SOFTWARE ABSTRACTION PRINCIPLES:

TUTORIAL EXAMPLES OF AN OPERATING SYSTEM COMMAND LANGUAGE SPECIFICATION AND A PL/I-LIKE ON-CONDI= TION LANGUAGE DEFINITION

Dines Bjørner

Abstract:

Four groups of two, either complementing or contrasting abstraction principles are isolated: REPRESENTATIONAL and OPERATIONAL abstraction; CONFIGURATIONAL versus HIERAR= CHICAL abstraction; STATE-MACHINE- versus REFERENTIALLY TRANSPARENT, FUNCTIONAL- abstraction; and DENOTATIONAL versus MECHANICAL specification. Tools, techniques and examples are presented for, respectively of, each of the eight principles.

CONTENTS

1. INTRODUCTION

The problem to be solved by the methods outlined in this paper is ultimately the construction of correctly functioning, well-understood, pleasing, yet complex software. It is our thesis that one way of achieving this is to use a systematic software development method which provides a formalized structure for stepwise, increasingly more detailed arguments of correctness -- a method based on systematically deriving abstractions into concrete realizations; i.e. on a-priori, synthetic, constructive proofs rather than a-posteriori, analytic proofs. Thus we see our methodology as starting with an abstract specification of the desired software item. This paper then is concerned with some of the techniques used in achieving such definitions. In [Bjørner 77c] we cover the problems of mapping abstractions into concretizations. We are there in particular concerned with the systematic derivation- and proof techniques.

The objectives of an abstract software specification are basically twofold: that the resulting document serve as the basis from which an implementation take place formally, and with respect to which correctness criteria be stated, and a proof given. Hence we require that the specification (or: meta-) language be formal. Also: that the document, and it alone, be the specification from which we develop user's reference (and other) manuals!

The objectives of the abstraction principles explicitly expounded in this paper are several: That the specifications be precise, non-contradictory and complete; that they be short, well-organized and comprehensible; that the described systems be well-conceived, free from mis-conceptions, conceptually clean, lean and with an optimum of semantically relevant notions; that their properties be well understood, possesing desirable properties and with a minimum of ad-hoc ideas. We find [Liskov 75] to give a fine discussion of the above points.

On Abstraction Techniques

Before going on to exemplify uses of the meta-language let us first also summarize the principles used in applying constructs of this language. One thing is the notation: its syntax & semantics. Another thing is the intent with which it was to be applied; its pragmatics. Any notation can be used against its will, even a good one.

If you consider META-IV as an ultra-high-level programming language, i.e. one which although it is intended only to specify software actually results in programs which can be considered implementations, albeit on a very abstract, and in most cases not mechanizable, level, then which are the programming disciplines around which the meta-language evolved, and whose application exploits its capabilities to the fullest?

We consider these to be the pre-dominant abstraction techniques: representational- & operational- abstraction; configurational vs. hierarchical abstraction; referential transparency vs. abstract state machine modeling; and mechanical- vs. denotational abstraction.

At each design step and at each specification stage we carefully review the appropriateness of each abstraction choice: its level when considering e.g. representational- & operational; and mechanical- vs. denotational abstractions; its mixture or blend of configuration and hierachy, i.e. bottom-up synthetic vs. top-down analytic features; respectively of referential transparency or applicativeness vs. abstract state machine imperativeness; and finally also its balance of explicitness vs. implicitness.

Before going into a brief characterization of each of the eight abstraction principles, an outline is first given of the basic parts that make up our specification document. A rationale is given for their inclusion.

The software 'function' to be modelled normally accepts inputs, emits outputs, achieves the desired transformations of inputs into outputs by means of internal data structures, and specifies transformations in terms of function definitions (procedures, process descriptions, operations). Our model hence consists of two basically distinct parts: one containing the descriptions of the input/output and internal domains -- subsequently referred to as syntactic-, respectively semantic domains. Another part containing a number of elaboration- (and auxiliary-) function definitions which to combinations of input- and semantic- domain objects ascribes their meaning, in terms of either semantic domain object transformations or these latter combined with output domain objects.

In the next paragraphs we now treat the abstraction principles individually.

By REPRESENTATIONAL ABSTRACTION we understand the specification of objects irrespective of their implementation, and such that the chosen abstractions as closely as possible only reflect relevant and intrinsic properties.

Representational abstraction of classes of objects is here expressed in terms of so-called abstract syntax. Individual instances of objects can be abstracted by corresponding expressions of the meta-language. Representational abstraction is applied in the definition of both syntactic- and semantic- domains and domain objects.

By OPERATIONAL ABSTRACTION we understand the specification of functions in extension. That is: we are primarily interested in the properties of the functions we define, notably in the properties of that which our defined functions define (be they functions themselves), i.e. in what they compute; less -- if at all -- interested in how results are computed, i.e. not in functions in extension.

Operational abstraction is here expressed primarily in the form of function definitions. We express these either by a pair of *pre-* and *post-*conditions on the functions sought, or by a constructive function definition. The former kind are thus usually more implicit, i.e. abstract, than the latter kind (of more explicit definitions). This latter form is normally still abstract, in that it usually internally employs operational abstraction on representationally abstract objects. Operational abstraction is used in the definition of the elaboration functions, as well as functions of our model auxiliary to these, and to (*is-wf-*θ) *well-formedness-* context, static condition and dynamic constraint-, predicates 'narrowing' the (θ-) domains otherwise defined by abstract syntaxes.

By CONFIGURATIONAL ABSTRACTION we understand the step-wise definition and realization of a model, or major model components, which proceeds in a synthetic manner in conceiving and documenting the desired system -- from the bottom-up -- by building layers of abstraction upon more concrete bases. From 'physical machines' we create (the illusion of) 'virtual machines': changing raw capabilities into sophisticated concepts. Configurational abstraction composes low-level abstractions (or rather 'mechanizations') into higher-level abstractions.

Configurational abstraction, in its inner, foundational steps, is usually expressed in rather concrete representational- and operational

forms. In its outer, so-called 'abstract' layers, expressional means are normally tied to those of the procedure, module-, and class- like 'abstractions'. Configurational abstraction -- as a specification technique -- is brought into play whereever uncertainties concerning either desired functions, and/or efficient realizability dominate our understanding of what system we are in fact aiming at.

The resulting design: its abstraction & implementation can usually, and to great advantage however, be hierarchically documented.

By HIERARCHICAL ABSTRACTION we understand the stepwise definition of a model which proceeds in conceiving and documenting, in an analytical fashion -- from the top down -- the desired system (components) by decomposing basic overall dominating concepts and transformations into constituent ones.

Hierarchical abstraction techniques can fully exploit the representational and operational abstraction techniques descussed and elsewhere illustrated in this paper. And hierarchical abstraction is applied where a sufficiently deep understanding of our system haș eventually transpired.

Configurational abstraction have been used extensively in operating systems designs [Dijkstra 68; Hansen 73]. Hierarchical abstraction mostly in e.g. programming language semantics [e.g. Bekić 74, Henhapl 78] and relational data base (system and query language) formalization [Hansal 76, Nilsson 76].

Any one abstraction, and almost any actual, conventional program algorithm, usually exhibits some mixture of both. Only when the choice between configurational- and hierarchical abstraction has been made as the consequence of a careful study, and only when the resulting documentation (respectively program code) is clear, does the specification appear transparent. The subject of choosing a bottom-up versus a top-down abstract, and/or algorithmic design idea programming strategy is, however, a seriously undeveloped one and we shall unfortunately not contribute much to a clarification in this paper. It is our hope, though, to return later to a study of their duality. Step-wise refinement, i.e. top-down, hierarchical abstraction in program algorithm and data structure design is extensively convered in [Wirth 71,73,76].

By a DENOTATIONAL SPECIFICATION [Scott 71,72; Tennent 73,76; Mosses 75; Milne 76] we understand a definition which ascribes meaning to (composite) syntactic domain objects by functionally composing meanings of proper constituent parts. Thus denotational abstraction almost invariably calls for 'homomorphic' programming [Burstall 69, Morris 73, Reynolds 74, ADJ 77], i.e. referential transparency together with syntactic object 'driven' function specifications. And denotational definitions achieve their characteristics by employing semantic domain objects of high, functional order. Thus the meaning of a program (construct) is generally seen as a state transformation function, a state transformer, independent of program (input) data.

By a MECHANICAL ABSTRACTION (which may hardly be an abstraction at all!) we understand a description which assigns meaning to a program (construct) by explicitly prescribing computation (i.e. state-) sequences given input data, thus computing result values. The meaning then becomes the state transition sequence, not the function from begin states to end states. A mechanical definition is said to be so (or to be operational) since its direct realization is immediate (and programmable in most languages).

Examples

We now illustrate some of the abstraction techniques through two examples. The software item to be specified in section 2 is a command language for an operating system -- naturally: of hypothetical, illustrative nature. In section 3 we give the denotational semantics of a non-trivial language with PL/I-like On-Conditions. The abstraction principles examplified are these: representational- and operational abstraction; functional, referentially transparent abstraction in section 2, and abstract machine/state programming featuring both local and global states, and local semantic domain objects in section 3. Finally, and almost exclusively: denotational semantics, whereby the model is almost invariably forced to be hierarchically specified.

It is, however, an alltogether not un-important aim also to convince you of the utility of abstractly specifying software in general, and -- to take the first choice as an example, to suggest that future, professional paper proposals for e.g. command- and data base query languages be formally, hence precisely stated.

The presentations are both according to our basic principle: formulae first, and then their national/natural language explication immediately subsequent. No introduction smoothly 'tricking' you into a subsequent formalism -- as if to excuse this latter!

2. EXAMPLE I: An Abstract Processor for an Interactive, Operating System Command Language

A. Syntactic Domains

1	*Cmd*	=	*In* | *Clg* | *Dl*					
2	*In*	::	(*Input*|*Source*|*Link*)	*Id*				
3	*Clg*	=	*C* | *CL* | *CLG* | *L* | *LG* | *G*					
4	*C*	::	*Cid*	(*Source*|*Id*)	[*Id*]			
5	*CL*	::	*Cid*	(*Source*|*Id*)	[*Id*]	(*Link*|*Id*)	[*Id*]	
6	*CLG*	::	*Cid*	(*Source*|*Id*)	[*Id*]	(*Link*|*Id*)	[*Id*]	(*Input*|*Id*)
7	*L*	::			*Id*	(*Link*|*Id*)	[*Id*]	
8	*LG*	::			*Id*	(*Link*|*Id*	[*Id*]	(*Input*|*Id*)
9	*G*	::					*Id*	(*Input*|*Id*)
10	*Dl*	::	*Id*					

Annotation

1 A job control *C*ommand is either a file *In*put data command, or (|) a (partial or complete) *C*ompile-*l*ink-*go* command, or it is a *De*lete file command.

2 An *In*put command has two parts: the data part containing either *Input*, *Source* or *Link* data itself, and the part which *Id*entifies the file name to be given to this data.

3 A *C*ompile-*l*ink-*go* command is either a *C*OMPILE, a *C*OMPILE-*L*INK, a *C*OMPILE-*L*INK-*GO*, a *L*INK, a *L*INK-*GO* or just a *GO* command.

4 A *C*OMPILE command has three parts: one part *Id*entifies a *C*ompiler, another either directly contains the *Source* text or *Id*entifies such a text (in the *FILE* state component, see below), and a third, optional ([]) part *Id*entifies the name to be given to the object (module) resulting from compilation and to be optionally stored in the *FILE*.

...

6 A *CLG* command additionally has a component which is either the *Link* data itself or *Id*entifies such a link data file, a component which optionally *Id*entifies a file name for the linked load (module), and a component which either is the *Input* data for the executing load module, or *Id*entifies such an input data file.

Comments concerning Abstraction Principles

Observe that we have attempted only to describe syntactically essential components of commands -- and then only abstractly, irrespective of their possible written forms:

"ALGOL" compile "SID" with link ["FID" → "PRINT"]
and execute with "DID"

Thus we have as far as possible avoided any mention of what the commands effect, i.e. their meaning. Of course, your previous technical knowledge may already have initiated some personal 'feel' for what they might stand for. This is because I have chosen suggestive mnemonics. I could as well have chosen x's, y's and z's, and still obtained exactly the same domains of mathematical objects. Only when I deal with concepts for which there either is no previous familiarity or which may be ambiguously understood when applying only an intuitive understanding, i.e. when not reading the entire model, shall I have to take extraordinary care in my annotations, and in judiciously keeping these and the discussion within purely syntactic domain, purely semantic domain, respectively purely semantic *Elab*oration function subject boundaries.

The commands have been representationally abstracted. There is no word here about positional parameters, mnemonic keywords nor of default such, no talk of delimiters or other syntactic 'sugar'. The objects denoted by this abstract syntax (for a definition of *Id*, *Cid*, *Input*, *Source* and *Link*, see below) are in fact mathematical, not characterstrings. We use the same kind of abstract syntax definitional facility for specifying both syntactic and semantic domain objects, as well as their definiens logical type expressions are used in *type* definitions for *Elab*oration- and Auxiliary functions.

The presentational structure of this abstract syntax is basically hierarchical.

B. Semantic Domains

1	Σ	::	*FILE SYS UTIL*
2	*FILE*	=	*Id* $\vec{m}$ *Data*
3	*SYS*	=	*Cid* $\vec{m}$ *Comp*
4	*UTIL*	=	*Uid* $\vec{m}$ *(Data | QUOT*)*
5	*Data*	=	*Input | Source | Object | Link | Load*
6	*Input*	=	...
7	*Source*	=	...
8	*Object*	=	*Link* $\vec{m}$ *Load*
9	*Link*	=	*Id* $\vec{m}$ *(Id | Const)*
10	*Load*	=	*Input* $\overset{\sim}{\rightarrow}$ *(*Σ $\overset{\sim}{\rightarrow}$ *(*Σ *Output))*
11	*Output*	=	...
12	*Comp*	=	*Source* $\overset{\sim}{\rightarrow}$ *(Object | Text)*
13	*Cid*	=	*FORTRAN | ALGOL | COBOL | PL/I | ...*
14	*Uid*	=	*INPUT | OBJ | LINK | LOAD | OUTPUT | MSG | ...*
15	*Text*	=	...
16	*Id*	=	*TOKEN*

Annotations

1 In order to explain the meaning of our operating system job command and control language we introduce an abstract state space Σ. A state, $\sigma \in \Sigma$, has three components: a *FILE*-, a *SYS*tem programs-, and a *UTIL*ity object.

2 A *FILE* object is a finite domain (= ...) *m*ap from *Id*entifiers (i.e. file names) to *Data* (-sets).

3 The *SYS*tem object is a ... map from (here only:) *C*ompiler *id*entifier names to the *Comp*ilers themselves. (Subsequently we might contemplate adding other systems programs to *SYS*: sort-, copy-, merge-, etc..)

4 The *UTIL*ity component is a ... map from *id*entifications of *U*tilities to either *Data* or lists (*) of *QUOT*ations (objects which you may wish to think of as characterstrings).

5 Besides *Input-*, *Source-* and *Link- Data* (which can be directly inserted into the *FILE* by the command language user) *Object-* and *Load* can be filed (as the result of successfully executing one of the commands *C*, *CL*, *CLG* respectively *CL*, *CLG*, *L*, *LG*).

...

8 An *Object* (module) is a map from *Link* to *Load*. (That is: the result of a compilation is to be an object of type *Object*. The free *Id*entifiers of the (compiled) *Source* text have not yet been bound to their meaning -- which are to be those of *Id*entified (names of) filed *Data* or *Const*ants.)

9 *Link* is a ... *m*ap from (free) *Id*entifiers (of (compiler) texts) to *Id*entifiers (of filed *Data*) or *Const*ants.

10 A *Load* (module) is a function from *Input* to *state transforming functions* yielding *Output*.

...

12 A(ny) *Comp*iler is a (pure, i.e. not state, $\sigma \in \Sigma$, dependent) function from *Source* to the union (|) domain of *Object* and *Text* (-- the former are to be the result of successful compilation the latter of a syntactically, and otherwise erroneous, text (instead yielding *diagnostics Text*)).

13 Suggests possible compilers!

14 The *UTIL*ity components are here primarily intended to 'store' temporary results in/between the *C-L-G* steps; for details see the *Elab-clg*, *compile* and *bind* function definitions below.

...

16 *Id*entifiers are further unspecified *TOKEN*s.

Comments on Abstraction Principles

1 Suggests or relies on a configurative abstraction: from the more primitive components is built a more sophisticated. The user -- we conjecture -- need think only of Σ, and does, as far as conceptual understanding goes, not need to know of its decomposition.

8-10,12 Suggests not only a hierarchical decomposition, but relies on *Comp* and *Load* objects as primitives. These are representationally highly abstracted.

Discussion of Abstraction Choices

The crucial abstractions are these: *Comp* and *Object*. That of *Load* -- and to an even lesser degree that of *Link* -- we consider almost trivial. Anticipating subsequent semantic *Elab*oration function descriptions -- which, of course, really were developed in 'parallel' with the above semantic domain definitions -- we hinge our model on the ability of the *Comp*iler to produce exactly a function of the logical type *Object* (disregarding here diagnostic *Texts*). With the types we have ascribed to *Object*, and in particular to *Load*, it can be shown that the *Comp*iler in fact is the function denoted by an appropriate mathematical- or denotational semantics definition, Ψ, of the *Source* language. Ψ is to take the *Source* text and produce a function which permits an act of *binding*. *Binding* is a function which takes an *Object* module and some *Link Data* and produces some *Load* module. Ψ creates this function by letting the free *Id*entifiers of the *Source* text be mappable to a variety of *Const*ants or *FILE Data Id*entifiers:

Example:

$$
\textit{obj-k:}\left[\begin{array}{ll}\left[\begin{array}{ll}\textit{pid1} & \rightarrow\ \textit{fid-i1}\\ \textit{pid2} & \rightarrow\ \textit{fid-i2}\\ \ldots & \\ \textit{pidn} & \rightarrow\ \textit{fid-in}\end{array}\right] & \rightarrow\ \textit{load-i}\\ \ldots & \\ \left[\begin{array}{ll}\textit{pid1} & \rightarrow\ \textit{fid-j1}\\ \textit{pid1} & \rightarrow\ \textit{fid-j2}\\ \ldots & \\ \textit{pidn} & \rightarrow\ \textit{fid-jn}\end{array}\right] & \rightarrow\ \textit{load-j}\end{array}\right]
$$

$obj\text{-}k \in Object$
$load\text{-}i \in Load$
$pid... \in Id$
$fid... \in Id$

end example.

In fact, we can impose varying degrees of easily formalizable, hence tersely expressible constraints on *Object*'s and *Link*'s:

C. Semantic Domain Consistency Constraints

Simple, Lax Version:

```
1.0  is-wf-Σ(mk-Σ(f,s,u))=
 .1      (∀o∈rng f)
 .2          (is-Object(o) ⊃ (∀l1,l2 ∈ dom o)(dom l1 = dom l2))
```

Restrictive Version:

```
2.0  is-wf-Σ(mk-Σ(f,s,u))=
 .1      (∀o∈rng f)
 .2          (is-Object(o) ⊃ (∀l1,l2 ∈ dom o)
 .3                              ((dom l1 = dom l2)
 .4                              ∧(rng l1↘Const ⊆ dom f)))
 .5     ∧(∀l∈rng f)
 .6          (is-Link(l)  ⊃  (rng l↘Const ⊆ dom f))
```

Annotations:

(1.0-1.2 ≡ 2.0-2.3)

2.1-2.4 For each *Object*, *o*, in *fILE* it must be the case that all of its *dom*ain *l*inks have the same *dom*ain of (free *Source* text) *Id*entifiers -- since, naturally, that object, *o*, is the result of just one compilation of exactly one *Source* text. (But: binding these free *Id*entifiers to different *fILE*d *Data* and *Const*ants should certainly create distinct *Load* modules, cf. example above.)

2.4 -- And range *Id*entifiers of *Object* module *Link*'s must (in this restrictive version) already have been defined (i.e. *fILE*d *Data*.

2.5-2.6 -- Similar for *fILE*d *Data*.

Comments on Abstraction Principles

The domain consistency (inspection) functions are operationally abstracted in terms of applicative, referentially transparent expressions exploying quantified predicates, i.e. staying aloof of order of inspection!

D. Dynamic Command-State Consistency/Constraint Relations

```
1.0  pre-Elab-cmd(<cmd,σ>)=
 .1      (let mk-Σ(f,,) =.σ in
 .2       cases cmd:
 .3          (mk-In(,id)  → id ~∈ dom f,
 .4           mk-Dl(id)   → id ∈ dom f,
 .5           T           → pre-Elab-clg(<cmd,σ>))
```

```
2.0   pre-Elab-clg(<clg,σ>)=
 .1     (let mk-Σ(file,sys,) = σ in
 .2      cases clg:
 .3        (mk-C(k,s,o)          → ((k ∈ dom sys)                                  ∧
 .4                                  (is-Id(s) ⊃ ((s ∈ dom file)                   ∧
 .5                                               is-Source(file(s))))             ∧
 .6                                  ((o≠nil) ⊃ o ~∈ dom file)),
 .7         mk-CL(k,s,o,l,e)     → (pre-Elab-clg(<mk-C(k,s,o),σ>)                  ∧
 .8                                  (is-Id(l) → ((l ∈ dom file)                   ∧
 .9                                               is-Link(file(l))                 ∧
 .10                                              (rng(file(l))\Const ⊆ dom file)),
 .11                                 T         → (rng l\Const ⊆ dom file))       ∧
 .12                                 ((e≠nil)  ⊃ e ~∈ dom file)),
 .13        mk-CLG(k,s,o,l,e,i) → (pre-Elab-clg(<mk-CL(k,s,o,l,e),σ>)             ∧
 .14                                 (is-Id(i) ⊃ ((i ∈ dom file)                   ∧
 .15                                              is-Input(file(i))))),
 .16        mk-L(o,l,e)          → ((o ∈ dom file) ∧ is-Object(file(o))           ∧
 .17                                 (is-Id(l) → (l ∈ dom file)                    ∧
 .18                                              is-Link(file(l))                 ∧
 .19                                              (rng(file(l))\Const ⊆ dom file)),
 .20                                 T         → (rng l\Const ⊆ dom file))       ∧
 .21                                 ((e≠nil)  ⊃ (e ~∈ dom file))),
 .22        mk-LG(o,l,e,i)       → (pre-Elab-clg(<mk-L(o,l,e),σ>)                 ∧
 .23                                 (is-Id(i) → ((i ∈ dom file)                   ∧
 .24                                              is-Input(file(i))))),
 .25        mk-G(e,i)            → ((e ∈ dom file) ∧ is-Load(file(e))             ∧
 .26                                 (is-Id(i) ⊃ ((i ∈ dom file)                   ∧
 .27                                              is-Input(file(i))))))))
```

Annotation:

Successful *Elab*oration of *command*s depend on basically three aspects: (1) one is checkable without actually applying the functions implied (e.g.: *C*ompile, *L*ink and *G*o), but depends on the relation between the static *command* and the dynamic state, $\sigma \in \Sigma$. (2) The other can only be known by actually applying the implied (C,L,G) functions. In this example there are no, (0), static context conditions imposable on *command*s only, as is e.g. the case with the definition and use of (variable, procedure and label) identifiers in block-structured procedure-oriented programming languages with a fixed, strong type system. *pre-Elab-cmd* and *pre-Elab-clg* deals with (1). The Auxiliary functions *compile* and *bind* invoked by the *Elab-clg* functions takes care of (2).

1.3 The *Id*entifier naming *Data* to be *fILE*d must not already be used, i.e. be 'defined'.

...

2.5 The *fILE*d *Source* text *Id*entifier must identify a *Data* object of type *Source*.

2.6 If an identifier, *o*, is specified (for the thus implied *f*iling of the *Object* module to result from *C*ompilation) then *o* must not already be defined.

...

2.10,19 If only C.1 (not C.2) is specified, then this is the last 'time'/opportunity to 'catch' the equivalent of C.2.4.

(Notice that we have not checked *Input Link Data* in D.1.3!)

Comments on Abstraction Principles

Again the functions are operational abstractly specified. The structure of the function definition follows that of the abstract syntax definition of (primarily) *Command*s and (secondarily) Σ.

E. Elaboration Function Types

1 *type:* *Elab-cmd:* *Cmd* $\xrightarrow{\sim}$ (Σ $\xrightarrow{\sim}$ Σ)
2 *Elab-clg:* *Clg* $\xrightarrow{\sim}$ (Σ $\xrightarrow{\sim}$ Σ)

Annotation:

Given a *command* the *Elaborate-command* function ascribes to it, as its semantics, a state transformer, i.e. a function from (Operating System) states to states. This is the denotational semantics viewpoint. Thus, when a specific *command*, which denotes the function, say ψ, is executed in a state σ, $\sigma \in \Sigma$, then a new state σ', $\sigma' \in \Sigma$, will arise:

let ψ = *Elab-cmd(cmd)*
$\Psi(\sigma) = \sigma'$

Comments on Abstraction Principle:

We choose this level of abstraction, in contrast to a mechanical semantics definition, since we have not yet decided on which machine, and/or how we specifically intend, to implement our command language.

Also the choice concerning the operational characterstics has been limited to the denotational ones since we anyway have decided not yet to consider how e.g. *Comp*ilers are to be realized, only that they perform some function, and the type of this function.

F. Elaboration Function Definitions

```
1.0  Elab-cmd(cmd) =
 .1     if pre-Elab-cmd(cmd)σ
 .2        then
 .3           (let mk-Σ(f,s,u) = σ in
 .4            cases cmd:
 .5               (mk-Id(d,i) → mk-Σ(fU[i→d],s,
 .6                                  u + [MSG → u(MSG)⌢<FILED>]),
 .7                mk-Dl(id)  → mk-Σ(f\{id},s,
 .8                                  u + [MSG → u(MSG)⌢<DELETED>]),
 .9                T          → Elab-clg(cmd)σ))
 .10       else
 .11          error


2.0  Elab-clg(clg)σ=
 .1     (let mk-Σ(f,s,u) = σ in
 .2     (trap exit(ξ) with ξ in
 .3      cases clg:
 .4         (mk-C(k,t,o)          → compile(k,t,o)σ,
 .5          mk-CL(k,t,o,l,e)     → (let σ' = compile(k,t,o)σ in
 .6                                  bind(l,e)σ'),
 .7          mk-CLG(k,t,o,l,e,i) → (let σ' = compile(k,t,o)σ in
 .8                                  let σ" = bind(l,e)σ'    in
 .9                                  execute(i)σ"),
 .10         mk-L(o,l,e)          → (let σ' = mk-Σ(f,s,u+[OBJ→f(o)])  in
 .11                                 bind(l,e)σ'),
 .12         mk-LG(o,l,e,i)       → (let σ' = mk-Σ(f,s,u+[OBJ→f(o)])  in
 .13                                 let σ" = bind(l,e)σ'              in
 .14                                 execute(i)σ"),
 .15         mk-G(e,i)            → (let σ' = mk-Σ(f,s,u+[LOAD→f(e)]) in
 .16                                 execute(i)σ')))
```

Annotation:

1.1 Execution of a *cmd* depends on it satisfying the function independent, syntactic- and semantic domain dependent consistency constraints specified by *pre-Elab-cmd* (and detailed there and in *pre-Elab-clg*).

1.6,8 *MeSsaGes* concerning successful completion states (status's) are 'posted' in the *UTILity* MesSaGe component.

2.2 Erroneous, execution-time checkable only, execution of the *compile* or *bind* functions shall lead to *exit*s being *trap*ped here -- aborting further execution of steps in the *CL*, *CLG*, *LG* commands. (Abnormal termination in the *C*, *L* cases are of course also *trap*-ped here with the same effect as if not abnormally terminated through an *exit*!)

2.4 To Compile is to *compile*.

...

2.7 To Compile-Link-Go is to *compile* (in one state, σ), then (;) to *bind* in the state, σ', resulting from compilation; and to *execute* in the state, σ", resulting from binding.

2.10 The *bind* operation expects the *UTILity* OBJect component to contain the named (*o*) *Object* module (from the *file*).

2.5,7 Thus the *compile* operation deposits a successfully compiled *Object* in the *UTILity* OBJect component.

2.15 like 2.10 but now for *execute* and LOAD,

2.8,13 -- like 2.5,7, but now for the objects mentioned above!

2.7-9 These three lines could be written:

execute(i)(bind(l,e)(compile(k,t,o)σ))

and so could lines 2.5-6 and 2.13-14, in their form.

Discussion of Abstraction Choices:

The restriction that a *CLG* command must have not only its "*C*-", but also its "*L*-" and "*G*-" (syntactic) components agreeing with certain state components in order that *Elab*oration of any part of the *CLG* command may be commenced, may seem rather limiting. We have, however, brought this semantics only for the purposes of exemplifying modelling techniques -- not in order to advocate the virtues of one particular command language over those of another. Only when we master our specification tools do we feel ready to seriously, and sensibly, embark on 'architectural' designs. Thus it is relatively easy, to 'chop' the *pre*-condition specfication up into separate parts, and merge these (or 'calls' thereon) with the constructive parts of the present *Elab*oration functions, thus permitting partial *Elab*oration of e.g. *CLG* commands.

Comments on Abstraction Principles:

The semantics assignment has been part implicitly-, part explicitly specified: one could replace *Elab-cmd* with a *post-Elab-cmd* specification:

```
3.0  post-Elab-cmd(<cmd,σ>,σ')=
 .1      (let mk-Σ(f,,)  = σ,
 .2           mk-Σ(f',,) = σ' in
 .3       cases cmd:
 .4          (mk-In(d,i)  → f' = fU[i→d],
 .5           mk-Dl(id)   → f' = f\{id},
 .6           T           → post- lab-clg(<cmd,σ>,σ')
```

which, together with *pre-Elab-cmd*, uniquely determines *Elab-cmd*. (Provided of course we either specify *Elab-clg* accordingly through its *post*-, or imply the *post-Elab-clg* defined by 2 above!)

In general, if:

type: *F*: $A \xrightarrow{\sim} B$

then:

type: *pre-F*: *A* → *BOOL*
type: *post-F*: *(A B)* → *BOOL*

with:

$$pre\text{-}F(a) \supset (\exists!b \in B)(F(a) = b)$$
$$pre\text{-}F(a) \wedge F(a) = b \supset post\text{-}F(a,b).$$

The definition of the *Elab-clg* proceeded on the basis of an iterative stepwise refinement: existence of *compile*, *bind* and *execute* was postulated after the crucial issue of their logical types were first settled -- see G below! The iteration from the internal specification of the first two of these functions occured as the result of first planning that there be an *exit* within them, and then actually fixing the places of these *exit*s and the type of the value(s) (Σ) being "returned".

G. Auxiliary Function Types

1	*type:*	*compile:*	*Cid*	*(Source* | *Id)*	*[Id]*	$\tilde{\rightarrow}$	*(*Σ $\tilde{\rightarrow}$ Σ*)*
2		*bind:*		*(Link* | *Id)*	*[Id]*	$\tilde{\rightarrow}$	*(*Σ $\tilde{\rightarrow}$ Σ*)*
3		*execute:*		*(Input* | *Id)*		$\tilde{\rightarrow}$	*(*Σ $\tilde{\rightarrow}$ Σ*)*

Discussion/Comments:

There is an unfortunate asymmetry between these functions: *compile* receives all the information it requires through its three arguments, but both *bind* and *execute* are not explicitly passed information about the *Object* module to be linked, respectively the *Load* module to be executed -- instead these objects are to be looked up in the *UTIL*ity components: *OBJ*, respectively *LOAD*; a fact which is hidden. This abstraction choice was made after some (trivial) experiments with explicit passing: the present solution was found not only to balance the needs better between on one hand the *CL* and *CLG* commands, and those of the *LG* and *G* commands on the other hand, but also to be in some accord with actual operating system practice (SYSLINK, ...).

H. Auxiliary Function Definitions

```
1.0  compile(k,t,o)σ=
 .1     (let mk-Σ(f,s,u) = σ                                  in
 .2      let source = if is-Id(t) then f(t) else t            in
 .3      let obj = (s(k))(source)                             in
 .4      if  is-Text(obj)
 .5         then exit(mk-Σ(f,s,u+[MSG → u(MSG)⌢<obj>]))
 .6         else (let u' = u+[OBJ → obj,MSG → u(MSG)⌢<COMPILED>] in
 .7                 if o=nil
 .8                    then mk-Σ(f,s,u')
 .9                    else mk-Σ(f∪[o→obj],s,u')))


2.0  bind(l,e)σ=
 .1     (let mk-Σ(f,s,u∪[OBJ→obj]) = σ                        in
 .2      let lnk = if is-Id(l) then f(l) else l               in
 .3      if lnk ∈ dom obj
 .4         then (let u' = u+[LOAD → obj(lnk),MSG → u(MSG)⌢<LINKED>]  in
 .5                 if e=nil
 .6                    then mk-Σ(f,s,u')
 .7                    else mk-Σ(f∪[e→obj(lnk)],s,u'+[MSG→u(MSG)⌢<FILED>])
 .8         else exit(mk-Σ(f,s,u+[MSG → u(MSG)⌢<ERRONEOUS-LINK>])))


3.0  execute(i)σ=
 .1     (let mk-Σ(f,s,u∪[LOAD→load]) = σ                      in
 .2      let input = if is-Id(i) then f(i) else i             in
 .3      let (mk-Σ(f',s,u'),output) = load(input)             in
 .4      mk-Σ(f',s,u'+[OUTPUT → output]))
```

Comments on Abstraction Principles

It is especially in this specification step that the real 'power' of our abstraction appears to yield their maximum return.

Annotations:

1.3 Recalling that the logical type of the *Compiler*, $s(k)$, is *Source* $\tilde{\rightarrow}$ *(Object | Text)* and that that of *source* is *Source*, we see that that of *obj* is either *Object* or *Text* -- concerning which we assume disjointness of domains, although that has not yet been imposed.

1.9 If *o* was specified, then the *Object obj* is to be *filed*.

1.6 In any case a successfull *compile* leaves *obj* in the *UTILity* under *OBJ*.

2.3 A *bind* is only successful if the right *Link* information is provided.

3.3 The logical type of *Load* is (see B.10) *Input* $\tilde{\rightarrow}$ *(*Σ $\tilde{\rightarrow}$ Σ *Output)* permitting executing (user) programs to access (Σ $\tilde{\rightarrow}$ ...) and update (... $\tilde{\rightarrow}$ Σ) e.g. *files*, thus changing the state.

Discussion of F & H, Functional versus Machine State Programming:

The specification of *Elab-cmd* has been kept completely functional, thus referentially transparent. The meaning of a command is the simple functional composition of the meanings of the command components -- and the overall meaning remains unchanged if we alter any syntactic component to another, syntactically different one (*Id* for *Source*, or: *Id* for *Link*, or: *Id* for *Input* H.1.2, H.2.2. respectively H.3.3) having the same constituent meaning.

3. EXAMPLE II: A PL/I-like On-Condition Language

We begin by listing and annotating formulae. Then we end by discussing abstraction principles.

A. Syntactic Domains

Progr = *Block*
Block :: *Id-set* $(Id \xrightarrow[m]{} Proc)$ $Stmt^{+}$
Proc :: *s-pml:(s-Id:Id s-Tp:(LOC|PROC))** *Block*
Stmt = *El-Stmt* | *Call* | *On-Unit* | *Signal* | *Revert*
Call :: *Id* *Expr**
On-Unit :: *Cid* *Proc*
Signal :: *Cid* *Id**
Revert :: *Cid*
Expr = *Id* | *Const* | *Infix*
Const :: *INTG*
Infix :: *Expr* *Op* *Expr*
Op = *ADD* | *SUB* | *MPY* | *DIV* | *EQ* | *NEQ*
Id ⊃ *TOKEN* } *Id* ∪ *Lbl* = {}
Lbl ⊃ *TOKEN* }
Cid = *OFL* | *ZERO* | *UFL* | ...

Annotations

A *Program* is a *Block*. A *Block* has three parts: a *set* of variable *Id*entifiers, a set of uniquely *Id*entified *Proc*edures (hence abstracted as a *map*), and a list of *Statements*. A *Proc*edure has a *parameter list* and a *Block*. The *parameter list* consists of pairs of formal parameter *Id*entifiers and their corresponding *LOC*ation or *PROC*edure type. A *Statement* is either an *El*elentary *Statement*, a subroutine *Call*, an *On-Unit*, a *Signal*, or a *Revert* statement. An *Exp*ression is either a variable or a formal parameter *Id*entification, a *Const*ant, or an *Infix* expression. An *Infix* expression has three parts: a left- and a right operand *Exp*ression, and an *Op*erator.

B. Static Context Condition Function Types

```
1  type:  is-wf-Progr:      Progr                → BOOL
2         is-wf-Block:      Block   → DICT → BOOL
3         is-wf-Procedure:  Id Proc → DICT → BOOL
4         is-wf-Stmt:       Stmt    → DICT → BOOL
5         is-wf-Expr:       Expr    → DICT → BOOL
```

C. Auxiliary Text Function Types

```
6  type:  e-tp:     Expr → DICT → Type
```

D. Static (Compile-time) Domains:

$$DICT = (Id \xrightarrow[m]{} Type)$$
$$Type = \underline{LOC} \mid \underline{PROC} \mid (Id\ (\underline{LOC} \mid \underline{PROC}))^*$$

E. Static Context Conditions:

```
1  is-wf-Progr(p) =
       is-wf-Block(p)[]
```

A *Progr*am *is* well-formed if its *Block is* well-formed.

```
2  is-wf-Block(mk-Block(ids,pm,stl))dict =
 .1    (let dict' = dict + ([id→LOC | id∈ids]
 .2                        ∪[id→s-pml(pm(id)) | id∈dompm]) in
3      (ids ∩ dompm = {})                                    ∧
4      (∀id ∈ dompm)(is-wf-Procedure(id,pm(id))dict')        ∧
5      (∀stmt ∈ rng stl) (is-wf-Stmt(stmt)idct'))
```

A *Block is* well-formed if:

2.3 No *Id*entifier is defined both as a variable and as a *Proc*edure name, and

2.4 all *Procedures* are *well-formed* in the lexicographically embracing scope, *dict'*, defined up till now, and

2-5 all *Statements* are *well-formed*, also in the context so far defined.

dict' (2.1-2) is the association (*DICT*) which to any variable name binds the fact, LOC, that it is a variable, and to any *Proc*edure name that it is a PROCedure -- in particular it then binds defined *Proc*edures to the formal *parameter list* with its type indications.

```
3  is-wf-Procedure(id,mk-Proc(pml,bl))dict =
 .1   (id ¬∈ {pml[i,1]|i∈indpml})                                    ∧
 .2   (∀i,j∈indpml)(s-Id(pml[i])=s-Id(pml[j]) ⊃ i=j                  ∧
 .3   (let dict' = dict +[s-Id(pml[i])→s-Tp(pml[i])|i∈indpml] in
 .4    is-wf-Block(bl)dict')
```

A *Procedure is well-formed*, in the context *dict*, if

3.1 the procedure name, *id*, is not also that of a formal *parameter* name, and

3.2 no two formal *parameters* have the same name, and

3.3 otherwise the body, *bl*, of the procedure *is well-formed* in the context, *dict'*, which to *dict* additionally binds formal *parameter Id*entifiers to their type indicator.

```
4  is-wf-Stmt(s)dict =
 .1   cases s:
 .2    (mk-Call(id,el)      → ((id ∈ domdict)                        ∧
 .3                            (∀e ∈ rngel)(is-wf-Expr(e)dict)       ∧
 .4                            ((dict(id) = PROC)                    ∨
 .5                            (LOC ≠ dict(id)) ⊃
 .6                            (let pml = dict(id);
 .7                            (lpml = lel)                          ∧
 .8                            (∀i ∈ indel)
 .9                              (e-tp(el[i])dict ≡ LOC ≡ s-Tp(pml[i])))))),
 .10   mk-On-Unit(cid,p)  → is-wf-Procedure(cid,p)dict,
 .11   mk-Signal(cid,idl) → (rngidl ∈ domdict),
 .12   mk-Revert(cid)     → true,
 .13   T                  → is-wf-El-Stmt(s)dict) /* not written */
```

The *well-formedness* of a *Statement*, in the context *dict*, depends on which kind of (what *case* of) *Statement* it is:

4.2-9 In a subroutine *Call* statement, which consists of a *Proc*edure *id*entifier and an *e*xpression *l*ist:
4.2 The *Proc*edure *id*entifier must be known,
4.4 and must be that of a *PROC*edure,
4.3 and all *e*xpressions of the actual argument *e*xpression *l*ist must be *well-formed*.
4.5 If the procedure *id*entifier is that of an actually defined, i.e. not formal, procedure,
4.6 then:
4.7 the length of the formal *para*meter *l*ist and the actual argument *e*xpression *l*ist must be the same,
4.8 and all corresponding (non-formal procedure)
4.9 argument *e*xpressions and formal *para*meter must have assignable value types.

4.10 In an *On-Unit* statement, which consists of a *c*ondition *id*entifier and a *p*rocedure body this combination, since it semantically corresponds very much to a procedure, must be a *well-formed Procedure* in the defining *dict*ionary context.

4.11 In a *Signal* statement, which consists of a *c*ondition *id*entifier and an argument *l*ist of *id*entifiers, these latter must be known in the *dict*ionary context -- it is not possible, due to the dynamic inheritance of associated *On-Unit*s, to check, as it was in 4.4-4.9, that the type of these arguments 'match' the type of the intended *On-Unit* 'procedure' *p*arameter *l*ist!

4.12 A *Revert* on any *c*ondition *id*entifier is always OK!

```
5   is-wf-Expr(e)dict =
.1    cases:
.2     (mk-Infix(e1,op,e2)→(is-wf-Expr(e1)dict ∧
.3                          is-wf-Expr(e2)dict ∧
.4                         (e-tp(e1)dict = LOC = e-tp(e2)dict)),
.5      mk-Const(i)       →true
.6      T                 →(e ∈ domdict))
```

F. Auxiliary Text Functions

```
7   e-tp(e)dict =
.1    cases e:
.2     (mk-Infix(e1,op,e2)→(op ∈ {EQ,NEQ}) → BOOL,
.3                          T               → LOC),
.4      mk-Const(i)        →LOC,
.5      T                  →dict(e))
```

G. Semantic Domains

$$
\begin{aligned}
STG &= LOC \xrightarrow[m]{} NUM \\
OE &= Cid \xrightarrow[m]{} FCT
\end{aligned}
$$

$$
\begin{aligned}
ENV &= Id \xrightarrow[m]{} DEN \\
DEN &= LOC \mid FCT \\
FCT &= DEN^* \to (OE \to (\Sigma \to \Sigma))
\end{aligned}
$$

$$
\begin{aligned}
LOC &\subset TOKEN \\
VAL &= NUM \mid BOOL
\end{aligned}
$$

$$
\Sigma = (\underset{\sim\sim}{STG} \xrightarrow[m]{} STG) \cup (\underline{ref}\ OE \xrightarrow[m]{} OE)
$$

Annotations:

A *STor*a*G*e is a finite domain *m*ap from *LOC*ations to assignable values, these are the retional *NUM*bers. An *O*n *E*stablishment is a finite domain *m*ap from *C*ondition *id*entifiers to the *FunCT*ions they denote.

To model the concept of scope we use the *ENV*ironment abstraction. An *ENV*ironment is a finite domain from *Prog*ram text *Id*entifiers to their *DEN*otations. The *DEN*otation of a *Prog*ram text *Id*entifier is either that of a *LOC*ation (if the *Id*entifier names a variable), or that of a *FunCT*ion (if it names a *Proc*edure). A *FunCT*ion is a (mathematical) function from a list of *DEN*otations (i.e. argument values) to functions from *O*n *E*stablishments to functions from states to states! that is: given an *On-Unit* or a *Proc*edure it denotes a function. In the case of the former the argument list is usually predefined, whereas in the latter it is programmer definable. Both denote *FunCT*ions which can be considered evaluated in the dynamic context of the defining *ENV*ironment,

but the calling *O*n *E*stablishment. Since they are all subroutines, no values are returned, but side-effects, i.e. state transformations, are effected.

A *LOC*ation is an otherwise un-analyzed elementary object. The auxiliary category, *VAL*, stands for the union of rational *NUM*ber and *BOOL*ean values.

The state space, Σ, omitting input/output, is a map from one *ST*ora*G*e reference to *ST*ora*G*es, and a multitude of zero, one, or more *ref*erences to *O*n *E*stablishments to *O*n *E*stablishments.

H. Global State Initialization:

dcl STG := [] *type STG*

I. Elaboration Function Types:

1	*type:*	*int-Progr:*	*Progr*	$\tilde{\rightarrow}$ (Σ $\tilde{\rightarrow}$ Σ)
2		*int-Block:*	*Block* $\tilde{\rightarrow}$ *ENV* $\tilde{\rightarrow}$ *OE*	$\tilde{\rightarrow}$ (Σ $\tilde{\rightarrow}$ Σ)
3		*int-Stl:*	*Stmt** $\tilde{\rightarrow}$ *ENV* $\tilde{\rightarrow}$ (*OE ref OE*)	$\tilde{\rightarrow}$ (Σ $\tilde{\rightarrow}$ Σ)
4		*int-Stmt:*	*Stmt* $\tilde{\rightarrow}$ *ENV* $\tilde{\rightarrow}$ (*OE ref OE*)	$\tilde{\rightarrow}$ (Σ $\tilde{\rightarrow}$ Σ)
5		*int-Call:*	*Call* $\tilde{\rightarrow}$ *ENV* $\tilde{\rightarrow}$ *OE*	$\tilde{\rightarrow}$ (Σ $\tilde{\rightarrow}$ Σ)
6		*eval-Proc:*	*Proc* $\tilde{\rightarrow}$ *ENV* $\tilde{\rightarrow}$	$\tilde{\rightarrow}$ *FCT*
7		*eval-arg:*	*Expr* $\tilde{\rightarrow}$ *ENV* $\tilde{\rightarrow}$ *OE*	$\tilde{\rightarrow}$ (Σ $\tilde{\rightarrow}$ Σ (*FCT* | *LOC*))
8		*eval-Expr:*	*Expr* $\tilde{\rightarrow}$ *ENV* $\tilde{\rightarrow}$ *OE*	$\tilde{\rightarrow}$ (Σ $\tilde{\rightarrow}$ Σ *VAL*)

J. Auxiliary Function Types

9	*type:*	*get-loc:*		→(Σ → Σ *LOC*)
10		*free-locs:*	*Id-set ENV*	$\tilde{\rightarrow}$(Σ → Σ)
11		*type-chk:*	*DEN* (Id (LOC|PROC))**	→*BOOL*
12		*free-dummy-locs:*	*DEN* Expr**	→(Σ → Σ)

K. Semantic Elaboration Function Definitions

```
1  int-Progr(p) =
       int-Block(p)([])([])
```

To *int*erpret a *Progr*am is the same as *int*erpreting the *Block* it is in an empty *ENV*ironment and an empty *O*n-*E*stablishment.

```
2  int-Block(mk-Block(ids,pm,stl,stl))(env)(boe) =
 .1    (let env' : env + ([id → get-loc() | i∈ids]
 .2                       ∪[id → eval-proc(pm(id))(env') | id∈dompm]);
 .3     dcl loe := boe;
 .4     int-stl(stl)(env)((boe,loe));
 .5     free-locs(ids,env'))
```

*Int*erpreting a *Block* whose locally defined variables are represented by *ids*, locally defined procedures by *pm*, and statement list by *stl*, is:

2.1 first to associate with each variable identifier a fresh *LOC*ation, and

2.2 with each procedure identifier the *Fun*C*T*ion it is, the latter in the, thus circularly defined, defining *env*ironment (').

2.3 Then to establish a *l*ocal *o*n *e*stablishment which inherits the value of the embracing *b*locks' *o*n *e*stablishment,

2.4 whereupon actual execution, after these prologue actions, can take place of the *s*tatement *l*ist.

2.5 Storage allocated in 2.1 is freed here.

```
9  get-loc() =
 .1    (let l ∈ LOC be s.t. l ~∈ dom(c STG);
 .2     STG := c STG ∪ [l→undefined];
 .3     return(l))
```

This function allocates and 'initializes' to *undefined*, *ST*ora*G*e on a per-*l*ocation basis.

```
10  free-locs(ids,env) =
        STG := cSTG\{env(id) | id∈ids}
```

This is block termination STorage freeing epilogue action.

```
7   eval-proc(mk-Proc(pml,bl))(env) =
.1      (let f(al)(oe) =
.2          (if ~type-match(al,pml)
.3              then error
.4              else (let env'=env+[s-Id(pml[i])→al[i]|i∈indal];
.5                    int-block(bl)(env')(oe))) in
        f)
```

```
12  type-match(al,pml) =
.1      ((lal = lpml) ∧
.2       (∀i∈indal)(is-LOC(al[i]) ≡ s-Tp(pml[i]) = LOC ))
```

The meaning of a *Proc*edure is the *f*unction it denotes. This *f*unction is implicitly defined by what happens if it is *Call*ed. Then:

7.4 the defining *env*ironment is augmented with the bindings between formal *p*arameter *l*ist *id*entifiers and the passed actual *a*rgument *l*ist *DEN*otations,

7.5 whereupon the *bl*ock of the *Proc*edure (the 'body') is elaborated in the callling state, but essentially the defining *env*ironment! Since the calling state involves the *o*n-*e*stablishment, and since each *Block* potentially defines its own 'copy' which may be dynamically updated, one needs to pass the value of the current blocks' local *o*n *e*stablishment to the invocation of the *Proc*edure denotation; hence, in line :

7.1 the functional dependency on the calling states' *o*n *e*stablishment.

7.2 The type-check is statically decidable for ordinary procedures, but not for *On-Unit* procedures.

```
3   int-Stl(stl)(env)(oep) =
.1      for i=1 do len stl do int-Stmt(stl[i])(env)(oep)
```

To *int*erpret a *l*ist is the same as *int*erpreting each of its *S*tate*m*ents in the order listed.

```
4    int-Stmt(s)(env)((boev,loer)) =
.1       cases s:
2         (mk-Call(id,el)→
3                (let al : <eval-arg(el[i])(env)(cloer)|i∈indel>;
4                 let f = env(id) in
5                 f(al)(cloer);
6                 free-dummy-locs(al,el)),
7          mk-On-Unit(cid,p)→
8                (let f = eval-proc(p)(env) in
9                 loer := cloer + [cid→f]),
10         mk-Signal(cid,idl)→
11               (let al = <env(idl[i])|i∈indidl> in
12                if cid ∈ dom(cloer)
13                  then (let f : (cloer)(cid);
14                              f(al)(cloer))
15                  else error),
16         mk-Revert(cid)→
17               if cid ~∈ domboev
18                  then loer := cloer\{cid}
19                  else loer := cloer + [cid→boev(cid)],
20         T      →int-El-Stmt(s)(env)(cloer))
```

4. To *int*erpret a *S*tate*m*ent is a function of what statement it is:

4.2-6 *int*erpreting a subroutine *Call* statement consists of the following sequence of actions:

4.3 Each *e*xpression of the *Arg*ument *l*ist is *eval*uated,

4.4 and the procedure (i.e. -*id*entifier) denotation retrieved from the scope,

4.5 whereupon it is being applied to the evaluated *a*rgument *l*ist and the value (i.e. *c*ontents) of the current, *l*ocal *o*n *e*stablishment.

4.6 The locations allocated during *Arg*ument *eval*uation -- see 13 below -- are freed.

4.7-9 *int*erpreting an *On-Unit* results in the update of the *l*ocal *o*n *e*stablishment (known by *r*eference) with the function denoted by the *On-Unit p*rocedure body in the position known as *cid*, i.e. for that on *c*ondition *id*entifier.

4.10-15 *int*erpreting a *Signal* statement is like *Call*ing a subroutine, but there are some significant differences.

4.11 First all *e*xpressions of the argument *l*ist must all be *id*entifiers, whereby their denotation can be extracted right from the calling (i.e. *Signal*ling) *env*ironment.

4.12 If the designated (*cid*) *On-Unit* has not been defined (by some *On-Unit* of the embracing scope) then

4.15 an *error* situation has arisen,

4.13 otherwise the *f*unction denoted by the specifically *Signal*led (i.e. *id*entified) *c*ondition *On-Unit*

4.14 is applied to the *a*rgument *l*ist concocted in line 4.11. Observe that no *env*ironment is supplied, but that the *c*ontents of the current, the *l*ocal *o*n *e*stablishment is. The former is 'embedded' in the *f*unction denotation, the latter part of its functionality.

4.16-19 *int*erpreting a *Revert* statement has the effect of letting the current, the *l*ocal *o*n *e*stablishment henceforth associate the denotation of the *c*ondition *id*entifier with its value in the *o*n *e*stablishment of the embracing *b*lock.

... etcetera.

```
8  eval-arg(e)(env)(loev) =
 .1    cases e:
 .2     (mk-Infix(,,) → (let v : eval-expr(e)(env)(loev),
 .3                           l ∈ LOC be s.t. l~∈ dom(cSTG);
 .4                      STG := cSTG ∪ [l→v];
 .5                      return(l)),
 .6      mk-Const(i)  → (let l ∈ LOC be s.t. l ~∈ dom(cSTG);
 .7                      STG := cSTG ∪ [l→i];
 .8                      return(l)),
 .9      T            → return(env(e)))
```

*eval*uating a subroutine *Call arg*ument proceeds according to the following basic scheme. The exemplified language has *Call*-by-*LOC*ation for objects other than procedures. Thus:

8.2.5 *Infix arg*ument *ex*pressions are *eval*uated, a fresh STorage pseudo *l*ocation is 'fetched', and STorage initialized to the argument expression *v*alue, with the new *l*ocation being *return*ed.

8.6-8 Likewise for *Const*ant expressions.

8.9 All other expressions, i.e. variable- and *Proc*edure *id*entifiers result directly in their denotation being retrieved from the scope.

Thus *Proc*edure denotations is passed by-worth!

```
13 free-dummy-locs(al,el) =
 .1      (let locs = {al[i] | ~is-Id(el[i]) ∧ i∈indel};
 .2       STG := cSTG\locs)
```

```
9  eval-expr(e)(env)(oe) =
 .1    cases e:
 .2     (mk-Infix(e1,op,e2)
 .3        → (let v1 : eval-expr(e1)(env)(oe),
 .4               v2 : eval-expr(e2)(env)(oe);
 .5           cases op:
 .6            (ADD → ((v1+v2 ≥ 2↑64)
 .7                       → (if OFL ∈ domoe
 .8                             then (dcl r := v1+v2;
 .9                                   (oe(OFL))(r);
 .10                                  return(cr))
 .11                            else return(2↑64)),
 .12                  (v1+v2 < -2↑64)
 .13                       → (if UFL ∈ domoe
 .14                             ...),
 .15                   T   → return(v1+v2)),
 .16             SUB → ...
 .17             ...
 .18             EQ  → return(v1=v2)
 .19             ...)),
 .20     mk-Const(i)
 .21       → return(i),
 .22     T → (cSTG)(env(id)))
```

We concentrate on lines 9.6-9.11.

If evaluation of an arithmetic expression leads to overflow (9.6), then either of two situations occur.

Either there is defined, by the programmer, an *On-Unit*, in the current *On Establishment*, for handling *OverFLow*, and then this unit is called (9.9) passing to it -- as a hypothetical example -- reference to a meta-variable initialized to the overflow value. The value of the expression becomes the contents of this variable (9.10) after execution of the defined *On Unit* procedure (9.9). Thus we expect the programmer to define the *OFL* on-units with one formal parameter of type *LOC*ation. We do not show a static test for this -- but could have.

Or: the programmer has not defined an appropriate *OFL On-Unit* in which case the SYSTEM action is to return, say the maximum arithmetic-value (9.11)!

Comment

From the definition we can informally derive the following informal, technical english, users programming reference manual-like, description of the On-Condition concept.

*On-Unit*s are like assignment statements. The target reference is one, of a limited variety, of condition codes (*cid*). The right-hand side expression is restricted to be a procedure (4.7-9).

To *Signal* is to invoke the most recently 'assigned' *On-Unit* of the name (*cid*) signalled. Thus a *Signal* is like a *Call*.

To *Revert* is to locally re-assign the *On-Unit* most recently 'assigned' in the immediately, dynamically containing block.

Further: To each block activation we let there correspond an association of *cid*s to *On-Unit* procedure values called an On-Establishment (2.3). A block activation inherits the value of this association in the invoking block (respectively calling procedure) (2.3 from 4.5 + 7.5). 'Falling' back to the interpretation of an invoking block brings us back to the on-establishment of this latter block current when this block invoked the block just terminated.

Finally: Procedures are elaborated in the defining environment (2.2 → 7.4-5), but in the calling on-establishment (4.5 → 7.1-5).

Discussion

We shall only discuss the local/global state modeling chosen in our conceptualization of the source language On Condition-, respectively Variable constructs. Our first example illustrated that model components, like the state (Σ), which are transformable by any syntactic construct, can indeed be an explicit parameter to functions elaborating these constructs, provided, of course, that the possibly changed state is likewise explicitly yielded as part, or all, of the result. This is the rule followed in all of the Oxford models; many examples in [Bjørner 78b] also exemplified this specification style. For block-structured imperative programming languages it soon, however, becomes rather cumbersome to write, and read, all these explicit passings and returns of such all-pervasive, components.

Hence itwas decided, as a concession to readability, as well as to engineering, to permit variables in the meta-language. The point is now to use variables sparingly, and to have their introduction, the fact whether they are local or global, and their manipulation, reflect the very essence of the concept they are intended to model. Therefore:

Since *source-language* variables, declared at any (source-language-) block- & procedure level, can be changed at any other, "inner" and "outer" level, the storage component of the state was chosen to be modeled by a single, global meta-variable. (That *sl*-variables can be updated on levels outside their scope is due to the by-*LOC*ation parameter passing capability.)

And: since *On-Units* correspond to assignments to variables (names in *Cid*) of type procedure- (or, as in PL/I, entry-) value, the model component (*on-establishment*), keeping track of current *Cid* to procedure value associations, was also chosen to be a meta-variable. Further: since such 'assignments' in one block (*loer*) are not to disturb the associations recorded in any containing block (*boe*), we introduce one such meta-variable, *loe*, per block activation. To shield the *boe*, which is needed in a directly contained block due to *Reverts*, it is passed by value, i.e. its content (4.5 → 7.5 → 2.0 → 2.4); whereas the *local oe* is passed by *reference* (*loer* ∈ *ref OE*) (2.3 → 2.4 → 4.0 → 4.9,4.18, 4.19).

Modeling on-establishments by locally declared meta-language variables shifts the burden of 'stacking' embracing on-establishments from the definer, and of understanding these usually rather mechanical descriptions away from the reader, and onto the meta-language: its semantics, respectively the readers understanding of, in this case, recursion.

ACKNOWLEDGEMENTS:

The author is very pleased to express, once more, his deep gratitude for his former colleagues at the IBM Vienna Laboratory, and to Mrs. Annie Rasmussen for her virtuoso juggling of eight distinct IBM "golf ball" type fonts.