in *Prc* (example 40), is an object in $ARG^{*}\tilde{\rightarrow}(\Sigma\tilde{\rightarrow}\Sigma)$. The "thing" which takes a *Prc* object and yields its denotation, i.e. an $ARG^{*}\tilde{\rightarrow}(\Sigma\tilde{\rightarrow}\Sigma)$ object, we call a semantic elaboration function *(V-Prc)*. It is a function. This function, like all the *FunCTion* objects of section 5, is described by the meta-language construct we call a function definition.

12.1 The Syntax of Function Definitions

A function definition consists of three parts:

a header: $fid(d_1,d_2,\ldots,d_k)(d_{k+1})\ldots(d_n)=$

a body: $C(\ldots)$

a type clause: *type*: $D_1\ D_2\ \ldots\ D_k \rightarrow (D_{k+1} \rightarrow (\ldots \rightarrow (D_n \rightarrow D')\ldots)$

The body is any statement or expression clause you choose. The d_i for $1\leq i\leq n$ are usually formal parameter identifiers. They, or any identifiers occurring in d_i's, and the function name, *fid*, bind free occurrences of these identifiers in $C(\ldots)$. The D_i's and D are domain expressions.

Observe the following relations between the d_i's of the formal parameter list and the D_i's of the *type* clause.

(1) If d_i is of the form $\langle d_{i1},d_{i2},\ldots,d_{im}\rangle$, the D_i is either an identifier, and is then either the name of a domain defined by some abstract syntax rule:

$D_i = D^*$, $D_i = D^+$ or

$D_i \subset D^*$;

or D_i is (directly) either D^* or D^+ -- for some D.

(2) If d_i is of the form $\{d_{i1}, d_{i2}, \ldots, d_{im}\}$ then:

$D_i = D\text{-}set$

etcetera.

(3) If d_i is of the form $mk\text{-}D_i(d_{i1}, d_{i2}, \ldots, d_{im})$, then there is an abstract syntax rule:

$D_i :: D_{i1}\ D_{i2} \cdots D_{im}$

(4) If d_i is of the form $(d_{i1}, d_{i2}, \ldots, d_{im})$, respectively $mk(d_{i2}, d_{i2}, \ldots, d_{im})$, then D_i is of the form:

$(D_{i1}\ D_{i2} \cdots D_{im})$

We do not presently see a need for writing d_i's in forms other than the above, where ultimately d_{ij}'s end up as identifiers, or letting the d_i's be identifiers. In the latter case D_i's may be any appropriate domain expression, not just an identifier. Experience shows forms (1)-(2) to rarely occur.

Example 65:

This continues example 40. Now, let:

$PROC = ARG^* \xrightarrow{\sim} (\Sigma \xrightarrow{\sim} \Sigma)$

$Prc :: Id^*\ Block$

and:

$ENV = Id \xrightarrow[m]{} DEN$

$DEN = LOC \mid PROC$

With these prerequisites we could also write the *V-Prc* elaboration function definition in this way:

```
V-Prc(p)(ρ)=
   (let mk-Prc(idl,bl) = p  in
    let fct(al)(ξ)=
           (...
            ...)  in
    fct))
```

type: $Prc \xrightarrow{\sim} (ENV \xrightarrow{\sim} PROC)$

or

type: $Prc \xrightarrow{\sim} ((Id \xrightarrow[m]{} DEN) \xrightarrow{\sim} (ARG^* \xrightarrow{\sim} (\Sigma \xrightarrow{\sim} \Sigma)))$

Relating the last *type* clause to the formal parameter list $((p)(\rho))$ we observe that p is of type *Prc*, ρ of type $(Id \xrightarrow[m]{} DEN)$, and the result yielded by *V-Prc* is whatever remains to the right of the arrow after the *ENV* specification!

end-of-Example

The form of D remains to be constrained.

(5) If the body, $C(...)$, of the function definition is a statement (-block), then D is of the form:

$\Sigma \to \Sigma$, or: $\Sigma \xrightarrow{\sim} \Sigma$

where Σ is the state resulting from global variable declarations.

(6) If the body, $C(...)$, is instead an impure expression (-block), then D is of the form:

$\Sigma \xrightarrow{\sim} (\Sigma\ D')$, or: $\Sigma \xrightarrow{\sim} (\Sigma\ D')$

where D' denotes the domain of results yielded by $C(...)$ with Σ as in (5).

(7) If finally $C(...)$ is a pure expression, D is any domain expression (etcetera -- concerning relation to $C(...)$).

Example 66:

Observe that D of e.g. the pure (applicative) *I-Block*, *I-Stmt-list*, and *I-Stmt*, definitions of example 42, were all Σ. But then the Σ was not

that of any global variable state space -- since there was none --, but explicitly defined in example 42 formula 2!

end-of-example

Schönfinckeling/Currying

The "back-to-back" ("dos-á-dos") parentheses, "*)(*", of the formal parameter list, e.g. to the left of d_i, may be replaced, starting with $i=k+1$, by commas "," -- provided we simultaneoursly delete the "→ (" in the type clause, to the left of the corresponding D_i, and delete a (matching) right parentheses ("*)*").

Formal Examples

$f(a)(b)(c)(d) = C(...)$ *type*: $A \to (B \to (C \to (D \to E)))$

is one way,

$f(a,b)(c)(d) = C(...)$ *type*: $A\ B \to (C \to (D \to E))$

is another way,

$f(a,b,c)(d) = C(...)$ *type*: $A\ B\ C \to (D \to E)$

and:

$f(a,b,c,d) = C(...)$ *type*: $A\ B\ C\ D \to E$

is a final way of following the so-called de-Schönfinckeling or de-Currying rule given above.

The form:

$f'(a)(b,c)(d) = C(...)$ *type* $A \to ((B\ C) \to (D \to E))$

is, however, not a currying of f's definition above. De-Currying f' would lead to:

$f'(a,(b,c),d) = C(...)$ *type* $A\ (B\ C)\ D \to E$

The conclusion to be drawn from the last example is that the type clause:

type: g': $(D_1\ D_2\ \ldots\ D_n) \to D$

is altogether distinct from:

type: g'': $D_1\ D_2\ \ldots\ D_n \to D$

Although we might (sloppily) write:

$$g'(d_1,d_2,\ldots,d_n)$$

where, more correctly, we should write:

$$g'((d_1,d_2,\ldots,d_n))$$

to distinguish it from the (correct):

$$g''(d_1,d_2,\ldots,d_n)$$

end-of-formal-examples

The rule for removing "$)($"s etc. can be used in reverse -- and this was covered in section 6.2 (see subsection on λ-Expressions). In 6.2 we "Curry", whereas here we "de-Curry". As implied by the last, formal, example above, we must observe the dual role of parentheses: delimiting and anonymous tree domain & tree object construction! If your keyboard has an extra pair of bracket/brace/parenthesis delimiters (other than our {}, <>, (), [], then use them to distinguish!

The Currying/de-Currying Rule is one form of syntactic transformation on function definitions. It involved the header/*type*-clause 'pair'. Semantics remained invariant!

Formal Parameter Syntactic Transformations

Another kind of syntactic transformation rule involves the header/body 'pair'. It can most simply be dealt with by listing the following semantic identities:

$$f(\langle d_{i1},d_{i2},\ldots,d_{im}\rangle) = C(\ldots)$$

(1) ≡

$$f(d) = (\underline{let}\ \langle d_{i1},d_{i2},\ldots,d_{im}\rangle = d\ \underline{in}\ C(\ldots))$$

$$f(mk\text{-}D_i(d_{i1},d_{i2},\ldots,d_{im})) = C(\ldots)$$

(2) ≡

$$f(d) = (\underline{let}\ mk\text{-}D_i(d_{i1},d_{i2},\ldots,d_{im}) = d\ \underline{in}\ C(\ldots))$$

etc.. It is here assumed that d is not free in $C(\ldots)$.

12.2 The Semantics of Function Definitions & Abstract Models

Scope & Binding

The identifiers occurring in the header of a function definition bind all their (free) occurrences in the body! Usually there is no need to re-define identifiers in any block.

General

An abstract model consists of a set of function definitions, a set of abstract syntaxes, and possibly also a set of variable declarations. All names of defined functions, domains and variables are globally known.

The function denoted by any such, global function definition headers' first identifier, i.e. by the function identifier, *fid*, is the one you would expect. Formally: it is the least fix-point solution to the equation set that the function definitions form.

Informally we observe that an abstract model is all definitions -- and no real "action"! Nobody really invokes any of the specified functions.

An abstract model defines a class of systems.

Examples

Example 0 defined the structure and possible behavioral patterns of a class of grocery stores.

Examples 3-8, 12-13, 17-18, 20-25, 27-28 & 35 defined fragments of classes and manipulations of file systems.

Examples 19,29&62 defined the structure and operations on a class of operating system directories.

Examples 30-34, 36-38,40,42-43,47,49,61,63,70 defined fragments of structures and meaning of some class of programs, i.e. a source language.

If you come with a particular grocery store, or a particular file system, or a particular directory and a corresponding action (e.g. *pur-*

chase, *read*, respectively *catalog*), then the definitions, i.e. the models, tell you what to expect happening. If you come with a particular state configuration and a program then the model will compute you a result.

However, you need not come with a semantic and a syntactic object. Come just with a syntactic object. And the model will produce the answer: this syntactic object denotes such-and-such a function from semantic objects to semantic objects.

If you are asking for the meaning of say a *Purchase*, example 0, then "stick" your *mk-Purchase(...)* object into the *Elab-Purchase* semantic function. That is: you decide which part of your model to concentrate on. Therefore we, the model builders, do not tell you which one semantic function is the most important, i.e. which one should always be invoked first when "starting up" a model!

end-of-example

A Note on Input/Output

Based on the discussion (of example 69) it should now be easy to understand why the meta-language has no *input/output* constructs.

A program in the meta-language defines a function. The meaning function from syntactic objects to semantic objects. The prime means for expressing this function is the set of function definitions. The abstract syntaxes describe the domain and range of this meaning function: the class of syntactic objects, and the class of semantic objects.

PART V: MISCELLANEAE

There remains only to formally introduce the elementary domains used in examples throughout this tutorial.

13 *ELEM*entary Data Types

Usually a programming language comes ready-made, equipped with a set of basic data types.

A number of such have been employed in the examples given so far. These are:

.1 Rational *NUM*bers, *INT*e*G*ers (...,-2,-1,0,1,2,...), *N*atural numbers -- with the operations: +, -, *, /, <, ≤, =, ≠, ≥, >, exponentiation, *ceil*, *floor*, etc.

.2 *BOOL*eans (*true*, *false*) -- with the operations: ~, ∨, ∧, ⊃, ≡.

.3 *QUOT*ations (*A*, *AB*, *ABC*, ...) -- with the operations: =, ≠.

.4 *TOKEN*s (-- for which no representation is required --) -- with the operations: =, ≠.

Other elementary data types could be considered.

(A data type is a set of objects and a set of operations. For a data type to be elementary its objects must all be considered elementary, i.e. having no structure.

The data types so far formally introduced were:

SET; TUPLE; MAP, FCT; TREE

They were all *COMP*osite. Thus the *OBJ*ects of the meta-language satisfy:

OBJ = ELEM | COMP
COMP = SET | TUPLE | MAP | FCT | TREE

where the five sub-domains of *COMP* are all considered disjoint.

In this tutorial we take *ELEM* to be:

ELEM = NUM | BOOL | QUOT | TOKEN

The following relations apply:

$$N_1 \subset N_0 \subset INTG \subset NUM$$

Otherwise the *ELEM* sub-domains are considered disjoint.)

If you need to introduce further objects or domains of objects -- by almost all means! At least as long as they are in keeping with the principles of abstractions (exemplified), and thus of the meta-language.

13.1 Rational *NUM*bers

Useful domains are:

NUM	Rational numbers, i.e. the numbers that arise as the result of integer (non-zero) divisions.
INTG	Integers
N_0	Natural numbers, in particular the positive integers including *0*.
N_1	Natural numbers larger than or equal to 1.

We only find a need to represent the integers:

$$\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$$

Useful operators/operations are:

+	addition
-	subtraction
*	multiplication
/	division: integer division leaving quotient voiding remainder, if *INTG* (N_0, or N_1) result is expected.

etc. If you would like to use:

ceil *r*	smallest integer larger than or equal to its only operand,
floor *r*	largest integer smaller than or equal to its only operand,
i^j	the exponentiation of i by j

etc., then we see no problem in you doing so. Etcetera -- for the introduction and proper explanation of your own operators (*mod*, *ln*, *rmd*, *sgn*, ...).

Programming Note

We abstract numerals by the numbers they denote. Thus in example 49 constants of expressions are *INTeGers* (and *BOOLeans*).

13.2 *BOOLeans* & Logic Expressions

We name the domain in question:

BOOL	Booleans values

In this tutorial we count on it having just two objects which we represent:

true, *false*

Useful operators/operations are:

~	negation,
∨	or,
∧, &	and (two forms provided), and
⊃, ⇒	implication (two forms provided).

In order to compact formulae we take the ∧, ∨, ⊃ operators as non-commutative! Thus if in *b1*∧*b2* *b1* is *false*, *b2* is never evaluated. Likewise if *b1* in *b1*∨*b2* is *true*.

Predicate Expressions

Besides the propositional expressions which can be formed using expressions denoting booleans and the above, conventional operators, proper use of the meta-language heavily relies on the possibility of forming quantified expressions:

$(\forall x)(P(x))$ $\qquad$ $(\forall x \in Set)(P(x))$

$(\exists x)(P(x))$ $\qquad$ $(\exists x \in Set)(P(x))$

$(\exists!x)(P(x))$ $\qquad$ $(\exists!x \in Set)(P(x))$

We 'read' the above expressions: "for all x the predicate $P(x)$ is true", "for all x in the set Set the predicate $P(x)$ is true", "there exists an x for which the predicate $P(x)$ is true", "there exists an x in the set Set for which the predicate $P(x)$ is true", "there exists a unique x for which $P(x)$ holds", and "there exists a unique x in the set Set for which $P(x)$ holds". In your reading the formulae 'aloud' you can e.g. vary as we did in the last two forms -- and you can instead of 'x' say 'objects x' or 'object x', etc.

Examples

We refer to examples 0, 9, 10, 11, 31 (10), etc.

Comments:

For the case where

$$Set = \{o_1, o_2, \ldots, o_n\}$$

the quantified expressions can be simply transliterated:

$(\forall x \in Set)(P(x))$

$\triangleq \quad P(o_1) \wedge P(o_2) \wedge \ldots \wedge P(o_n)$

$(\exists x \in Set)(P(x))$

$\triangleq \quad P(o_1) \vee P(o_2) \vee \ldots \vee P(o_n)$

$(\exists!x \in Set)(P(x))$

$\triangleq$ *if* $(\exists x \in Set)(P(x))$
then (*let* $o \in Set$ *be s.t.* $P(o)$ *in*
$(\forall x \in Set \setminus \{o\})(\sim P(x))$)
else false

Also:

$\sim(\exists x)(P(x)) \quad \equiv \quad (\forall x)(\sim P(x))$

$\sim(\exists x \in Set)(P(x)) \quad \equiv \quad (\forall x \in Set)(\sim P(x))$

13.3 *QUOT*ations

The subject domain is here named:

QUOT

Quotations are objects whose representations can be said to denote themselves. They are not to be confused with characters and characterstrings of conventional languages. We choose under-dashed sequences of (preferably) upper-case letters, and (rarely) other symbols:

$\underline{A}$, $\underline{B}$, ..., $\underline{Z}$, $\underline{AA}$, $\underline{AB}$, ..., $\underline{AZ}$, $\underline{BA}$, $\underline{BB}$, ..., $\underline{BZ}$, ..., $\underline{AA}\cdots\underline{A}$, ...

The only two operations intended are:

$=$	equality
$\neq$	non-equality

(Quotations correspond to the enumerated type objects of PASCAL, but we define no ordering on them.) If you wish to use quotations to model characters then observe that $\underline{ABC}$ is indivisible; and if you wish to use these modeling strings, then make tuples of quotations!

Example 67:

The following formulae may complete the syntax of example 49:

Int-Op = $\underline{ADD} \mid \underline{SUB} \mid \underline{MPY} \mid \underline{DIV}$

Bool-Op = $\underline{AND} \mid \underline{OR} \mid \underline{IMP}$

Rel-Op = $\underline{LARG} \mid \underline{LAEQ} \mid \underline{EQ} \mid \underline{NEQ} \mid \underline{SMAL} \mid \underline{SMEQ}$

An evaluation function for the expressions of example 49 might use these quotations:

```
eval-Expr(e)(ρ)=
    cases e:
       (mk-Infix(le,op,re)
          → (let lv : eval-Expr(le)(ρ),
                 rv : eval-Expr(re)(ρ);
             cases op:
                (ADD  → return(lv+rv),
                 SUB  → return(lv-rv),
                 ...
                 AND  → return(lv∧rv),
                 OR   → return(lv∨rv),
                 ...
                 LARG → return(lv>rv),
                 ...
                 SMEQ → return(lv≤rv))),
        ...)
    type: Expr ~→ (ENV ~→ (Σ ~→ (Σ VAL)))
```

Here it was assumed that *lv* and *rv* were of the appropriate type. Suppose the language is statically type checkable, then:

```
is-wf-Expr(e)(dict)=
   cases e:
      (mk-Infix(le,op,re)
         → (is-wf-Expr(le)∧
            is-wf-Expr(re)∧
            (let lt = e-tp(le)(dict),
                 rt = e-tp(re)(dict);
            (is-Bool-Op(op) → lt = BOOL = rt,
             T              → lt = INT = rt))),
       ...)
   type: Expr → (DICT → BOOL)
```

where:

$$DICT = Id \xrightarrow[m]{} (INT \mid BOOL)$$

13.4 *TOKEN*s

The subject domain is named:

TOKEN

Tokens are objects for which we seek no representation.

The only two operations provided are:

=	equality
≠	in-equality

Example 68:

When modeling the identifiers of software systems (e.g. the operating system directory, examples 19 & 29; the file system file identifiers, examples 20, 24-25, 27-28 & 36; the source language variable procedure and formal parameter identifiers, examples 30-31-32-33, 36 etcetera), when modeling labels of a goto language (example 61), or when modeling storage addresses, or as we call them: Locations (example 30-31) we use *TOKEN*s. That is:

Rid	=	*TOKEN*	(exs. 19),
Fid	=	*TOKEN*	(exs. 20),
Id	=	*TOKEN*	(exs. 30-33, 36,...),
Lbl	=	*TOKEN*	(ex. 61),
LOC	=	*TOKEN*	(ex. 30), and
Scalar-Loc	=	*TOKEN*	(ex. 31).

When in e.g. examples 24-25 & 27 we write suitably decorated *fid*'s, these are meta-language identifiers naming otherwise unrepresented *Fid* objects.

When in example 31 we need to 'generate' new *Scalar-Loc*ations, then we "pull them out of the *Scalar-Loc* bag":

(let sloc ∈ Scalar-Loc be s.t. P(sloc)
...)

sloc names a *TOKEN* object. We need not know "what it looks like"!

PART VI: POSTLUDE

This completes the informal introduction to the meta-language.

ACKNOWLEDGEMENTS:

The author is pleased to acknowledge all of his former colleagues of the IBM Vienna Laboratory, and Mrs.Jytte Søllested for her expert handling of eight IBM "golf ball" type-fonts!

EXAMPLE INDEX:

14.	Fibonacci Numbers	Tuple Enumerations.
15.	Pascal Triangle.	Implicit Tuple Enumeration.
16.	Value Stacks.	Tuple Domain & Primitive Tuple Operations.
17.	File Handling.	Primitive Tuple Operations.
18.	-"-	Applicative & Imperative Tuple Operations.
19.	System Directory.	Section Introduction: Maps, Abstract Syntax, Domains, Objects, Operations, Function Definitions.
20.	File System & Files.	Map Domains.
21.	Files.	-"-
22.	-"-	-"-
23.	-"-	-"-
24.	File System & Files.	Explicit Map Constructions.
25.	-"-	Implicit Map Constructions.
26.	Conceptual Example.	Implicit Map Construction: the Scope of Identifiers, Abstract Syntax.
27.	File System & Files.	Primitive Map Operations.
28.	-"-	Imperative Map Operations.
29.	System Directory.	Function Definitions, Map Operations.
30.	Progr.Lang.:States & Environments as Models of Variables & Scopes.	Map Domains, Primitive Operations, Abstract Syntax.

31.	Prgr.Lang.:Model of Block Concept & Storage.	Abstract Syntax, Map Domains, Function Definitions.
32.	Prgr.Lang.:Blocks.	Trees, Syntactic Domains.
33.	-"-	Trees, Abstract Syntax.
34.	Prgr.Lang.:Statements.	-"-
35.	System Directory & Direc= tory Update Commands.	-"-
36.	Prgr.Lang.:Procedures.	-"-
37.	Prgr.Lang.:Statements.	Selector Functions, Abstract Syntax.
38.	-"-	-"-
39.	Factorial Function.	Section Introduction: Functions, Function Definitions, Block Ex= pressions.
40.	Prgr.Lang.:Procedures.	Function Denotations, Semantic Domains, Syntactic Domains, Ab= stract Syntax, Elaboration Func= tions.
41.	Meta-Language: Type of Primitive Operations.	Function Types.
42.	Prgr.Lang.:Blocks & Sta= tements.	Abstract Syntax, Elaboration Function Definitions, Function Application.
43.	Prgr.Lang.	Section Introduction: Abstract Syntax.
44-46.	Conceptual Examples.	Abstract Syntax.
47.	Prgr.Lang.:Syntax	*is*-functions.

48.	Reflexive Domains.	Abstract Syntax, Map Domains, Map Objects.
49.	Prgr.Lang.:Expressions.	Abstract Syntax, McCarthy Condi= tional Clause, Function Definition.
50.	Factorial Function.	Section Introduction: Variables, Applicative & Imperative Function Definitions, Expression Blocks, The State.
51.	Prgr.Lang.:Modeling Variables, Input, Output.	Global Variables, The State.
52.	Prgr.Lang.:Assignment, Input, Output.	Elaboration- & Auxiliary Function Definitions, Variables, Assign= ment.
53.	Conceptual Example.	Imperative & Applicative Combi= nators.
54.	Prgr.Lang.:Structured Statements.	Abstract Syntax, Semantic Domains, Structured Clauses: *cases*, *if*, Applicative Definitions of Impe= rative Features.
55.	Prgr.Lang.:For-To-Do Statement.	Abstract Syntax, Semantic Domains, Elaboration Function Definitions.
56.	Prgr.Lang.:Compound Statement.	-"- &: *for-to-do*, Imperative De= finition.
57.	-"-	-"-, but Applicative Definition.
58.	Prgr.Lang.:Arbitrary Ordering.	-"-
59.	-"-, For-All Statement.	-"-
60.	Prgr.Lang.: While-Do	Recursive ("sledge-hammer") Defi= nition of ("egg") *while*.

61.	Prgr.Lang.:Guarded Repe= titive Command.	Elaboration Function Definition, Applicative Definition.
62.	Prgr.Lang.:GOTO & Block Concept Model.	*exit* and *tixe*
63.	System Directory: Hierar= chical Entry Access.	*exit* and *trap*
64.	Prgr.Lang.:Zahn's Event Mechanism.	*exit* and *tixe*
65.	Prgr.Lang.:Procedures.	Abstract Syntax, Syntactic & Se= mantic Domains, Elaboration Func= tion Definitions.
66.	Prgr.Lang.:States.	Imperative & Applicative 'States'.
67.	Prgr.Lang.:Expressions.	Quotations.
68.	Prgr.Lang.:Names & Ad= dresses.	Tokens.