Program Specification and Verification in VDM*

C.B.Jones

November 28, 1986

UMCS 86-10-5

Program Specification and Verification in VDM*

C.B. Jones

November 28, 1986

Abstract

Formal methods employ mathematical notation in writing specifications and use mathematical reasoning in justifying designs with respect to such specifications. One avenue of formal methods research is known as the Vienna Development Method. *VDM* has been used on programming language and non-language applications. In this paper, programming languages and their compilers are ignored; the focus is on the specification and verification of programs.

VDM emphasizes the model-oriented approach to the specification of data. The reification of abstract objects to representations gives rise to proof obligations; one such set which has wide applicability assumes an increase in implementation bias during the design process. The incompleteness of this approach and an alternative set of rules are discussed.

The decision to show the input/output relation by post-conditions of two states is also a feature of *VDM*. In early publications, the proof obligations which support decomposition were poorly worked out; those presented below are as convenient to use as the original "Hoare-logic". Recent work on a logic (which is tailored to partial functions) is also described.

1 Introduction

The term "Formal Methods" applies to the use of mathematical notation in the specification, and the use of such specifications as a basis for the verified design, of computer systems. There are several more-or-less distinct approaches in use. The approach which matches normal software development practice most closely is to specify a task in a specification language; to record design decisions in languages which are close to the final implementation language; and to then generate and discharge consequent proof obligations.

A second approach (see, for example, [12]) is to make the starting point of the development a very clear description which can—possibly with very poor performance—be executed. Design steps then consist of transforming such a description into a "program" of acceptable efficiency.

The third general approach (see [15]) also begins with a specification but then provides a constructive proof of the existence of a result; from this proof a program can be extracted.

The so-called *Vienna Development Method* (*VDM*) subscribes to the first of these approaches. In [32] specifications are written using pre-/post-conditions which are truth-valued functions over state-like objects. Such states are models defined in terms of basic objects like sets. Design proceeds by *data reification* and *operation decomposition*.

Steps of data reification make the transition from abstract objects to objects which are representable in the chosen implementation language. Operation decomposition is the process of realizing implicit specifications by statements written in the programming language. For significant systems, several steps of both sorts of design step might be required.

*This is an expanded version of the paper which will appear in the proceedings of the 1986 Marktoberdorf Summer School: "Logic of Programming and Calculi of Discrete Design", (ed.) M. Broy, Springer-Verlag. The main change is the insertion of the proofs in Appendix A.

Either sort of design step gives rise to *proof obligations*. These are sequents which must be true for the design step to satisfy its specification. The required proofs can be conducted at an appropriate level of formality.

In the extreme, these proof obligations could be compared to the output of *verification condition generators* (VCG) (cf. [9]). The VCG approach has received considerable criticism because of the difficulty of relating the created logical expressions to the original programming task. This author's own experience both with hand proofs and with Jim King's *EFFIGY* system (cf. [18]) has confirmed the validity of this criticism. The attempt to prove a complete program correct using VCG's is like trying to solve equations in large numbers of variables—unfortunately, failure to find a proof corresponds to the lack of a solution and the ensuing hunt for alternative assertions is very tedious. How does *VDM* avoid this problem? The approach is to use the steps of development themselves as a way of decomposing the correctness argument. Well-chosen steps provide an informal proof outline of the type used by mathematicians. This reduces the need for formal proof. If it is decided to construct a formal proof, such a proof is likely to be relatively simple. Perhaps most importantly, any errors are relatively easy to locate. An essential property is "compositionality" in a development method. It has been argued elsewhere (e.g. [31]) that this approach can increase the productivity of the development process by locating—soon after insertion—any errors which are made in the early stages of design.

This paper reviews some recent and on-going research relevant to formal methods like *VDM*. Sections 2 and 3 review, respectively, work on data reification and operation decomposition. The need to establish a logic which recognises the rôle of partial functions is reviewed in Section 4. A separate paper ([35]) discusses the form of support system which might help in the use of formal methods. The general direction of this work can be seen in [13]. (The *VDM* approach to language specification is not discussed in this paper. Interested readers are referred to [6].)

2 Data Reification

In order to achieve full advantage from the application of formal methods, it is necessary to apply them to the early stages of development. Clearly, this implies the construction of formal specifications. After that, one must ask: what activities are most common in the early (high-level) design stages? Typically the choice of data representations is made before detailed algorithm design. Thus *VDM* tends to put more emphasis on proof of data reification than on operation decomposition[1]. This section reviews the most straightforward rules for the justification of design steps of data reification, the shortcomings of these rules, and a new set of rules which are—in some sense—complete.

A specification in *VDM* normally consists of a set of states and a set of operations which rely on, and transform, these states. Operation specifications are discussed below; initially, attention is focussed on the objects which comprise the states.

Many computing applications need some form of access to data via a *Key*. A convenient *map* object is used in *VDM* which makes it possible to define:

$$Keyed = \text{map } Key \text{ to } Data$$

with an initial object corresponding to the empty map[2]:

$$m_0 = \{\}$$

[1] It is an accident of history—reiterated, as Kuhn predicts, in many text books—that operation decomposition proofs were studied first: see, for example, [42].

[2] The identification of a set of initial states in [32] differs from [30] where initialising operations were used.

Such an abstraction is very convenient in the specification precisely because it hides all of the implementation problem. Clearly, the design process must choose—and justify—a representation.

One possible way of storing large volumes of keyed data is in a binary tree. Such a set of trees can be described by[3]:

$$Bintree = [Binnode]$$

$$\begin{array}{lll} Binnode :: & lt & : Bintree \\ & k & : Key \\ & d & : Data \\ & rt & : Bintree \end{array}$$

where

$$inv\text{-}Binnode(mk\text{-}Binnode(lt,k,d,rt)) \triangleq (\forall lk \in collkeys(lt) \cdot lk < k) \wedge (\forall rk \in collkeys(rt) \cdot k < rk)$$

$$collkeys : Bintree \rightarrow \text{set of } Key$$

$$\begin{array}{lll} collkeys(t) \triangleq & \text{cases } t \text{ of} & \\ & \text{nil} & \rightarrow \{\} \\ & mk\text{-}Binnode(lt,k,d,rt) & \rightarrow collkeys(lt) \cup \{k\} \cup collkeys(rt) \\ & \text{end} & \end{array}$$

The set of objects is considered to be restricted by the invariant[4]—thus:

$$Binnode = \{mk\text{-}Binnode(lt,k,d,rt) \mid lt,rt \in Bintree \wedge k \in Key \wedge d \in Data \wedge inv\text{-}Binnode(mk\text{-}Binnode(lt,k,d,rt))\}$$

The initial object corresponds to the empty tree:

$$t_0 = \text{nil}$$

The representation, *Bintree*, must be related to the abstraction *Keyed*. In early work in Vienna this was normally done by a relation or (cf. [38]) by building up a combined state with an invariant to control the relationship to the *ghost variables*. During the late 60's, it was realized that a special case arose very often in design. It was noticed (cf. [34]) that a one-to-many relation often existed between elements of the abstraction and those of the representation. This was no accident. It is desirable to make states of specifications as abstract as possible; the structure of the implementation language (or machine) forces the introduction of extra information and redundancy; it is, therefore, very common that a one-to-many relationship arises[5]. Precisely this situation holds here. There are many possible tree representations of any (non-trivial) map

[3] The optional brackets define:

$$Bintree = Binnode \cup \{\text{nil}\}$$

The "::" notation defines *Binnode* in terms of a constructor or projection function:

$$mk\text{-}Binnode: Bintree \times Key \times Data \times Bintree \rightarrow Binnode$$

[4] This, more central, rôle for invariants is also a change from [30].

[5] In fact, this discussion gives rise to the notion of *bias* discussed in [29] where the avoidance of redundancy is taken as a test for the acceptability of a set of states to be used as the basis of a specification.

object. Both [24] and [34] relate the set of abstractions to their representations by a function from the latter to the former. Here they are called (following [27]) *retrieve functions* because they get back the abstract values from the representation details. For the example in hand:

$$retrm : Bintree \to Keyed$$

$$retrm(t) \triangleq$$
cases t of
nil $\to \{\}$
$mk\text{-}Binnode(lt, k, d, rt) \to retrm(lt) \cup \{k \mapsto d\} \cup retrm(rt)$
end

In the set of rules used most commonly in *VDM*, such retrieve functions must be total. That this property is satisfied by *retrm* follows from the invariant which ensures that the domains of the maps to be united are disjoint. Another property required of the representation (strictly—with respect to the retrieve function) is *adequacy*: there must be at least one representation for each abstract element. For the case in hand:

$$\forall m \in Keyed \cdot \exists t \in Bintree \cdot retrm(t) = m$$

It is straightforward to prove this by induction on the domain of m. It is, in fact, worth providing a function which inserts *Key*/*Data* pairs into a tree (cf. *insb* below) and use this. In practice, it would be worth defining a number of functions and developing the *theory* of the data type. For example:

$$\forall t \in Bintree \cdot \mathrm{dom}\ retrm(t) = collkeys(t)$$

can be proved by structural induction on *Bintree*. One of the advantages of this set of proof rules is that they do isolate useful proof obligations about the state alone and, in practice, these proofs are a very useful check on a representation before proceeding to look at the individual operations. It is, however, also necessary to consider the operations. The initial states are trivially related:

$$retrm(t_0) = m_0$$

On *Keyed*, the insert operation is specified trivially:

$$INSERT\ (k: Key, d: Data)$$
ext wr m : $Keyed$
pre $k \notin \mathrm{dom}\ m$
post $m = \overleftarrow{m} \cup \{k \mapsto d\}$

Such a specification[6] defines a partial and possibly non-deterministic state transformation. The pre-condition defines the set of states over which the implementor must make the operation terminate and yield a result which, together with the input state (variables decorated with backwards pointing hooks) must satisfy the post-condition[7].

[6] The rôle of the side effect on the external variables is emphasized in *VDM* by the presentation of operation specifications. The meaning is:

$$INSERT: Key \times Data \times Keyed \to Keyed$$
$$\forall k \in Key, d \in Data, \overline{m} \in Keyed \cdot k \notin \mathrm{dom}\ \overline{m} \wedge INSERT(k, d, \overline{m}) = m \Rightarrow m = \overline{m} \cup \{k \mapsto d\}$$

[7] During the Summer School, Bernard von Stengel pointed out that more expressive power could be achieved by adding the requirement that, if termination did occur even for states not satisfying the pre-condition, the results should still be constrained by the post-condition.

Such a specification should satisfy the *implementability* proof obligation, in this case:

$$\forall \overleftarrow{m} \in Keyed, k \in Key, d \in Data \cdot pre\text{-}INSERT(k, d, \overleftarrow{m}) \Rightarrow \exists m \in Keyed \cdot post\text{-}INSERT(k, d, \overleftarrow{m}, m)$$

The corresponding operation on *Bintree* is defined:

$INSERT_B$ $(k: Key, d: Data)$
ext wr t : $Bintree$
pre $k \notin collkeys(t)$
post $t = insb(k, d, \overleftarrow{t})$

The auxiliary function is defined:

$insb$ $(k: Key, d: Data, t: Bintree)$ $r: Bintree$
pre $k \notin collkeys(t)$

$insb(k, d, t) \triangleq$
cases t of
nil $\rightarrow$ $mk\text{-}Binnode(\text{nil}, k, d, \text{nil})$
$mk\text{-}Binnode(lt, mk, md, rt) \rightarrow$
if $k < mk$
then $mk\text{-}Binnode(insb(k, d, lt), mk, md, rt)$
else $mk\text{-}Binnode(lt, mk, md, insb(k, d, rt))$
end

The relevant proof obligations for this operation are:

$$\forall t \in Bintree \cdot pre\text{-}INSERT(k, d, retrm(t)) \Rightarrow pre\text{-}INSERT_B(k, d, t)$$

$$\forall \overleftarrow{t}, t \in Bintree \cdot pre\text{-}INSERT(k, d, retrm(\overleftarrow{t})) \wedge post\text{-}INSERT_B(k, d, \overleftarrow{t}, t) \Rightarrow post\text{-}INSERT(k, d, retrm(\overleftarrow{t}), retrm(t))$$

The first of these requires that the domain of the operation on the representation is large enough; the second requires that the transition on the representation—when viewed under the retrieve function—nowhere contradicts the specification on the abstract states. Proofs of these results are straightforward—the first appeals to the lemma mentioned above.

These proof rules are more general than required for this situation but it should be remembered that the post-condition of $INSERT_B$ could have been:

$$retrm(t) = retrm(\overleftarrow{t}) \cup \{k \mapsto d\}$$

which is non-deterministic[8]. Furthermore, at the next stage of development, the operations on *Bintree* would play the part of the specification so the rules need to cater with partial, non-deterministic operations in both the specification and the representation[9].

The interest here, however, focuses on the *incompleteness* of the above rules. It was known at the time the rules were published in [30] that there were valid steps of development which

[8] This is illustrative of the way in which non-determinacy is most useful in the design process: with a deterministic specification, and the intention to design a deterministic program, non-determinism can be used to structure the design decisions (e.g. the more general $post\text{-}INSERT_B$ could be used to reflect the fact that the precise tree balancing algorithm has not been chosen at this design step).

[9] A second step of development of the abstract *Bintree* objects onto Pascal records and pointers is given in [32].

they would not support. In particular, it was obvious that any step of development which reversed the normal one-to-many relationship between abstraction and representation (e.g. to have *Bintree* in the specification and *Keyed* in the design) could not be justified since a retrieve function could not be found. This restriction was viewed as a virtue in so far as it tended to minimize the danger of biased specifications. Lockwood Morris also pointed out a technical problem in the need to tighten invariants so as to fulfil the requirements on retrieve functions.

What has become apparent more recently is that there are perfectly good specifications for which valid implementations cannot be justified by the above set of rules. The essence of the problem is explained below—an example is presented first.

In her work (cf. [40]) on the specification of GKS, Lynn Marshall uncovered a situation where more information was needed in the specification state than in that of valid implementations: there is a need to place, in the state, information required only to express non-determinacy; an implementation which is constrained to a deterministic answer needs less information. A simple example (due to Ib Sørensen) of this situation results from the specification:

$$s_0 = \{\}$$

ARB () r:N
ext wr s : set of N
pre true
post $r \notin \overleftarrow{s} \wedge s = \overleftarrow{s} \cup \{r\}$

This requires that each invocation of the operation ARB returns a result which it has never returned before. The state is initialized to the empty set and ARB adds each element which it returns. This specification is (unboundedly!) non-deterministic. An implementation which simply returns the "next" natural number on each invocation violates none of the specified requirements—thus:

$$n_0 = 0$$

ARB_n () r:N
ext wr n : N
pre true
post $r = \overleftarrow{n} \wedge n = \overleftarrow{n} + 1$

Intuitively ARB_n is correct with respect to the specification ARB: it has the same domain and yields answers which do not contradict the specification. This notion of *satisfaction* can be formalised. An operation is defined by a pair:

$$(S, R)$$

where S is the set of states over which termination is guaranteed and R is the relation expressing the input/output relation[10]. There is a requirement that:

$$S \subseteq \text{dom } R$$

For a specification, the corresponding semantic object is[11]:

$$(\{\sigma \mid pre\text{-}OP(\sigma)\}, \{(\overleftarrow{\sigma}, \sigma) \mid post\text{-}OP(\overleftarrow{\sigma}, \sigma)\})$$

[10] There is not a requirement for bounded non-determinacy.

[11] The requirement corresponding to $S \subseteq \text{dom } R$ is the *implementability* proof obligation.

The formal notion of satisfaction is defined[12]:

$$(S_1, R_1) \text{ sat } (S_2, R_2) \triangleq S_2 \subseteq S_1 \wedge S_2 \lhd R_1 \subseteq R_2$$

That is (S_1, R_1) satisfies (S_2, R_2) iff the termination domain S_1 is at least as large as S_2 and the meaning relation R_1—restricted to the required termination set—nowhere contradicts R_2. Notice that sat is a partial order. Thus, with appropriate use of the retrieve function, it can be seen that ARB_n satisfies ARB but it cannot be proved by the rules used above for the development of *Bintree*. Were this the only sort of counter-example, it would be possible to introduce special steps of development for the situation where a reduction in non-determinacy reduces the complexity of states. There is, unfortunately, another class of counter-examples. The root of the further weakness discovered in the proof obligations used above for *Bintree* is that they were formulated around the aim of showing that each individual operation on the representation satisfied the corresponding abstract operation. This property is not necessary since it is the external behaviour of the collection of operations which needs to be preserved. Once this point is recognised it is a simple matter to generate further counter-examples. (It is, of course, the case that these rules can be—straightforwardly—proved consistent with sat .)

The problems discussed above have been overcome by a rule which is based on a relation between the abstract and representation state spaces:

$$\_ \sqsubseteq \_ : Rep \times Abs \to B$$

There are no proof obligations such as adequacy on $\sqsubseteq$; those for the operations are:

$$\forall a \in Abs, r \in Rep \cdot r \sqsubseteq a \wedge pre_A(a) \Rightarrow pre_R(r)$$

$$\forall \overleftarrow{a} \in Abs, \overleftarrow{r}, r \in Rep \cdot \overleftarrow{r} \sqsubseteq \overleftarrow{a} \wedge post_R(\overleftarrow{r}, r) \Rightarrow \exists a \in Abs \cdot post_A(\overleftarrow{a}, a) \wedge r \sqsubseteq a$$

It is possible to avoid the need for special rules on the initial states if an initialization operation is included which behaves like a (possibly non-deterministic) constant. The other omission from these rules is an assumption in the result rule of $pre_A(\overleftarrow{a})$. To see that this is not required, it is necessary to realize that the simulation relation ($\sqsubseteq$) need only involve those elements of *Abs* which are reachable by the operations[13].

Basically similar forms of this rule were found independently by Tobias Nipkow and researchers at the Programming Research Group in Oxford. This work led to joint discussions and is reported in [44], [43], [20] and [21].

The rule given here is certainly more powerful than that used above on the *Bintree* example. It is natural to question whether it is complete. Loosely what this amounts to is the question whether any correct data reification can be verified by the rule. But what is the independent notion of "correct"? If the behaviour of sequences of the operations is to be used in defining this notion, one approach is to define a language for combining operations. Issues such as the presence or absence of (angelic or otherwise) non-deterministic statements must be resolved. This linguistic approach is taken in [43] and the rule proven to be complete under suitable assumptions.

This then leaves the pedagogic question of which rule to teach (first). It is argued in [1] that their version of the above rule is easier to use; [32] takes a different view. It can be claimed that the avoidance of bias (cf. Section 9.1 of [32]) is an important objective in specifications and that this, combined with the use of proof obligations which avoid the need for existential

[12] $S \lhd R \triangleq \{(x, x') \in R \mid x \in S\}$

[13] The insertion of $pre_A\ (\overleftarrow{a})$ does however make the link to the sat relation above more obvious. This link can be seen simply by substituting the identity relation for $\sqsubseteq$; in the form present here, the identity must be suitably restricted.

quantifiers, justifies the use of the older ([30]) rule whenever it is applicable. Even the need to tighten invariants can be seen as an advantage in that it makes both the task of changing a specification safer and the range of potential representations clearer. A final argument in favour of the more restrictive rule is that the isolation of the adequacy proof obligation makes it possible to conduct a significant part of the verification work once (on the state) rather than delaying it to the (many) operation proofs. Thus [32] postpones the more general rule to Chapter 9.

3 Decomposition Rules with Input/Output Relations

The need to have post-conditions which relate final to initial states[14] is illustrated above by the choice of examples. It is such a natural way of thinking about the specification of a system that it comes as a surprise to notice that much of the work on program proofs uses post-conditions of the final state alone (cf. [23], [14], [17], [4] but not [25]). Since most computer programs are clearly written to transform a state, some way of describing the input/output relation must be found. One way of achieving this (cf. [17]) is to store the initial values of variables in the state. This would not always be acceptable in the final program but such variables can be marked as "ghost variables" and dropped once they have played their part in the proof. Strictly, this approach still needs a way of expressing the fact that these special variables cannot be changed during execution (cf. glocon, glovar, etc. in [14]). A second approach to the gap left by post-conditions having no direct way of referring to the initial state is to use free variables[15]. Unfortunately, the fact that such variables span more than one formula makes their treatment difficult.

A third approach to the gap is to govern the relationship by a free predicate symbol—thus, using the weakest precondition of [14]:

$$wp(GCD, p(x)) = p(gcd(x, y))$$

None of these methods is entirely satisfactory and this section presents proof obligations which directly handle post-conditions of two states[16].

Another issue which divides some specification methods from *VDM* is its separation of the pre-condition. It should be clear that (a form of) the pre-condition could be conjoined to the post-condition so as to yield a single predicate which comprises the whole specification of an operation. In *Z*, for example, there is one predicate to define an operation (cf. [19]). The decision to separate the pre- and post-conditions in *VDM*'s operation specification was initially motivated by purely pragmatic considerations. It does appear to be good discipline to make a distinction between the assumptions (that an implementor is invited to make) and the requirement that the (output of the created) program must satisfy. In many industrial specifications with which this author has had contact, the requirement was better thought out than the assumption. Thus, it seems wise to put some focus on the pre-condition. It so happens that the separation of the pre-condition has a number of formal advantages including the rôle that pre-conditions play in the various data reification proof obligations.

What form are the proof rules for operation decomposition to have when post-conditions do relate final to initial states? Unfortunately, the proof obligations given in [30] are rather heavy. They do succeed in splitting the task of checking a decomposition step into small, separate,

[14] It must be conceded that the term "post-condition" is not well-chosen; its wide use, however, makes it preferable to the introduction of a new term like "input/output relation".

[15] These free variables can be made more apparent by selecting some special fount.

[16] The method used here should be compared with [49] where pre-conditions are also truth-valued functions of pairs of states.

proof steps. But the rules are certainly not memorable. The suggestions made by Peter Aczel (cf. [3]), however, have led to rules which bear comparison with those in [23].

The proof rules, whose application gives rise to proof obligations, are explained here together with a style in which programs can be annotated with their correctness arguments. Ways in which the proof ideas can be used in the development of programs are discussed at the end of this section.

This section presents proof rules only for some simple programming language constructs. The general form of these rules is similar to those of logic. Here, the conditions under which a rule can be applied require that certain properties hold for sub-operations and the conclusions are that (other) properties hold for combinations of the sub-operations[17]. The proof rules facilitate proofs that pieces of program satisfy specifications. Thus it can be shown that:

if $i < 0$ then $i, j := -i, -j$ else skip

satisfies the specification:

MAKEPOS
ext wr i: Z, wr j: Z
pre true
post $0 \leq i \wedge i * j = \overleftarrow{i} * \overleftarrow{j}$

For small examples, it is convenient to record the specification and its implementation together, thus:

MAKEPOS
ext wr i: Z, wr j: Z
pre true
  if $i < 0$ then $i, j := -i, -j$ else skip
post $0 \leq i \wedge i * j = \overleftarrow{i} * \overleftarrow{j}$

Such an annotated program is written when the code has been shown to satisfy the specification. The name of the operation and the externals line are sometimes omitted when they are clear from context.

A natural extension of this style is to write specifications for sub-operations—rather than their code; see Figure 1. This can be read as saying that the composition of two sub-operations *MAKEPOS* and *POSMULT* would satisfy the specification of *MULT*. The sub-operations are not (yet) coded—rather, their specifications are given[18]. The proof rule for sequential execution is discussed below.

[17] It is, in part, the form of the proof rules used here which prompted the decision to mark initial values with a hook (rather than priming the final values) in post-conditions.

[18] A design can be presented as a combination of (specified) sub-problems. A *compositional* development method permits the verification of a design in terms of the specification of its (syntactic) sub-programs. Thus, one step of development is independent of subsequent steps in the sense that any implementation of a sub-program can be used to form the implementation of the specification which gave rise to the sub-specification. In a non-compositional development method, the correctness of one step of development might depend on the subsequent development of the sub-programs.

MULT
ext wr i, j, m: Z
pre true
    MAKEPOS
    ext wr i, j: Z
    pre true
    post $i \geq 0 \wedge i * j = \overleftarrow{i} * \overleftarrow{j}$
    ;
    POSMULT
    pre $i \geq 0$
    post $m = \overleftarrow{i} * \overleftarrow{j}$
post $m = \overleftarrow{i} * \overleftarrow{j}$

Figure 1: Example of Specifications in Place of Code

When programs are presented in this way, the effect is intentionally similar to natural deduction proofs (cf. Section 4). (There are, however, some important differences which are discussed below.) The proof rules are somewhat similar to the deduction rules for logic. Broadly, there is one proof rule for each language construct. In order to present the proof rules[19] in a compact way, the pre- and post-conditions are written in braces before and after the piece of code to which they relate—thus:

$$\{pre\}S\{post\}$$

It is sometimes necessary to use information from a pre-condition in the post-condition. Decorating P with a hook to denote a logical expression which is the same as P except that all free variables are decorated with a hook, the relevant proof rule is:

$$\frac{\{P\}S\{R\}}{\{P\}S\{\overleftarrow{P} \wedge R\}}$$

Thus, for example, from:

$$\{fn = 1\}FACTB\{fn = \overleftarrow{fn} * \overleftarrow{n}!\}$$

it follows that:

$$\{fn = 1\}FACTB\{fn = \overleftarrow{fn} * \overleftarrow{n}! \wedge \overleftarrow{fn} = 1\}$$

and thus:

$$\{fn = 1\}FACTB\{fn = \overleftarrow{n}!\}$$

Notice that the hooking of P (and thus its free variables) is crucial—it is not true that, in the final state:

[19] Readers who are familiar with the original form of Hoare-logic should be reassured that the assertions written here are for total correctness: termination is required for all states satisfying *pre*.

$fn = 1$

The most basic way of combining two operations is to execute them in sequence. It would be reasonable to expect that the first operation must leave the state so that the pre-condition of the second operation is satisfied. In order to write this, a distinction must be made between the relational and single-state properties guaranteed by the first statement. The names of the truth-valued functions have been chosen as a reminder of the distinction between:

$$P: \Sigma \rightarrow \mathbf{B}$$

$$R: \Sigma \times \Sigma \rightarrow \mathbf{B}$$

Writing:

$$R_1 \mid R_2$$

for[20]:

$$\exists \sigma_i \cdot R_1(\overleftarrow{\sigma}, \sigma_i) \wedge R_2(\sigma_i, \sigma)$$

the sequence rule is:

$$\frac{\{P_1\}S_1\{P_2 \wedge R_1\},\ \{P_2\}S_2\{R_2\}}{\{P_1\}S_1; S_2\{R_1 \mid R_2\}}$$

The predicate P_2 can be seen as the designer's choice of interface between S_1 and S_2 whereas R_1 and R_2 fix the functionality of the two components.

Referring to Figure 1, the first conjunct of *post-MAKEPOS* is also seen in *pre-POSMULT*: this defines the interface between the two (as yet to be coded) components. The condition given as *post-POSMULT* is the same as *post-MULT* but the position of the former shows that its hooked variables refer to values which will arise between the execution of *MAKEPOS* and *POSMULT*. The second conjunct of *post-MAKEPOS* is the simplest expression (for R_1 in the rule) which ensures that *post-MULT* is satisfied—in detail, *post-MAKEPOS* can be written as:

$$\textit{post-MAKEPOS}(\overleftarrow{i}, \overleftarrow{j}, \overleftarrow{m}, i, j, m) \ \triangleq \ i * j = \overleftarrow{i} * \overleftarrow{j} \wedge m = \overleftarrow{m}$$

(The non-appearance of m in the external clause of *MAKEPOS* justifies the second conjunct.) Also:

$$\textit{post-POSMULT}(\overleftarrow{i}, \overleftarrow{j}, \overleftarrow{m}, i, j, m) \ \triangleq \ m = \overleftarrow{i} * \overleftarrow{j}$$

Thus:

$$\textit{post-MAKEPOS} \mid \textit{post-POSMULT}$$

becomes:

$$\exists i_i, j_i, m_i \cdot i_i * j_i = \overleftarrow{i} * \overleftarrow{j} \wedge m_i = \overleftarrow{m} \wedge m = i_i * j_i$$

from which:

[20]This is, of course, familiar relational composition. The reason that this does not imply "angelic non-determinism" is the assumption (implementability) given above on pre/post pairs.

$$m = \overleftarrow{i} * \overleftarrow{j}$$

follows.

In practice, it is not normally necessary to proceed formally with such proofs. It becomes rather easy to check an annotated text like that for *MULT*. The only care required is the association of the hooked variables with the values at the beginning of the appropriate operation. A good visual check is given by the nesting. (The generalization to a sequence of more than two statements is straightforward.)

Proof obligations for conditional statements are given by:

$$\frac{\{P \wedge B\}TH\{R\},\ \{P \wedge \neg B\}EL\{R\}}{\{P\}\ \textsf{if}\ B\ \textsf{then}\ TH\ \textsf{else}\ EL\{R\}}$$

Looking at this proof rule, it would appear that the designer has little freedom of choice other than the selection of cases. Consideration of even a simple application—again taken from the multiplication example—shows how the designer's freedom actually arises:

MAKEPOS
ext wr i, j: Z
pre true
  if $i < 0$
  then pre $i < 0$
    post $0 \leq i \wedge i = -\overleftarrow{i} \wedge j = -\overleftarrow{j}$
  else pre $0 \leq i$
    post $0 \leq i \wedge i = \overleftarrow{i} \wedge j = \overleftarrow{j}$
post $0 \leq i \wedge i * j = \overleftarrow{i} * \overleftarrow{j}$

This argument is also using a rule which permits the use of (stronger pre-conditions or) weaker post-conditions:

$$\frac{PP \Rightarrow P,\ \{P\}S\{RR\},\ RR \Rightarrow R}{\{PP\}S\{R\}}$$

The post-conditions of the two statements imbedded within the conditional have been chosen to express the intentions of the designer rather than just being copies of *post-MAKEPOS*. In this way—on a problem of greater size—the designer decouples the design of sub-components from their context.

The proof obligation for iteration—as would be expected—is the most interesting. The general form is:

$$\frac{\{P \wedge B\}S\{P \wedge R\}}{\{P\}\ \textsf{while}\ B\ \textsf{do}\ S\{P \wedge \neg B \wedge R^*\}}$$

R is required to be well-founded and transitive. In order to be well-founded, the logical expression R must be irreflexive, for example:

$$x < \overleftarrow{x}$$

Since the body of the loop might not be executed at all, the state might not be changed by the while loop. Thus the overall post-condition can (only) assume R^* which is the reflexive closure of R—for example:

$x \leq \overleftarrow{x}$

There is a significant advantage in requiring that R be well-founded since the above proof obligation then establishes termination. The rest of this rule is easy to understand. The expression P is an invariant which is true after any number of iterations (including zero) of the loop body. This is, in fact, just a special use of a data type invariant. (Notice that such an invariant could fail to be satisfied within the body of the loop.) The falseness of B after the loop follows immediately from the meaning of the loop construct[21]. Returning again to the multiplication example, *POSMULT* might be implemented as in Figure 2. The reader should check carefully how the terms in these logical expressions relate to the proof rule. Notice that rel is well-founded, since i is always positive and cannot be decreased indefinitely (cf. inv).

POSMULT
ext wr i, m: Z, rd j: Z
pre $0 \leq i$
  $m := 0$
  ;
  pre $0 \leq i$
    while $i \neq 0$ do
    inv $0 \leq i$
      $i := i - 1$;
      $m := m + j$;

    rel $m = \overleftarrow{m} + (\overleftarrow{i} - i) * \overleftarrow{j} \wedge i < \overleftarrow{i}$
  post $m = \overleftarrow{m} + \overleftarrow{i} * \overleftarrow{j}$
post $m = \overleftarrow{i} * \overleftarrow{j}$

Figure 2: Development for Multiplication Example

The implementation in Figure 2 is slow in that it is linear in the value of i. Using the ability of a binary computer to detect the difference between even and odd numbers (by checking the least significant bit) and to multiply or divide by two (by shifting), an algorithm which takes time proportional to the logarithm (base 2) of i is shown in Figure 3. The outer loop of these two algorithms is the same. Notice, however, that the externals clause of *POSMULT* has been modified to permit the necessary assignments to j; the rel clause of the loop has also been changed to cater for the more general case[22].

The comparison is made above between annotated program texts and natural deduction proofs. Although this similarity can be useful, it is important to notice the differences. In the program texts, the same expression denotes different things in different places. The effect of assignment statements is to destroy so-called "referential transparency". It is therefore *not* possible to simply refer to any earlier line in a text in the same way as is done in natural deduction proofs.

It would be reasonable, at this point, to ask how the inv/rel expressions are discovered. The discovery of proofs from code is not the main objective, and this discussion is avoided. It is, however, possible to observe that the proof step is, in some sense, the inverse of the program

[21] The proof rule here and its use of R can be compared with the "decreasing function" in [14]—there it is used only in the termination proof. (This function is called a "variant" in [22].)

[22] A comparison of these two predicates is actually quite interesting. The earlier one shows directly the remaining work to be done; the latter predicate shows an expression whose value is to be kept constant.

```
POSMULT
ext wr i, j, m: Z
pre 0 ≤ i
    m := 0
    ;
    pre 0 ≤ i
        while i ≠ 0 do
        inv 0 ≤ i
            ext wr i, j: Z
            pre i ≠ 0
                while is-even(i) do
                inv 1 ≤ i
                    i := i/2
                    j := j * 2

                rel i * j = ←i * ←j ∧ i < ←i
            post i * j = ←i * ←j ∧ i ≤ ←i
            ;
            m := m + j;
            i := i − 1

        rel m + i * j = ←m + ←i * ←j ∧ i < ←i
    post m = ←m + ←i * ←j
post m = ←i * ←j
```

Figure 3: Alternative Development for Multiplication Example

design activity. As such it serves as a check in the same way that differentiation of an expression derived by integration is a standard check in the infinitesimal calculus.

It is now shown how the proof obligations for programming constructs can be used to stimulate program design steps. It is, however, important that one does not expect too much from this idea. Design requires intuition and cannot, in general, be automated. What is offered is a framework into which the designer's commitments can be placed. If done with care, the verification then represents almost no extra burden. Even so, false steps of design cannot be avoided in the sense that even a verified decision can lead to a blind alley (e.g. a decomposition which has unacceptable performance implications). If this happens, there is no choice but to reconsider the design decision which led to the problem. A mould is being given into which a design explanation can be fitted; it aims only to show that the need for verification can also help the design process.

A simple example of the way in which a proof rule can help a designer's thoughts about decomposition is given by the rule for sequential composition—the assertion P_2 fixes an interface between the two sub-operations. There are advantages in not making such interfaces unnecessarily restrictive. The choice of a general pre-condition for the second operation can result in the specification—and eventual implementation—of a piece of software which is applicable outside the context of the first operation. Such meaningful decompositions are to be sought in all designs.

The design problems presented by iterative constructs are more interesting. Here, judicious use of the rel/inv clauses lead to the specification of the loop body. Two different approaches to the problem of design can be illustrated on the simple example of computing (general) addition by successor. The first of these programs in its annotated form is[23]:

pre $j \geq 0$
  $t := 0; r := i$
  ;
  pre $t \leq j \wedge r = i + t$
    while $t \neq j$ do
    inv $t \leq j \wedge r = i + t$
      $t := t + 1;$
      $r := r + 1$
    rel $\overleftarrow{t} < t \wedge i = \overleftarrow{i} \wedge j = \overleftarrow{j}$
  post $r = \overleftarrow{i} + \overleftarrow{j}$
post $r = \overleftarrow{i} + \overleftarrow{j}$

The overall post-condition here is

$$r = \overleftarrow{i} + \overleftarrow{j}$$

The design decisions to not change i and j and to introduce a temporary variable (t) suggests an invariant (inv):

$$r = i + t$$

to express the progress of the calculation. This easy to establish by initialization. This only leaves, for the relation (rel), the establishment of termination (the relation must be well-founded) and the preservation of the initial values. It is obvious that this relation is transitive. (Strictly in a subsequent step of development) the assignment statements in the body of the loop can be seen to preserve the invariant and to satisfy the relation.

A different program which satisfies the same overall specification is:

[23]These should really be presented in a step-by-step design but their size is such that this would be a waste of space.

pre $j \geq 0$
  $r := i$
  ;
  pre $0 \leq j$
    while $j \neq 0$ do
    inv $0 \leq j$
      $j := j - 1$;
      $r := r + 1$
    rel $r = \overleftarrow{r} + \overleftarrow{j} - j \wedge j < \overleftarrow{j}$
  post $r = \overleftarrow{r} + \overleftarrow{j}$
post $r = \overleftarrow{i} + \overleftarrow{j}$

In this program, the decision to avoid a temporary variable gives rise to a different pattern. The initialization does not obviously establish an invariant which relates the variables. The plan to reduce j suggests something like:

$$r = i + (\overleftarrow{j} - j)$$

but this is not an expression in a single state. However, rel does not have to be: it reflects the work which remains to be done. This time the invariant is simpler because it only serves as a data type invariant (which plays a part in checking that rel is well-founded).

The following table attempts to capture the main differences between loops which work "up" using temporary variables and those which work "down" avoiding temporaries.

	"up"	"down"
	compute the required function of temporary variables	eliminate work "remaining" at each iteration
temporaries	yes	no
initialization	temporaries and results set to "zero"	reflect whole task as "remaining"
initial state	undisturbed	changed
inv	locals = f (temporaries)	data type invariant (only)
rel	current = initial temporaries decreased	f(current) = f(initial) distances to intial decreased
loop test	temporary = some initial	test for "zero"

The two approaches[24] to the task of computing factorial yields a similar analysis of the assertions. The overall post-condition is:

$$fn = \overleftarrow{n}!$$

Taking this as an invariant of the temporary variable leaves only the preservation of n and well-foundedness for rel—see Figure 4. The body of the loop can be completed with the assignments shown.

With the version of the program which does not have a temporary variable, the factorial is computed backwards ($n * (n-1) * \ldots$). This is done by overwriting the value in n, and rel captures this with an expression equivalent to:

$$fn * n! = \overleftarrow{fn} * \overleftarrow{n}!$$

This gives rise to the development shown in Figure 5.

pre $0 \leq n$

$fn := 1; t := 0$

;

pre $t \leq n \wedge fn = t!$

while $t \neq n$ do

inv $t \leq n \wedge fn = t!$

$t := t + 1;\ fn := fn * t$

rel $n = \overleftarrow{n} \wedge \overleftarrow{t} < t$

post $fn = t! \wedge t = n = \overleftarrow{n}$

post $fn = \overleftarrow{n}!$

Figure 4: Development of Factorial

pre $0 \leq n$

$fn := 1$

;

pre $0 \leq n$

while $n \neq 0$ do

inv $0 \leq n$

$fn, n := fn * n, n - 1$

rel $fn = \overleftarrow{fn} * \overleftarrow{n}!/n! \wedge n < \overleftarrow{n}$

post $fn = \overleftarrow{fn} * \overleftarrow{n}!$

post $fn = \overleftarrow{n}!$

Figure 5: Alternative Development of Factorial

pre $j \neq 0$

$q := 0$

;

pre $j \neq 0$

while $i \geq j$ do

inv true

$i, q := i - j, q + 1$

rel $j = \overleftarrow{j} \wedge i < \overleftarrow{i} \wedge j * q + i = \overleftarrow{j} * \overleftarrow{q} + \overleftarrow{i}$

post $\overleftarrow{j} * (q - \overleftarrow{q}) + i = \overleftarrow{i} \wedge i < \overleftarrow{j}$

post $\overleftarrow{j} * q + i = \overleftarrow{i} \wedge i < \overleftarrow{j}$

Figure 6: Development of Integer Division Algorithm

A straightforward development of an integer division algorithm which does overwrite its initial values is shown in Figure 6. As an illustration of a more interesting problem, consider describing how a mechanical calculator performs the same task. In a first stage (SL), j is shifted left until it is larger than i—the number of shifts is recorded in n. The second stage (SR) shifts j back and at each step keeps the expression $j*q+i$ constant. There are two places this must be done: shifting at SRS and re-establishing $i < j$ by stepping down i at SRC. The presentation in Figure 7 is made simpler by assuming that all variables are natural numbers.

pre $j \neq 0$

pre $j \neq 0$ $\{SL\}$

ext rd i, wr j, q, n

$n := 0;$

while $j \leq i$ do

inv true

$j, n := j * 10, n + 1$

rel $j * 10^{\overleftarrow{n}} = \overleftarrow{j} * 10^n \wedge j > \overleftarrow{j}$

post $j = \overleftarrow{j} * 10^n \wedge i < j$

$; q := 0;$

pre 10^n divides $j \wedge i < j$ $\{SR\}$

while $n \neq 0$ do

inv 10^n divides $j \wedge i < j$

$n, j, q := n - 1, j/10, q * 10;$ $\{SRS\}$

while $j \leq i$ $\{SRC\}$ ext wr i, q, rd j

inv $(0 \leq i)$

$i, q := i - j, q + 1$

rel $\overleftarrow{j} * \overleftarrow{q} + \overleftarrow{i} = j * q + i \wedge i < \overleftarrow{i}$

rel $j/10^n = \overleftarrow{j}/10^{\overleftarrow{n}} \wedge j * q + i = \overleftarrow{j} * \overleftarrow{q} + \overleftarrow{i} \wedge n < \overleftarrow{n}$

post $j = \overleftarrow{j}/10^{\overleftarrow{n}} \wedge j * q + i = \overleftarrow{j} * \overleftarrow{q} + \overleftarrow{i} \wedge i < j$

post $\overleftarrow{j} * q + i = \overleftarrow{i} \wedge i < \overleftarrow{j}$

Figure 7: Development of Alternative Integer Division Algorithm

[24] It would be interesting to try to present these strategies in the "d-Calculus" proposed by Michel Sintzoff elsewhere in these proceedings.

Programs for searching and sorting (cf. [36]) provide many interesting examples for proof construction. In the case of searching, the basic vector involved is not changed and proofs using input/output relations differ little from those which use post-conditions of the final state alone. The utility of the more general post-conditions becomes apparent on sorting examples where the basic vector is changed[25]. Consider the following specification:

$SORT$ ()
ext wr l : seq of N
post $is\text{-}ord(l) \wedge is\text{-}perm(l, \overleftarrow{l})$

The truth-valued function for ordering:

$is\text{-}ord$: seq of N $\rightarrow B$

and that for permutations:

$is\text{-}perm$: seq of N $\times$ seq of N $\rightarrow B$

are obvious (although it might be interesting to define the latter via its properties rather than directly).

A very simple sorting strategy is to absorb, at each iteration of the loop, the "next" element into its correct position. This design decision can be shown by:

$SORT$
var i: N
$i := 1$;
while $i < n$ do
inv $is\text{-}ord(l(1, \ldots, i))$
$i := i + 1$
;
$BODY(i)$
pre $i \in$ dom l
post $l(i+1, \ldots) = \overleftarrow{l}(i+1, \ldots) \wedge$
$\exists j \in \{1, \ldots, i\} \cdot$
$l(j) = \overleftarrow{l}(i) \wedge \overleftarrow{l}(1, \ldots, i-1) = del(l(1, \ldots, i), j)$
rel $is\text{-}perm(l, \overleftarrow{l})$
post $is\text{-}ord(l) \wedge is\text{-}perm(l, \overleftarrow{l})$

An equally simple (and similarly inefficient) sorting algorithm is one which picks the lowest of the remaining elements and moves it to the next position on each iteration. The additional property is clearly shown in the following invariant:

[25] During the Summer School, Jon Garnsworthy pointed out that the same problem arises in the so-called "Dutch National Flag" problem—cf. [14].

$SORT$
var i:N
$i := 0$;
while $i < n - 1$ do
inv $is\text{-}ord(l(1,\ldots,i)) \wedge$
$\forall m \in \{1,\ldots,i\}, n \in \{i+1,\ldots\} \cdot l(m) \leq l(n)$
$i := i + 1$
;
$BODY(i)$
pre $i \in$ dom l
post $l(1,\ldots,i-1) = \overleftarrow{l}(1,\ldots,i-1) \wedge$
$\exists j \in \{i,\ldots\} \cdot$
$l(i) = \overleftarrow{l}(j) \wedge l(i+1,\ldots) = del(\overleftarrow{l}(i,\ldots),j)$
rel $is\text{-}perm(l, \overleftarrow{l})$
post $is\text{-}ord(l) \wedge is\text{-}perm(l, \overleftarrow{l})$

A comparison of *post-BODY* in the two cases shows the essence of the work to be performed. The development of more efficient algorithms is left as an exercise to the interested reader.

This section should have established that post-conditions of two states (input-output relations) can be profitably used in program design. The proof rules shown above[26] are only slightly more complicated than those in [23] and the examples here have shown that the separation of, for example, invariants and relations can actually aid the design process.

Several recent papers have attempted to show how programming and specification notation can be merged by translating the former into predicates (see, for example, [25,22,2]). In [25] a motivation is provided for the idea of "weakest pre-specification". The ordering corresponding to sat above is relational containment and does not cope with termination as here. The paper by Eric Hehner in these proceedings manages to use an implication ordering on predicates in a way which gives very attractive properties. The single predicate is formed, essentially, by an expression of the form:

pre $\Rightarrow$ post

As such, the definition is very similar to that used in this paper. Apart from the pragmatic arguments for separating the pre-condition, the system used here does allow additional distinctions to be made between specifications. In both [25] and [22], the basic definitions are used to derive convenient algebraic laws for programming constructs. The presence of these laws blurs the distinction made in the introduction to this paper between the transformational and specify/design/verify approaches to program development.

4 Logic for Partial Functions

One area of *VDM* research which has recently made some progress is the selection of a logic which handles partial expressions in a convenient way. Such partial expressions arise frequently in the specification and design of programs but earlier treatments have not been fully successful. As well as reviewing the sources of partial expressions, this section offers some requirements for an appropriate logic and compares the proposal in [5] and [10] with the requirements.

Many of the operators on the basic *VDM* data types are partial (e.g. hd, map application). They arise in expressions like:

[26] The proof rules presented above can be justified with respect to a denotational semantics of the programming language in question. For the while construct, this is done in Appendix A along with other consequences of the definitions.

$t = [\,] \vee t = append(\mathsf{hd}\ t, \mathsf{tl}\ t)$

if ρ is a member of map Id to Den:

$id \in \mathsf{dom}\ \rho \wedge \rho(id) \in Proctype$

The fact that the operators are partial gives rise to terms which may fail to denote a value. Another obvious source of partial terms is recursion—for example:

$subp : \mathrm{N} \times \mathrm{N} \to \mathrm{N}$

$subp(i,j) \triangleq \mathsf{if}\ i = j\ \mathsf{then}\ 0\ \mathsf{else}\ subp(i, j+1) + 1$

Providing that $i \geq j$, this function yields a defined result. This prompts the writing of expressions like:

$\forall i, j \in \mathrm{N} \cdot i \geq j \Rightarrow subp(i,j) = i - j$

It can clearly be seen how the problem of undefined terms propagates up to the meaning of the logical operators: what does this last expression mean when the antecedent of the implication is false?

There have been a number of historical approaches to this problem. John McCarthy showed how logical expressions could be defined by conditionals ([41])—for example:

$p \wedge q$

is defined as:

$\mathsf{if}\ p\ \mathsf{then}\ q\ \mathsf{else}\ \mathsf{false}$

Operators defined in this way are obviously not commutative. Thus VDL's (cf. [39]) adoption of such a set of operators led to a logic in which familiar properties do not hold. This was unfortunate because many of the operands were completely defined and proofs were hamstrung unnecessarily. (This approach is, however, being further explored in [8].)

In [28] and [14][27], a distinction is made between the conditional (cand, cor) and the classical (and, or) operators. Unfortunately, neither reference offers an axiomatization of the logic and there are some slightly messy properties. For example, while it is obvious that:

$$\neg(E_1\ \underline{or}\ (E_2\ \underline{cand}\ E_3)), \qquad \neg E_1\ \underline{and}\ (\neg E_2\ \underline{cor}\ \neg E_3)$$

are equivalent, it is perhaps less obvious that:

$$E_1\ \underline{and}\ (\neg E_1\ \underline{cor}\ E_2), \qquad E_1\ \underline{cand}\ E_2$$

are equivalent.

In [30] an attempt was made to limit variables by bounded quantifiers as a way of avoiding undefined terms—for example:

$is\text{-}ordered : \mathsf{seq\ of}\ \mathrm{N} \to B$

$is\text{-}ordered(t) \triangleq \forall i \in \{1, \ldots, \mathsf{len}\ t - 1\} \cdot t(i) \leq t(i+1)$

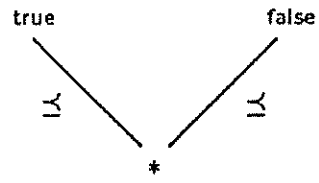

Figure 8: Ordering for Truth Values

This does not always work and in other places explicit conditional expressions were written.

None of these approaches is satisfactory when judged against the following criteria:

- Both a model and a proof theory should be given and the latter should be proved consistent and complete with respect to the former.
- There should be clear links to classical logic—for example:
  - The proof rules should be consistent with classical logic;
  - conjunction and disjunction should be commutative;
  - most standard laws of logic should hold;
  - and there should be a clear way of building a link to those which do not hold;
  - familiar operators should be monotone with respect to the ordering in Figure 8;
  - implication should fit the standard abbreviation($p \Rightarrow q$ as $\neg p \vee q$).
- If there is a need for non-classical, non-monotonic operators, their use should be localized and not inflicted on the developer of standard programs;
- It should be possible to prove results about functions (e.g. *subp*) without a separate proof of definedness of terms[28].

The model theory of [5] is summarized in the following truth tables in which * is used to denote a missing value. The extended truth table for disjunction is:

∨	true	*	false
true	true	true	true
*	true	*	*
false	true	*	false

Notice that the truth table is symmetrical, as also is that for conjunction:

∧	true	*	false
true	true	*	false
*	*	*	false
false	false	false	false

The table for negation is:

[27] The notation of the latter is used here since it is more widely known.

[28] This requirement was added after several proposals (e.g. Manfred Broy's proposals elsewhere in these proceedings which do require such a separation) were made at the Summer School.

¬	
true	false
*	*
false	true

The truth tables for implication and equivalence are derived by viewing them as the normal abbreviations:

⇒	true	*	false
true	true	*	false
*	true	*	*
false	true	true	true

⇔	true	*	false
true	true	*	false
*	*	*	*
false	false	*	true

The proof rules for this logic (derived from [37]) are presented in a way which is intended to be used in (linear-style) natural deduction proofs. The proof rules support the deduction of sequents of the form:

$$\Gamma \vdash E$$

where Γ is a list of expressions. The intended interpretation of such sequents is that E should be true in all worlds where all of the expressions in Γ are true. Notice that the sequent:

$$E_1 \vdash E_2$$

is valid if E_1 is false or undefined, whatever the value of E_2.

For the basic propositional operators ($\neg$, $\vee$) there are the obvious introduction and elimination rules. (All of the rules are given in Appendix B.) In addition, it is necessary to have rules for negated disjunctions (i.e. $\neg\vee$-I). The need for these rules arises from the fact that the "law of the excluded middle" does not hold in this logic. Conjunction and implication are introduced by definitions and their introduction and elimination rules are proved as derived results. An example of a natural deduction proof using these rules to show:

$$(E_1 \vee E_2) \wedge (E_1 \vee E_3) \vdash E_1 \vee E_2 \wedge E_3$$

is given[29] in [33]—the proof would be valid in classical logic whereas the normal proof written in classical logic (cf. [17]) is not valid here because it uses the "law of the excluded middle".

The axiomatization of the predicate calculus follows a similar pattern with the existential quantifier being treated as basic and the universal quantifier being introduced as an abbreviation for which inference rules have to be derived (cf. Appendix B)[30]. The main point with this treatment is to constrain the bound variable of a quantified expression to range only over "proper elements".

The basic characterization of sets like the natural numbers is given by constructor functions:

0: N

$succ$: N → N

[29] The functional style of the justifications permits compositions of steps to be used—the whole style is very much in the spirit of the "d-Calculus" proposed by Michel Sintzoff elsewhere in these proceedings.

[30] Considerable effort has been put into the development of derived rules for this version of logic. Many of the proofs are contained in [33].

and an induction axiom:

$$\text{N-ind} \qquad \frac{p(0);\ n \in \mathrm{N}, p(n) \vdash p(n+1)}{n \in \mathrm{N} \vdash p(n)}$$

Notice that the induction rule is presented via a turnstile rather than using implication. This simplifies subsequent proofs because it avoids the need to use the $\Rightarrow$-I rule. The induction rule is also presented without quantifiers since they can be inserted using the $\forall$-I rule. For recursively defined types such as *Binnode* an induction rule is generated for each type, thus for:

$$\begin{array}{lll} \mathit{Binnode} :: & \mathit{lt} & : [\mathit{Binnode}] \\ & k & : \mathit{Key} \\ & d & : \mathit{Data} \\ & \mathit{rt} & : [\mathit{Binnode}] \end{array}$$

the induction rule is:

$$\mathit{Binnode}\text{-ind} \qquad \frac{\begin{array}{l} p(\mathit{mk\text{-}Binnode}(\mathsf{nil}, k, d, \mathsf{nil})); \\ k \in \mathrm{N}, d \in \mathit{Data}, \mathit{lt}, \mathit{rt} \in \mathit{Binnode}, p(\mathit{lt}), p(\mathit{rt}) \vdash \\ \qquad\qquad p(\mathit{mk\text{-}Binnode}(\mathit{lt}, k, d, \mathit{rt})) \end{array}}{\mathit{bn} \in \mathit{Binnode} \vdash p(\mathit{bn})}$$

The requirement to minimize the use of non-monotonic operators has proved the most elusive and several attempts have been made in order to minimize the use of non-monotonic operators in normal proofs. The approach in [5] was to handle definitions like that for *subp* by generating inference rules of the form:

$$d_b \qquad \frac{}{\mathit{subp}(n, n) = 0}$$

$$d_i \qquad \frac{n_1 \neq n_2;\ \mathit{subp}(n_1, n_2 + 1) = n_3}{\mathit{subp}(n_1, n_2) = n_3 + 1}$$

The reason that this works revolves around the distinction between strong (==) and weak (=) equality and, in particular, the use of the latter in the hypothesis of d_i (rule d_i relies on the fact that the second hypothesis is undefined in exactly the cases needed to avoid relying on the conclusion of the rule). The differences between the strict weak equality and the non-monotonic strong equality can be seen from the following tables (here, the undefined values are shown as "bottom" elements ($\perp$)).

=	0	1	2	...	$\perp_{\mathrm{N}}$
0	true	false	false		$\perp_B$
1	false	true	false		$\perp_B$
2	false	false	true		$\perp_B$
...					
$\perp_{\mathrm{N}}$	$\perp_B$	$\perp_B$	$\perp_B$		$\perp_B$

==	0	1	2	...	$\perp_{\mathrm{N}}$
0	true	false	false		false
1	false	true	false		false
2	false	false	true		false
...					
$\perp_{\mathrm{N}}$	false	false	false		true

The justification of such rules (with respect to the definition of *subp*) does require the use of—and reasoning about—strong equality; but the proof of the appropriate property in the referenced paper only uses the rules d_b and d_i.

In [32] a slightly different approach is used which obviates the need to create the inference rules. The idea is to use definition rules in direct substitutions (cf. $=$ -*subs*, $\triangleq$-*subs*, if-*subs*). This permits proofs to avoid mentioning undefined values. The only non-obvious step in devising the proof shown in Figure 9 was deciding to conduct the main induction on $subp(i, i\text{-}n)$.

	from $i, j \in \mathrm{N}$	
1	$i - 0 = i \in \mathrm{N}$	h,N
2	$subp(i, i - 0) = 0$	ifth-subs/$subp$(h,1)
3	$0 \leq i \Rightarrow subp(i, i - 0) = 0$	vac $\Rightarrow$ -I(2)
4	from $n \in \mathrm{N};\ n \leq i \Rightarrow subp(i, i - n) = n$	
4.1	from $n + 1 \leq i$	
4.1.1	$n \leq i$	h4.1, N
4.1.2	$subp(i, i - n) = n$	vac $\Rightarrow$ -E(h4,4.1.1)
4.1.3	$i \neq i - (n + 1)$	N, h4
4.1.4	$n + 1 = n + 1$	=-term(h4, N)
4.1.5	$subp(i, i - n) + 1 = n + 1$	=t-subs(4.1.2, 4.1.4)
	infer $subp(i, i - (n + 1)) = n + 1$	ifel-subs/$subp$(h,4.1.3,4.1.5)
4.2	$(n + 1 \leq i) \in \mathrm{B}$	h4, h, N
	infer $n + 1 \leq i \Rightarrow subp(i, i - (n + 1)) = n + 1$	$\Rightarrow$-I(4.1,4.2)
5	$\forall n \in \mathrm{N} \cdot n \leq i \Rightarrow subp(i, i - n) = n$	$\forall$-I(N-ind(3,4))
6	from $i \geq j$	
6.1	$i - j \in \mathrm{N}$	N,h6
6.2	$0 \leq j \Rightarrow subp(i, j) = i - j$	$\forall$-E(5,6.1),N
	infer $subp(i, j) = i - j$	vac $\Rightarrow$ -E(6.2,h)
7	$(i \geq j) \in \mathrm{B}$	h,N
	infer $i \geq j \Rightarrow subp(i, j) = i - j$	$\Rightarrow$-I(6,7)

Figure 9: Proof about *subp*

The principal differences between *LPF* ("Logic for Partial Functions") and classical logic should be noted. It is an obvious consequence of the truth-tables that the law of the excluded middle does not hold. Nor does the deduction theorem hold without an additional hypothesis (cf. $\Rightarrow$ -I). For weak equality it is not necessarily true that $t = t$ for an arbitrary term t.

On the other hand, $\wedge$ and $\vee$ are commutative and monotone. Properties like:

$$x \in R \vdash x = 0 \vee x/x = 1$$

are easily proved. The implication operator fits its normal abbreviation and also has an interpretation which fits the needs of the result on *subp*. Many of the results from classical logic do hold (cf. Appendix B—even most of the properties of $\Leftrightarrow$ presented by Edsger Dijkstra in his Royal Society Lecture of 1985) although simple tautologies have to be re-expressed as sequents. Where properties would otherwise fail to hold, hypotheses can be added as to the definedness of expressions which bring the results back to those of classical logic.

In [10] (which should also be consulted for a full list of references) a number of completeness results are given:

- The operators tt, ff, uu, $\neg$, $\wedge$, $\vee$ form a set which are expressively complete for monotonic

operators (result due to Koletsos).

- The set $\neg$, $\vee$, uu, $\triangle$ are expressively complete for all operators (Cheng 3.1(1)).
- The basic axiomatization is consistent and complete (propositional calculus—Cheng 3.3; predicate calculus—Cheng 4.3/4.4).
- The "cut elimination" theorem holds (Cheng 5).
- The I/E rules for linear-style natural deduction proofs are consistent and complete (Cheng 7.4).

The thesis also contains a discussion of the influence of equality on the logic. There are, of course, still unresolved questions:

- Before this logic can be compared with others such as those in [45], [47], [7], [2], it will be necessary to conduct a number of experiments with typical application proofs.
- The use of undefined predicates needs further study.
- The problems caused by implementing such a monotonic logic in a programming language must be assessed.

Other approaches are the use of LCF in [16], the constructive approach in [15] and the work of Dana Scott (see, for example, [48]). A forthcoming paper will make a fuller comparison with the usability of various systems ([11])[31].

Acknowledgements

The author is grateful to the directors of the Summer School for the invitation to present this material. The ideas presented owe much to other researchers: Section 2 to Tobi Nipkow, Section 3 to Peter Aczel, and Section 4 to Jen Cheng. Jeff Sanders read the preprint in detail and suggested a number of corrections. Julie Hibbs typed and revised the LaTeX script. The author also acknowledges the financial support of SERC and the stimulus of the meetings of IFIP's WG 2.3.

References

[1] H^3.M.S^4 . Data refinement refined. May 1985. typescript, Oxford Programming Research Group.

[2] J.R. Abrial. Programming as a mathematical exercise. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 113–139, Prentice-Hall International, 1985.

[3] P. Aczel. A note on program verification. January 1982. manuscript.

[4] R.C. Backhouse. *Program Construction and Verification.* Prentice-Hall International, 1986.

[31] This will include a comparison with the work of Jean-Raymond Abrial and the Oxford group which was brought into discussion at Marktoberdorf by Mike Spivey and the proposal by Oliver Schoett to use "existential equality".

[5] H. Barringer, J.H. Cheng, and C.B. Jones. A logic covering undefinedness in program proofs. In *Acta Informatica Volume 21*, pages 251–269, Springer-Verlag, 1984.

[6] Dines Bjørner and Cliff B. Jones. *Formal Specification and Software Development.* Prentice Hall Int., 1982. 501 pages.

[7] S.R. Blamey. *Partial Valued Logic.* PhD thesis, Oxford University, 1980.

[8] A. Blikle. Denotational constructors. 1986. typescript.

[9] R.S. Boyer and J. Strother Moore. A verification condition generator for fortran. In R.S. Boyer and J. Strother Moore, editors, *The Correctness Problem in Computer Science*, pages 9–101, Academic Press, 1981.

[10] J.H. Cheng. *A Logic for Partial Functions.* PhD thesis, University of Manchester, 1986.

[11] J.H. Cheng and C.B. Jones. On the usability of logics which handle partial functions. forthcoming.

[12] CIP Language Group. *The Munich Project CIP—Volume 1: The Wide Spectrum Language CIP-L.* Volume 183 of *Lecture Notes in Computer Science*, Springer-Verlag, 1985.

[13] I.D. Cottam, C.B. Jones, T. Nipkow, A.C. Wills, and A. Yaghi. Project support environments for formal methods. In J. McDermid, editor, *Integrated Project Support Environments*, chapter 3, Peter Peregrinus Ltd., 1985.

[14] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976. In Series in Automatic Computation.

[15] R.L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall International, 1986.

[16] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF.* Volume 78 of *Lecture Notes in Computer Science*, Springer-Verlag, 1979.

[17] D. Gries. *The Science of Computer Programming.* Springer-Verlag, 1981.

[18] S.L. Hantler and J.C. King. An introduction to proving the correctness of programs. *ACM Computing Surveys*, 8(3):331–353, September 1976.

[19] I. Hayes. *Specification Case Studies.* Prentice-Hall International, 1987.

[20] J. He, C.A.R. Hoare, and J.W. Sanders. Data refinement refined: resume. In B. Robinet and R. Wilhelm, editors, *ESOP '86*, Springer-Verlag, 1986.

[21] J. He., C.A.R. Hoare, and J.W. Sanders. Prespecification in data refinement. 1986. submitted to Information Processing Letters.

[22] E.C.R. Hehner. *The Logic of Programming.* Prentice-Hall International, 1984.

[23] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[24] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[25] C.A.R. Hoare, He Jifeng, I.J. Hayes, C.C. Morgan, J.W. Sanders, I.H. Sørensen, J.M. Spivey, B.A. Sufrin, and A.W. Roscoe. *Laws of Programming: A Tutorial Paper.* Technical Report PRG-45, Oxford University Programming Research Group, May 1985.

[26] C.B. Jones: *Development Methods for Computer Programs including a Notion of Interference.* Technical Report 25, Oxford University Programming Research Group, June 1981.

[27] C.B. Jones. *Formal Definition in Compiler Development.* Technical Report 25.145, IBM Laboratory Vienna, February 1976. 76 pages.

[28] C.B. Jones. Formal development of correct algorithms: an example based on earley's recogniser. January 1972. ACM SIGPLAN Conference on "Proving Assertions about Programs" (Proceedings in SIGPLAN Notices Volume 7 Number 1).

[29] C.B. Jones. Implementation bias in constructive specification of abstract objects. September 1977. 16 pages - manuscript.

[30] C.B. Jones. *Software Development: A Rigorous Approach.* Prentice Hall Int., Englewood Cliffs, NJ, 1980. 400 pages.

[31] C.B. Jones. Systematic program development. In N. Gehani and A.D. McGettrick, editors, *Software Specification Techniques*, pages 89–110, Addison-Wesley, 1985.

[32] C.B. Jones. *Systematic Software Development Using VDM.* Prentice Hall Int., 1986.

[33] C.B. Jones. *Teaching Notes for Systematic Software Development Using VDM.* Technical Report UMCS 86-4-2, University of Manchester, 1986.

[34] C.B. Jones. A technique for showing that two functions preserve a relation between their domains. *Vienna LR*, 25.3.067, April 1970. 14 pages.

[35] C.B. Jones and C.P. Wadsworth. Some requirements for formal reasoning support. forthcoming.

[36] D.E. Knuth. *Sorting and Searching.* Volume III of *The Art of Computer Programming*, Addison-Wesley Publishing Company, 1975.

[37] G. Koletsos. *Sequent Calculus and Partial Logic.* Master's thesis, Manchester University, 1976.

[38] P. Lucas. *Two Constructive Realizations of the Block Concept and their Equivalence.* Technical Report TR 25.085, IBM Laboratory Vienna, June 1968.

[39] P. Lucas and K. Walk. *On The Formal Description of PL/I.* Volume 6 of *Annual Review in Automatic Programming Part 3*, Pergamon Press, 1969.

[40] Lynn S. Marshall. *A Formal Specification of Straight Lines on Graphic Devices*, pages 129–147. Volume 2: Colloquium on Software Engineering of *Formal Methods and Software Development — Proceedings of the Int. Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, Springer Verlag, Berlin, 1985. Lecture Notes in Computer Science, Vol. 186.

[41] J. McCarthy. A basis for a mathematical theory for computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70, North-Holland Publishing Company, 1967.

[42] F.L. Morris and C.B. Jones. An early program proof by Alan Turing. In *Annals of the History of Computing, Volume 6, No.2*, pages 139–143, April 1984.

[43] T. Nipkow. Non-deterministic data types: models and implementations. In *Acta Informatica Volume 22*, pages 629–661, Springer-Verlag, 1986.

[44] T. Nipkow. *Non-Deterministic Data Types: Models and Implementations.* Technical Report, University of Manchester, June 1985.

[45] O. Owe. An approach to program reasoning based on a first order logic for partial functions. June 1984. privately circulated.

[46] D.M.R. Park. On the semantics of fair parallelism. In D. Bjørner, editor, *Abstract Software Specifications*, Springer-Verlag, 1980.

[47] G.D. Plotkin. Partial function logic. 1985. Lectures at Edinburgh University.

[48] D. Scott. Existence and description in formal logic. In Schoemann, Allen, and Unwin, editors, *Bertrand Russell, Philosopher of the Century*, 1967.

[49] A. Tarlecki. A language of specified programs. *Science of Computer Programming*, 5(1):59–81, 1985.

A Proofs about Operation Decomposition

The main purposes of this Appendix are to provide a semantic model against which the proof rules for operation decomposition can be judged and to prove that the rules are consistent with that model. Some other important properties of the model are also discussed.

A.1 Semantics

The constructs used for operation decomposition in the body of this paper are sequential composition, conditional statements and a repetitive construct. The following abstract syntax shows the language to be considered:

Stmt = *Atomic* | *Composition* | *If* | *While*
Composition :: *Stmt Stmt*
If :: *Expr Stmt Stmt*
While :: *Expr Stmt*

The definition of this language has to cope with both non-termination and non-determinacy. It is clear that the *While* construct could result in non-termination; it is also assumed that the elements of *Atomic* might be undefined for some states. Non-determinacy can arise from the *Atomic* statements. (It is shown below that specifications in the pre-/post-condition form can be used in place of such statements.) The semantic model used here to cope with these problems consists of a pair with a first element which is a termination set (of elements of Σ) and a meaning relation (from $\Sigma \times \Sigma$)[1] Thus:

$$\mathcal{M}: Stmt \rightarrow \mathcal{P}(\Sigma \times \Sigma)$$

$$\mathcal{T}: Stmt \rightarrow \mathcal{P}(\Sigma)$$

The set given by $\mathcal{T}$ is the set over which termination is guaranteed; the domain of the relation $\mathcal{M}$ can be thought of as showing a set over which termination is possible. Intuitively, it should be clear that :

$$\forall s \in Stmt \cdot \mathcal{T}[\![s]\!] \subseteq \mathrm{dom}\, \mathcal{M}[\![s]\!]$$

That this property holds for non-atomic statements is proved below: it is an assumption on the meaning of the *Atomic* statements:
Assumption (A1)

$$\forall s \in Atomic \cdot \mathcal{T}[\![s]\!] \subseteq \mathrm{dom}\, \mathcal{M}[\![s]\!]$$

The meaning relation for the sequential composition of two statements is defined as the relational composition[2] of the meaning relations of these two statements:

$$\mathcal{M}[\![mk\text{-}Composition(S_1, S_2)]\!] \triangleq \mathcal{M}[\![S_1]\!]; \mathcal{M}[\![S_2]\!]$$

This would, taken alone, require so-called "angelic nondeterminism"; this is avoided by showing the required termination set to be (only)[3]:

$$\mathcal{T}[\![mk\text{-}Composition(S_1, S_2)]\!] \triangleq \mathcal{T}[\![S_1]\!] - \mathrm{dom}\,(\mathcal{M}[\![S_1]\!] \rhd \mathcal{T}[\![S_2]\!])$$

[1] This follows the approach used in [26] which was, in turn, prompted by [46].
[2] $R_1 \; ; R_2 \triangleq \{(\sigma, \sigma'') \mid \exists \sigma' \cdot (\sigma, \sigma') \in R_1 \;\wedge (\sigma', \sigma'') \in R_2 \;\}$
[3] $R \rhd S \triangleq \{(\overline{\sigma}, \sigma) \in R \mid \sigma \notin S\}$

This definition shows that the composition is only required to terminate if both S_1 terminates and if the meaning relation can not give rise to a state in which S_2 fails to terminate.

The meaning relation for this simple conditional construct[4] is given by[5]:

$$\mathcal{M}[\![mk\text{-}If(B, TH, EL)]\!] \triangleq (\overline{B} \lhd \mathcal{M}[\![TH]\!]) \cup (\overline{\neg B} \lhd \mathcal{M}[\![EL]\!])$$

The termination set is given by:

$$\mathcal{T}[\![mk\text{-}If(B, TH, EL)]\!] \triangleq (\overline{B} \cap \mathcal{T}[\![TH]\!]) \cup (\overline{\neg B} \cap \mathcal{T}[\![EL]\!])$$

The least fixed point operator (*fix*) is needed to define the semantics for *While* statements. The meaning relation is defined by:

$$\mathcal{M}[\![mk\text{-}While(B, S)]\!] \triangleq fix(\lambda r \cdot (\overline{\neg B} \lhd I) \cup (\overline{B} \lhd \mathcal{M}[\![S]\!]; r)$$

This presents the same (angelic non-determinism) problem as with sequential composition. A similar approach to its resolution can be given:

$$\mathcal{T}[\![mk\text{-}While(B, S)]\!] \triangleq fix(\lambda s \cdot \overline{\neg B} \cup ((\overline{B} \cap \mathcal{T}[\![S]\!]) - \mathrm{dom}(\mathcal{M}[\![S]\!] \rhd s)))$$

It is shown below that this definition does present some problems since it is equivalent to:

$$fix(\lambda s \cdot \overline{\neg B} \cup \{\overleftarrow{\sigma} \in (\overline{B} \cap \mathcal{T}[\![S]\!]) \mid \forall \sigma \cdot (\overleftarrow{\sigma}, \sigma) \in \mathcal{M}[\![S]\!] \Rightarrow \sigma \in s\})$$

because:

$$\begin{aligned}
&\{\overleftarrow{\sigma} \in s_1 \mid \forall \sigma \cdot (\overleftarrow{\sigma}, \sigma) \in r_1 \Rightarrow \sigma \in s_2\} \\
&\quad = \{\overleftarrow{\sigma} \in s_1 \mid \forall \sigma \cdot (\overleftarrow{\sigma}, \sigma) \notin r_1 \vee \sigma \in s_2\} \\
&\quad = \{\overleftarrow{\sigma} \in s_1 \mid \neg \exists \sigma \cdot (\overleftarrow{\sigma}, \sigma) \in r_1 \wedge \sigma \notin s_2\} \\
&\quad = \{\overleftarrow{\sigma} \in s_1 \mid \overleftarrow{\sigma} \notin \mathrm{dom}(r_1 \rhd s_2)\} \\
&\quad = s_1 - \mathrm{dom}(r_1 \rhd s_2)
\end{aligned}$$

and the equation with the embedded universal quantifier is not ω-continuous.

A.2 Properties of Semantics

The first property to be considered is the containment of the termination set within the domain of the meaning relation. This has been assumed for *Atomic* statements; it must now be shown to inherit across each of the constructions for composite statements.

The result required for *Composition* is:

Lemma (C1)

$$\begin{aligned}
&\mathcal{T}[\![S_1]\!] \subseteq \mathrm{dom}\,\mathcal{M}[\![S_1]\!], \mathcal{T}[\![S_2]\!] \subseteq \mathrm{dom}\,\mathcal{M}[\![S_2]\!] \vdash \\
&\qquad \mathcal{T}[\![mk\text{-}Composition(S_1, S_2)]\!] \subseteq \mathrm{dom}\,\mathcal{M}[\![mk\text{-}Composition(S_1, S_2)]\!]
\end{aligned}$$

The proof[6] is given in Figure 11 this proof uses a Lemma (T1) on $\mathcal{T}$[7] whose proof is given in Figure 10.

The result required for *If* statements is:

Lemma (I1)

$$\begin{aligned}
&\mathcal{T}[\![TH]\!] \subseteq \mathrm{dom}\,\mathcal{M}[\![TH]\!], \mathcal{T}[\![EL]\!] \subseteq \mathrm{dom}\,\mathcal{M}[\![EL]\!] \vdash \\
&\qquad \mathcal{T}[\![mk\text{-}If(B, TH, EL)]\!] \subseteq \mathrm{dom}\,\mathcal{M}[\![mk\text{-}If(B, TH, EL)]\!]
\end{aligned}$$

---

[4] The set of states satisfying the truth-valued function p is written $\overline{p}$; I is the identity relation (on Σ). Here, expression evaluation is assumed to be total; [26] shows how the more general case is handled by a ($\mathcal{T}$) function which gives the states over which an expression is defined.

[5] An additional source of non-determinacy can be brought in by using guarded conditional or loop constructs as discussed in [14]; the semantics is given for the former (and the relevant properties proved) in [26].

[6] In each of these proofs an appeal to $\mathcal{M}$ or $\mathcal{T}$ is to be read as a reference to the appropriate case of the semantic function.

[7] This Lemma was suggested by Lynn Marshall.

	from $s_1 \subseteq \mathsf{dom}\, r_1$	
1	$s - \mathsf{dom}\,(s \lhd r) = s - \mathsf{dom}\, r$	–
2	$\mathsf{dom}\, r_2 - \mathsf{dom}\, r_3 \subseteq \mathsf{dom}\,(r_2 - r_3)$	dom
3	$\mathsf{dom}\,(r - (r \rhd s)) = \mathsf{dom}\,(r \rhd s)$	dom, $\rhd$
4	$s_1 - \mathsf{dom}\,(r_1 \rhd s_2)$	1
	$= s_1 - \mathsf{dom}\,(s_1 \lhd r_1 \rhd s_2)$	
5	$= \mathsf{dom}\,(s_1 \lhd r_1) - \mathsf{dom}\,(s_1 \lhd r_1 \rhd s_2)$	4,h
6	$\subseteq \mathsf{dom}\,((s_1 \lhd r_1) - (s_1 \lhd r_1 \rhd s_2))$	5,2
	infer $s_1 - \mathsf{dom}\,(r_1 \rhd s_2) \subseteq \mathsf{dom}\,(s_1 \lhd r_1 \rhd s_2)$	6,3

Figure 10: Proof of Lemma T1

	from $\mathcal{T}[\![S_1]\!] \subseteq \mathsf{dom}\,\mathcal{M}[\![S_1]\!]$, $\mathcal{T}[\![S_2]\!] \subseteq \mathsf{dom}\,\mathcal{M}[\![S_2]\!]$	
1	$\mathsf{dom}\,(\mathcal{M}[\![mk\text{-}Composition(S_1, S_2)]\!]) = \mathsf{dom}\,(\mathcal{M}[\![S_1]\!]; \mathcal{M}[\![S_2]\!])$	$\mathcal{M}$
2	$\mathcal{T}[\![mk\text{-}Composition(S_1, S_2)]\!]$	$\mathcal{T}$
	$= \mathcal{T}[\![S_1]\!] - \mathsf{dom}\,(\mathcal{M}[\![S_1]\!] \rhd \mathcal{T}[\![S_2]\!])$	
3	$\subseteq \mathsf{dom}\,(\mathcal{T}[\![S_1]\!] \lhd \mathcal{M}[\![S_1]\!] \rhd \mathcal{T}[\![S_2]\!])$	T1(h)
4	$\subseteq \mathsf{dom}\,(\mathcal{M}[\![S_1]\!] \rhd \mathcal{T}[\![S_2]\!])$	3,$\lhd$,dom
5	$\subseteq \mathsf{dom}\,(\mathcal{M}[\![S_1]\!] \rhd \mathsf{dom}\,\mathcal{M}[\![S_2]\!])$	4,h
6	$\subseteq \mathsf{dom}\,(\mathcal{M}[\![S_1]\!]; \mathcal{M}[\![S_2]\!])$	5,dom
	infer $\mathcal{T}[\![mk\text{-}Composition(S_1, S_2)]\!] \subseteq \mathsf{dom}\,(\mathcal{M}[\![mk\text{-}Composition(S_1, S_2)]\!])$	1,6

Figure 11: Proof of Lemma C1

The proof of this is straightforward.

The result required for *While* statements is:

Lemma (W1)

$$\mathcal{T}[\![S]\!] \subseteq \mathsf{dom}\, \mathcal{M}[\![S]\!] \vdash \mathcal{T}[\![mk\text{-}While(B,S)]\!] \subseteq \mathsf{dom}\, \mathcal{M}[\![mk\text{-}While(B,S)]\!]$$

A direct proof of this result is made difficult to understand because the technicalities of the need to reason about a continuous version of $\mathcal{T}$ are obscured by the length of the specific definitions for *While* statements. It is worth proving a more general result about such recursive semantic equations:

Lemma (R1)

$$BS \subseteq \mathsf{dom}\, BR, IS \subseteq \mathsf{dom}\, IR \vdash$$
$$fix(\lambda s \cdot BS \cup (IS - \mathsf{dom}\,(IR \rhd s))) \subseteq \mathsf{dom}\, fix(\lambda r \cdot BR \cup (IR; r))$$

A continuous version of the expression in the first term is used in the proof:

$$\mathcal{F}(s) \triangleq BS \cup \{\overleftarrow{\sigma} \in IS \mid \exists \sigma \cdot (\overleftarrow{\sigma}, \sigma) \in IR \wedge \sigma \in s\}$$
$$\subseteq BS \cup \mathsf{dom}\,(IR \rhd s)$$

another expression used in the proof is:

$$\mathcal{G}(r) \triangleq BR \cup (IR; r)$$

The proof is given in Figure 12. Then, using the following identities:

$$BS = \overline{\neg B}$$
$$BR = \overline{\neg B} \lhd I$$
$$IS = \overline{B} \cap \mathcal{T}[\![S]\!]$$
$$IR = \overline{B} \lhd \mathcal{M}[\![S]\!]$$

and noting that:

$$BS = \mathsf{dom}\, BR$$
$$\mathcal{T}[\![S]\!] \subseteq \mathsf{dom}\, \mathcal{M}[\![S]\!] \vdash IS \subseteq \mathsf{dom}\, IR$$

the proof of W1 is an immediate corollary of R1.

The above results (A1, C1, I1, W1) combine to give the result for the whole language:

Theorem (L1)

$$\forall s \in Stmt \cdot \mathcal{T}[\![s]\!] \subseteq \mathsf{dom}\, \mathcal{M}[\![s]\!]$$

The notion of *satisfaction* is introduced in the body of this paper. It can be expressed in the notation used here as:

$$S' \;\mathsf{sat}\; S \triangleq \mathcal{T}[\![S]\!] \subseteq \mathcal{T}[\![S']\!] \wedge \mathcal{T}[\![S]\!] \lhd \mathcal{M}[\![S']\!] \subseteq \mathcal{M}[\![S]\!]$$

If S and S' are set/relation pairs, $\mathcal{T}$ and $\mathcal{M}$ select the first and second elements respectively. The reduction of a pre/post-condition specification to such pairs is described in Section 2. The semantics for *Stmt* given above completes the picture: specifications and programs can be treated on a common semantic footing. There is, furthermore, no difficulty with inserting specifications as *Atomic* statements in programs as was done in Section 3 (cf. the division and the *SORT* examples). Footnote 18 pinpoints the requirement that a development method be *compositional*. What does this mean in terms of the sat relation? It requires that each of the ways of forming composite statements is monotone in the sat ordering. Thus for sequential composition:

Lemma (C2)

$$S_1' \;\mathsf{sat}\; S_1, S_2' \;\mathsf{sat}\; S_2 \vdash mk\text{-}Composition(S_1', S_2') \;\mathsf{sat}\; mk\text{-}Composition(S_1, S_2)$$

	from $BS \subseteq \mathrm{dom}\, BR,\ IS \subseteq \mathrm{dom}\, IR$	
1	$\mathcal{F}^0(\{\}) \subseteq \mathrm{dom}\, \mathcal{G}^0(\{\})$	set
2	$\mathcal{F}^1(\{\}) \subseteq \mathrm{dom}\, \mathcal{G}^1(\{\})$	$\mathcal{F}$,$\mathcal{G}$,h
3	from $\mathcal{F}^n(\{\}) \subseteq \mathrm{dom}\, \mathcal{G}^n(\{\}), n \geq 1$	
3.1	$\mathcal{F}^{n+1}(\{\}) \subseteq BS \cup \mathrm{dom}\,(IR \rhd \mathcal{F}^n(\{\}))$	$\mathcal{F}$
3.2	$\subseteq BS \cup \mathrm{dom}\,(IR \rhd \mathrm{dom}\, \mathcal{G}^n(\{\}))$	3.1,h3
3.3	$\mathcal{G}^{n+1}(\{\}) = BR \cup (IR; \mathcal{G}^n(\{\}))$	$\mathcal{G}$
3.4	$\mathrm{dom}\, \mathcal{G}^{n+1}(\{\})$	3.2,dom
	$= \mathrm{dom}\, BR \cup \mathrm{dom}\,(IR; \mathcal{G}^n(\{\}))$	
3.5	$= \mathrm{dom}\, BR \cup \mathrm{dom}\,(IR \rhd \mathrm{dom}\, \mathcal{G}^n(\{\}))$	3.4,dom , $\rhd$
	infer $\mathcal{F}^{n+1}(\{\}) \subseteq \mathrm{dom}\, \mathcal{G}^{n+1}(\{\})$	3.2,3.5,h
4	$\forall n \in \mathrm{N} \cdot \mathcal{F}^n(\{\}) \subseteq \mathrm{dom}\, \mathcal{G}^n(\{\})$	N-ind(1,2,3)
5	$\bigcup_{n \leq 0} \mathcal{F}^n(\{\}) \subseteq \mathrm{dom} \bigcup_{n \leq 0} \mathcal{G}^n(\{\})$	4, monotonicity
6	$fix(\lambda s \cdot BS \cup (IS - \mathrm{dom}\,(IR \rhd s)))$	T1(6)
	$\subseteq fix(\lambda s \cdot BS \cup \mathrm{dom}\,(IS \lhd IR \rhd s))$	
7	$\subseteq fix(\lambda s \cdot BS \cup \mathrm{dom}\,(IR \rhd s))$	6,h
8	$\subseteq \bigcup_{n \geq 0} \mathcal{F}^n(\{\})$	7,$\mathcal{F}$,*fix*
	infer $fix(\lambda s \cdot BS \cup (IS - \mathrm{dom}\,(IR \rhd s))) \subseteq \mathrm{dom}\, fix(\lambda r \cdot BR \cup IR; r)$	8,5,$\mathcal{G}$,*fix*

Figure 12: Proof of Lemma R1

The proof of this result is straightforward, as is that for *If* statements:
Lemma(I2)

$$TH' \text{ sat } TH, EL' \text{ sat } EL \vdash mk\text{-}If(B, TH', EL') \text{ sat } mk\text{-}If(B, TH, EL)$$

The final result of this form on the whole language (L2 below) ensures that, if a series of statements are developed so as to satisfy a specification SP, then the statements can be used in the implementation of any design which needs a component specified as SP; and that implementation is bound to satisfy its specification even though its proof used only the properties SP and not the series of statements.

But the achievement of this goal requires that the corresponding property is proved for *While* statements. As would be expected, this result is a little more difficult and is split into three lemmas. Firstly:
Lemma (W2)

$$S' \text{ sat } S \vdash \mathcal{T}[\![mk\text{-}While(B, S)]\!] \subseteq \mathcal{T}[\![mk\text{-}While(B, S')]\!]$$

Lemma (W3)

$$\mathcal{T}[\![S]\!] \lhd \mathcal{M}[\![S']\!] \subseteq \mathcal{M}[\![S]\!] \vdash \mathcal{T}[\![mk\text{-}While(B, S)]\!] \lhd \mathcal{M}[\![mk\text{-}While(B, S')]\!] \subseteq \mathcal{M}[\![mk\text{-}While(B, S)]\!]$$

These proofs are given in Figures 13 and 14 respectively.

The final result:
Lemma (W4)

$$S' \text{ sat } S \vdash mk\text{-}While(B, S') \text{ sat } mk\text{-}While(B, S)$$

	from S' sat S	
1	$\mathcal{T}[\![S]\!] \subseteq \mathcal{T}[\![S']\!]$	h, sat
2	$\mathcal{T}[\![S]\!] \lhd \mathcal{M}[\![S']\!] \subseteq \mathcal{M}[\![S]\!]$	h, sat
3	$\mathcal{T}[\![mk\text{-}While(B,S)]\!]$	$\mathcal{T}$
	$= fix(\lambda s \cdot \overline{\neg B} \cup ((\overline{B} \cap \mathcal{T}[\![S]\!]) - \mathrm{dom}(\mathcal{M}[\![S]\!] \rhd s)))$	
4	$\subseteq fix(\lambda s \cdot \overline{\neg B} \cup ((\overline{B} \cap \mathcal{T}[\![S']\!]) - \mathrm{dom}(\mathcal{M}[\![S']\!] \rhd s)))$	3,2,1
	infer $\subseteq \mathcal{T}[\![mk\text{-}While(B,S')]\!]$	4,$\mathcal{T}$

Figure 13: Proof of Lemma W2

	from $\mathcal{T}[\![S]\!] \lhd \mathcal{M}[\![S']\!] \subseteq \mathcal{M}[\![S]\!]$	
1	$\mathcal{T}[\![mk\text{-}While(B,S)]\!] \subseteq \overline{\neg B} \cup \mathcal{T}[\![S]\!]$	$\mathcal{T}$
2	$\overline{\neg B} \lhd I \subseteq I$	I
3	$(\mathcal{T}[\![S]\!] - \overline{\neg B}) \lhd \mathcal{M}[\![S']\!] \subseteq \mathcal{M}[\![S]\!]$	h
	infer $\mathcal{T}[\![mk\text{-}While(B,S)]\!] \lhd \mathcal{M}[\![mk\text{-}While(B,S')]\!]$	1,2,3,*fix*,$\mathcal{M}$
	$\subseteq \mathcal{M}[\![mk\text{-}While(B,S)]\!]$	

Figure 14: Proof of Lemma W3

	from S' sat S	
1	$\mathcal{T}[\![mk\text{-}While(B,S)]\!] \subseteq \mathcal{T}[\![mk\text{-}While(B,S')]\!]$	h,W2
2	$\mathcal{T}[\![S]\!] \lhd \mathcal{M}[\![S']\!] \subseteq \mathcal{M}[\![S]\!]$	h, sat
3	$\mathcal{T}[\![mk\text{-}While(B,S)]\!] \lhd \mathcal{M}[\![mk\text{-}While(B,S')]\!]$	2,W3
	$\subseteq \mathcal{M}[\![mk\text{-}While(B,S)]\!]$	
	infer $mk\text{-}While(B,S')$ sat $mk\text{-}While(B,S)$	sat ,1,3

Figure 15: Proof of Lemma W4

is proved in Figure 15.

The overall result that all of the language constructs are monotone in sat follows from C2, I2 and W4.

A.3 Justification of the Proof Obligations

The proof obligations for the various language constructs can now be justified with respect to the $\mathcal{T}/\mathcal{M}$ semantics which is given above. The link between the triples:

$$\{P\}S\{R\}$$

where:

$$P: \Sigma \to B$$
$$R: \Sigma \times \Sigma \to B$$

is given by[8]:

$$\{P\}S\{R\} \triangleq \overline{P} \subseteq \mathcal{T}[\![S]\!] \wedge \overline{P} \lhd \mathcal{M}[\![S]\!] \subseteq \overline{R}$$

Here, only the proof obligation for while is considered. The use of a transitive, well-founded relation R in that rule gives rise to a (complete) induction rule:

$$\text{R-ind} \qquad \frac{s \subseteq S;\ \mathsf{rng}(S \lhd R^+) \subseteq T \vdash s \subseteq T}{S \subseteq T}$$

This rule can be used to reason about $\mathcal{T}$ (where Scott induction is not applicable because of the lack of continuity). As in the complete induction rule over integers, the induction hypothesis is taken here to cover all (R^+) predecessors of the required set. To find the apparently ommitted base case for the inductive proof one must consider those elements with no predecessors ($s - \mathsf{dom}\, R$). It is however possible to avoid this case distinction in some proofs and the proof of Lemma W5 is given below is simpler than the corresponding proof in [26] for precisely this reason.

The result about the termination of the while is:

Lemma (W5)

$$\overline{P} \cap \overline{B} \subseteq \mathcal{T}[\![S]\!],\ (\overline{P} \cap \overline{B}) \lhd \mathcal{M}[\![S]\!] \subseteq \overline{R},\ \mathsf{rng}((\overline{P} \cap \overline{B}) \lhd \mathcal{M}[\![S]\!]) \subseteq \overline{P} \vdash \overline{P} \subseteq \mathcal{T}[\![mk\text{-}While(B,S)]\!]$$

The proof of this is given in Figure 16.

The proof about the meaning function be conducted with:

$$\mathcal{G}(r) \triangleq (\overline{\neg B} \lhd I) \cup (\overline{B} \lhd \mathcal{M}[\![S]\!]; r)$$

This is suitable for:

$$\text{Scott-ind} \qquad \frac{pr(\{\}),\ pr(r) \vdash pr(\mathcal{G}(r))}{pr(fix\ \lambda r \cdot \mathcal{G}(r))}$$

The proof of:

Lemma (W6)

$$\overline{P} \cap \overline{B} \subseteq \mathcal{T}[\![S]\!],\ (\overline{P} \cap \overline{B}) \lhd \mathcal{M}[\![S]\!] \subseteq \overline{R},\ \mathsf{rng}((\overline{P} \cap \overline{B}) \lhd \mathcal{M}[\![S]\!]) \subseteq \overline{P} \vdash \overline{P} \lhd \mathcal{M}[\![mk\text{-}While(B,S)]\!] \subseteq \overline{R}^* \wedge \mathsf{rng}(\overline{P} \lhd \mathcal{M}[\![mk\text{-}While(B,S)]\!] \subseteq \overline{P} \wedge \mathsf{rng}(\overline{P} \lhd \mathcal{M}[\![mk\text{-}While(B,S)]\!] \subseteq \overline{\neg B}$$

	from $\overline{P} \cap \overline{B} \subseteq \mathcal{T}[\![S]\!]$,	
	$(\overline{P} \cap \overline{B}) \lhd \mathcal{M}[\![S]\!] \subseteq \overline{R}$,	
	$\mathrm{rng}((\overline{P} \cap \overline{B}) \lhd \mathcal{M}[\![S]\!]) \subseteq \overline{P}$	
1	from $s \subseteq \overline{P}$,	
	$\mathrm{rng}(s \lhd R^+) \subseteq \mathcal{T}[\![mk\text{-}While(B,S)]\!]$	
1.1	$s \cap \overline{\neg B} \subseteq \mathcal{T}[\![mk\text{-}While(B,S)]\!]$	$\mathcal{T}$
1.2	$s \cap \overline{B} \subseteq \mathcal{T}[\![S]\!]$	h,h1
1.3	$(s \cap \overline{B}) \lhd \mathcal{M}[\![S]\!] \subseteq \overline{R}$	h,h1
1.4	$((s \cap \overline{B}) \lhd \mathcal{M}[\![S]\!]) \rhd \mathcal{T}[\![mk\text{-}While(B,S)]\!] = \{\}$	1.3,h1
1.5	$s \cap \overline{B} \subseteq \mathcal{T}[\![mk\text{-}While(B,S)]\!]$	1.4,$\mathcal{T}$
	infer $s \subseteq \mathcal{T}[\![mk\text{-}While(B,S)]\!]$	1.1,1.5
	infer $\overline{P} \subseteq \mathcal{T}[\![mk\text{-}While(B,S)]\!]$	R-ind(1)

Figure 16: Proof of Lemma W5

	from $\overline{P} \cap \overline{B} \subseteq \mathcal{T}[\![S]\!]$,	
	$(\overline{P} \cap \overline{B}) \lhd \mathcal{M}[\![S]\!] \subseteq \overline{R}$,	
	$\mathrm{rng}((\overline{P} \cap \overline{B}) \lhd \mathcal{M}[\![S]\!]) \subseteq \overline{P}$	
1	$\overline{P} \lhd \{\} \subseteq \overline{R}^*$	set
2	$\mathrm{rng}(\overline{P} \lhd \{\}) \subseteq \overline{P} \cup \overline{\neg B}$	set
3	from $\overline{P} \lhd r \subseteq \overline{R}^*$,	
	$\mathrm{rng}(\overline{P} \lhd r) \subseteq \overline{P}$,	
	$\mathrm{rng}(\overline{P} \lhd r) \subseteq \overline{\neg B}$	
3.1	$(\overline{P} \cap \overline{\neg B}) \lhd I \subseteq \overline{R}^*$	R^* is reflexive
3.2	$\mathrm{rng}((\overline{P} \cap \overline{\neg B} \lhd I) \subseteq \overline{P} \cup \overline{\neg B}$	set
3.3	$(\overline{P} \cap \overline{B}) \lhd \mathcal{M}[\![S]\!]; r \subseteq \overline{R}^*$	h3,R^* is transitive,h
3.4	$\mathrm{rng}((\overline{P} \cap \overline{B}) \lhd \mathcal{M}[\![S]\!]; r) \subseteq \overline{P} \cup \overline{\neg B}$	h3,R^* is transitive,h
	infer $\overline{P} \lhd \mathcal{G}(r) \subseteq \overline{R}^* \wedge$	$\mathcal{G}$,3.1,3.3,2.3.4
	$\mathrm{rng}(\overline{P} \lhd \mathcal{G}(r)) \subseteq \overline{P} \wedge$	
	$\mathrm{rng}(\overline{P} \lhd \mathcal{G}(r)) \subseteq \overline{\neg B}$	
	infer $\overline{P} \lhd \mathcal{M}[\![mk\text{-}While(B,S)]\!] \subseteq \overline{R}^* \wedge$	Scott-ind (1,2,3),M
	$\mathrm{rng}(\overline{P} \lhd \mathcal{M}[\![mk\text{-}While(B,S)]\!] \subseteq \overline{P} \wedge$	
	$\mathrm{rng}(\overline{P} \lhd \mathcal{M}[\![mk\text{-}While(B,S)]\!] \subseteq \overline{\neg B}$	

Figure 17: Proof of Lemma W6

is given in Figure 17.

These two lemmas combine immediately to prove:

Theorem (W7)[8]

The rule:

$$\text{while} \qquad \frac{\{P \wedge B\}\ S\ \{P \wedge R\}}{\{P\}\ \textsf{while}\ B\ \textsf{do}\ S\{P \wedge R^* \wedge \neg B\}}$$

is consistent with the $\mathcal{T}/\mathcal{M}$ semantics.

[8] Some care is needed in the interpretation of post-conditions which are conjunctions.

B Inference rules for LPF

Conventions

1. $E, E_1, \ldots$ denote logical expressions.
2. $x, y, \ldots$ denote variables over proper elements in a universe.
3. $c, c_1, \ldots$ denote constants over proper elements in a universe.
4. $s, s_1, \ldots$ denote terms which may contain partial functions.
5. $E(x)$ denotes a formula in which x occurs free.
6. $E(s/x)$ denotes a formula obtained by substituting all free occurrences of x by s in E. If a clash between free and bound variables would occur, suitable renaming is performed before the substitution.
7. $E[s_2/s_1]$ denotes a formula obtained by substituting some occurrences of s_1 by s_2. If a clash between free and bound variables would occur, then suitable renaming is performed before the substitution.
8. X is a non-empty set.
9. An "arbitrary" variable is one about which no results have been established.

General Properties

inf $\quad \dfrac{E_1 \vdash E_2;\ E_1}{E_2}$

var-I $\quad \dfrac{}{x^9 \in X}$

commutativity ($\vee$/ $\wedge$ /$\Leftrightarrow$-comm)

$$\frac{E_1 \vee E_2}{E_2 \vee E_1} \qquad \frac{E_1 \wedge E_2}{E_2 \wedge E_1} \qquad \frac{E_1 \Leftrightarrow E_2}{E_2 \Leftrightarrow E_1}$$

associativity ($\vee$/ $\wedge$ /$\Leftrightarrow$-ass)

$$\frac{(E_1 \vee E_2) \vee E_3}{E_1 \vee (E_2 \vee E_3)} \qquad \frac{(E_1 \wedge E_2) \wedge E_3}{E_1 \wedge (E_2 \wedge E_3)} \qquad \frac{(E_1 \Leftrightarrow E_2) \Leftrightarrow E_3}{E_1 \Leftrightarrow (E_2 \Leftrightarrow E_3)}$$

transitivity ($\Rightarrow$ /$\Leftrightarrow$-trans)

$$\frac{E_1 \Rightarrow E_2;\ E_2 \Rightarrow E_3}{E_1 \Rightarrow E_3} \qquad \frac{E_1 \Leftrightarrow E_2;\ E_2 \Leftrightarrow E_3}{E_1 \Leftrightarrow E_3}$$

substitution

$=$t-subs $\quad \dfrac{s_1 = s_2;\ E}{E[s_2/s_1]}$

[9] x is arbitrary

=v-subs $\dfrac{s \in X;\ x \in X \vdash E(x)}{E(s/x)}$

=-comm $\dfrac{s_1 = s_2}{s_2 = s_1}$

=-trans $\dfrac{s_1 = s_2;\ s_2 = s_3}{s_1 = s_3}$

$f: D \to R$

$f(d) \triangleq e$

$e_0 = e(d_0/d)$

$\triangleq$-subs $\dfrac{d_0 \in D;\ E(e_0)}{E[f(d_0)/e_0]}$

$\triangleq$-inst $\dfrac{d_0 \in D;\ E(f(d_0))}{E[e_0/f(d_0)]}$

$f(d) \triangleq$ if e then et else ef

if-subs $\dfrac{d_0 \in D;\ e_0;\ E(et_0)}{E[f(d_0)/et_0]}$ $\dfrac{d_0 \in D;\ \neg e_0;\ E(ef_0)}{E[f(d_0)/ef_0]}$

Definitions of Connectives

f-defn $\dfrac{\neg \text{true}}{\text{false}}$

$\wedge$-defn $\dfrac{\neg(\neg E_1 \vee \neg E_2)}{E_1 \wedge E_2}$

$\Rightarrow$-defn $\dfrac{\neg E_1 \vee E_2}{E_1 \Rightarrow E_2}$

$\Leftrightarrow$-defn $\dfrac{(E_1 \Rightarrow E_2) \wedge (E_2 \Rightarrow E_1)}{E_1 \Leftrightarrow E_2}$

$\forall$-defn $\dfrac{\neg \exists x \in X \cdot \neg E(x)}{\forall x \in X \cdot E(x)}$

Relationships between Operators

deM $\quad \dfrac{\neg(E_1 \vee E_2)}{\neg E_1 \wedge \neg E_2} \qquad \dfrac{\neg(E_1 \wedge E_2)}{\neg E_1 \vee \neg E_2}$

$$\frac{\neg \exists x \in X \cdot E(x)}{\forall x \in X \cdot \neg E(x)} \qquad \frac{\neg \forall x \in X \cdot E(x)}{\exists x \in X \cdot \neg E(x)}$$

dist $\quad \dfrac{E_1 \vee E_2 \wedge E_3}{(E_1 \vee E_2) \wedge (E_1 \vee E_3)} \qquad \dfrac{E_1 \wedge (E_2 \vee E_3)}{E_1 \wedge E_2 \vee E_1 \wedge E3}$

$\exists\vee$-dist $\quad \dfrac{\exists x \in X \cdot E_1(x) \vee E_2(x)}{(\exists x \in X \cdot E_1(x)) \vee (\exists x \in X \cdot E_2(x))}$

$\exists\wedge$-dist $\quad \dfrac{\exists x \in X \cdot E_1(x) \wedge E_2(x)}{(\exists x \in X \cdot E_1(x)) \wedge (\exists x \in X \cdot E_2(x))}$

$\forall\vee$-dist $\quad \dfrac{(\forall x \in X \cdot E_1(x)) \vee (\forall x \in X \cdot E_2(x))}{\forall x \in X \cdot E_1(x) \vee E_2(x)}$

$\forall\wedge$-dist $\quad \dfrac{(\forall x \in X \cdot E_1(x)) \wedge (\forall x \in X \cdot E_2(x))}{\forall x \in X \cdot E_1(x) \wedge E_2(x)}$

Substitution

$\wedge$-subs $\quad \dfrac{E_1 \wedge \ldots \wedge E_i \wedge \ldots \wedge E_n;\ E_i \vdash E}{E_1 \wedge \ldots \wedge E \wedge \ldots \wedge E_n}$

$\vee$-subs $\quad \dfrac{E_1 \vee \ldots \vee E_i \vee \ldots \vee E_n;\ E_i \vdash E}{E_1 \vee \ldots \vee E \vee \ldots \vee E_n}$

$\exists$-subs $\quad \dfrac{\exists x \in X \cdot E_1(x);\ E_1(x) \vdash E(x)}{\exists x \in X \cdot E(x)}$

contr $\quad \dfrac{E_1;\ \neg E_1}{E_2}$

$\Rightarrow$-contrp $\quad \dfrac{E_1 \Rightarrow E_2}{\neg E_2 \Rightarrow \neg E_1}$

INTRODUCTION(*op-I*) *ELIMINATION*(*op-E*)

$\neg\neg$ $\quad \dfrac{E}{\neg\neg E}$ $\qquad \dfrac{\neg\neg E}{E}$

$\vee$ $\quad \dfrac{E_i}{E_1 \vee E_2 \vee \ldots \vee E_n}$ $\qquad \dfrac{\begin{array}{c}E_1 \vee \ldots \vee E_n;\\ E_1 \vdash E;\ \ldots;\ E_n \vdash E\end{array}}{E}$

$\wedge$ $\quad \dfrac{E_1;\ E_2;\ \ldots;\ E_n}{E_1 \wedge E_2 \wedge \ldots \wedge E_n}$ $\qquad \dfrac{E_1 \wedge E_2 \wedge \ldots \wedge E_n}{E_i}$

$\neg\vee$ $\quad \dfrac{\neg E_1;\ \neg E_2;\ \ldots;\ \neg E_n}{\neg(E_1 \vee E_2 \vee \ldots \vee E_n)}$ $\qquad \dfrac{\neg(E_1 \vee E_2 \vee \ldots \vee E_n)}{\neg E_i}$

$\neg\wedge$ $\quad \dfrac{\neg E_i}{\neg(E_1 \wedge \ldots \wedge E_n)}$ $\qquad \dfrac{\begin{array}{c}\neg(E_1 \wedge \ldots \wedge E_n);\\ \neg E_1 \vdash E;\ \ldots;\ \neg E_n \vdash E\end{array}}{E}$

$\Rightarrow$ $\quad \dfrac{E_1 \vdash E_2;\ E_1 \in \mathrm{B}}{E_1 \Rightarrow E_2}$

vac $\Rightarrow$ $\quad \dfrac{E_2}{E_1 \Rightarrow E_2}$ $\qquad \dfrac{E_1 \Rightarrow E_2;\ \neg E_2}{\neg E_1}$

$\dfrac{\neg E1}{E_1 \Rightarrow E_2}$ $\qquad \dfrac{E_1 \Rightarrow E_2;\ E_1}{E_2}$

$\Leftrightarrow$ $\quad \dfrac{E_1 \wedge E_2}{E_1 \Leftrightarrow E_2}$ $\qquad \dfrac{E_1 \Leftrightarrow E_2}{E_1 \wedge E_2 \vee \neg E_1 \wedge \neg E_2}$

$\dfrac{\neg E_1 \wedge \neg E_2}{E_1 \Leftrightarrow E_2}$

$\neg\Leftrightarrow$ $\quad \dfrac{E_1 \wedge \neg E_2}{\neg(E_1 \Leftrightarrow E_2)}$ $\qquad \dfrac{\neg(E_1 \Leftrightarrow E_2)}{E_1 \wedge \neg E_2 \vee \neg E_1 \wedge E_2}$

$\dfrac{\neg E_1 \wedge E_2}{\neg(E_1 \Leftrightarrow E_2)}$

$\exists$ $\quad \dfrac{s \in X;\ E(s/x)}{\exists x \in X \cdot E(x)} \qquad \dfrac{\exists x \in X \cdot E(x);\ y^{10} \in X, E(y/x) \vdash E_1}{E_1}$

$\forall$ $\quad \dfrac{x^{11} \in X \vdash E(x)}{\forall x \in X \cdot E(x)} \qquad \dfrac{\forall x \in X \cdot E(x);\ s \in X}{E(s/x)}$

$\neg\exists$ $\quad \dfrac{x \in X \vdash \neg E(x)}{\neg\exists x \in X \cdot E(x)} \qquad \dfrac{\neg\exists x \in X \cdot E(x);\ s \in X}{\neg E(s/x)}$

$\neg\forall$ $\quad \dfrac{s \in X;\ \neg E(s/x)}{\neg\forall x \in X \cdot E(x)} \qquad \dfrac{\neg\forall x \in X \cdot E(x);\ y^{12} \in X, \neg E(y/x) \vdash E}{E}$

Miscellaneous

$\exists$split $\quad \dfrac{\exists x \in X \cdot E(x,x)}{\exists x, y \in X \cdot E(x,y)}$

$\forall$fix $\quad \dfrac{\forall x, y \in X \cdot E(x,y)}{\forall x \in X \cdot E(x,x)}$

$\forall \rightarrow \exists$ $\quad \dfrac{\forall x \in X^{13} \cdot E(x)}{\exists x \in X \cdot E(x)}$

$$\frac{\exists x \in X \cdot \forall y \in Y \cdot E(x,y)}{\forall y \in Y \cdot \exists x \in X \cdot E(x,y)}$$

$$\frac{\forall x \in X \cdot E_1(x) \ \Leftrightarrow\ E_2(x)}{(\forall x \in X \cdot E_1(x)) \ \Leftrightarrow\ (\forall x \in X \cdot E_2(x))}$$

$=$-contr $\quad \dfrac{\neg(s = s)}{E}$

$=$-term $\quad \dfrac{s \in X}{s = s}$

---

[10] y is arbitrary and not free in E_1
[11] x is arbitrary
[12] y is arbitrary and not free in E
[13] X is non-empty

=-comp $\dfrac{s_1, s_2 \in X}{(s_1 = s_2) \vee \neg(s_1 = s_2)}$

Δ-I $\dfrac{E}{\Delta E}$ $\qquad$ $\dfrac{\neg E}{\Delta E}$

Δ-E $\dfrac{\Delta E;\ E \vdash E_1;\ \neg E \vdash E_1}{E_1}$

$\neg\Delta$-I $\dfrac{\Delta E \vdash E_1;\ \Delta E \vdash \neg E_1}{\neg \Delta E}$

$\neg\Delta$-E $\dfrac{\neg\Delta E \vdash E_1;\ \neg\Delta E \vdash \neg E_1}{\Delta E}$

==-refl $\dfrac{}{s == s}$

==-subs $\dfrac{s_1 == s_2;\ E}{E[s_2/s_1]}$

==-comm $\dfrac{s_1 == s_2}{s_2 == s_1}$

==-trans $\dfrac{s_1 == s_2;\ s_2 == s_3}{s_1 == s_3}$

$==\rightarrow=$ $\dfrac{s_1 == s_2;\ s_i \in X}{s_1 = s_2}$

$=\rightarrow==$ $\dfrac{s_1 = s_2}{s_1 == s_2}$