stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE) IW 153/80 DECEMBER

R.J.R. BACK

ON THE NOTION OF CORRECT REFINEMENT OF PROGRAMS

Preprint

kruislaan 413 1098 SJ amsterdam

Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

1980 Mathematical subject classification: 68B10

ACM Computing Reviews: 5.24

On the notion of correct refinement of programs[*)]

by

R.J.R. Back[**)]

ABSTRACT

The stepwise refinement technique is studied from a mathematical point of view. A relation of correct refinement between programs is defined, based on the principle that refinement steps should be correctness preserving. Refinement between programs will therefore depend on the criterion of program correctness used. The application of the refinement relation in showing the soundness of different techniques for refining programs is discussed. Special attention is given to the use of abstraction in program construction. Refinement with respect to partial and total correctness will be studied in more detail, both for deterministic and nondeterministic programs. The relationship between these refinement relations and the approximation relation of fixpoint semantics will be studied, as well as the connection with the predicate transformers used in program verification.

KEY WORDS & PHRASES: Stepwise refinement, abstraction, top-down program construction, approximations, weakest preconditions, strongest postconditions, total correctness, partial correctness

---

*) This report will be submitted for publication elsewhere.

**) Present address: University of Helsinki, Computing Centre, Tukholmankatu 2, 00250 Helsinki 25

1. INTRODUCTION

Stepwise refinement is a well-known program construction technique, originally proposed by DIJKSTRA [9,10,11] and WIRTH [21,22]. The basic idea behind this technique is to develop a program trough a sequence of refinement steps, starting from a specification of the program and (hopefully) ending up with an efficient program meeting the specification. Our aim here is to study the correctness of such refinement steps. We take as our starting point the intuitive requirement that a refinement step must preserve program correctness. This requirement is implicit in the writings by Dijkstra and Wirth and is explicitly stated by GERHART [12]. This means that correct refinement will depend on the criterion of correctness used. A refinement step which preserves partial correctness will e.g not necessary be correct if we wish to preserve total correctness.

We will start by giving a simple example of program construction by stepwise refinement, in section 2. In section 3 we present a formal definition of correct refinement, considered as a binary relation between programs. Section 4 discusses the application of this notion of refinement in showing the soundness of certain familiar techniques for refining programs. This section is intended to motivate the refinement relation and relate it to more familiar aspects of program construction.

Our main interest here will be in the mathematical aspects of the refinement relation itself. We will therefore not be concerned with programming language issues, nor will we consider the proof theory of refinement (these topics are treated quite extensively in [1]). We choose to identify programs with their denotations, treating programs as state transformation functions. Correctness criterions will also be semantic entities, thus ignoring questions of provability and validity. In section 5 we study refinement of deterministic programs, with respect to partial and total correctness. We give a simple characterization of these relations in terms of the approximation ordering used in fixpoint semantics. In section 6 we study refinement between nondeterministic programs with respect to these same correctness criterions. Also in this case is there a simple connection between refinement and the approximation ordering. In section 7 we will show how to characterize these refinement relations with predicate transformers, thus providing a basis for proving refinement between programs. Finally, in section 8, we return to the techniques for constructing refinements, this time considering them with respect to the specific refinement relations defined in sections 5 and 6.

This article is a revised and considerably expanded version of a paper which originally appeared as [2]. It forms a semantic counterpart to the more syntactically and proof-theoretically oriented investigation of stepwise refinement described in [1].

2. AN EXAMPLE OF STEPWISE REFINEMENT

Before showing how to formalize the stepwise refinement technique, we give a simple example of how this technique is used in program construction (the example is taken from [11], where it is treated in a somewhat different way). Consider the following programming problem. We are to construct a program for computing X^Y, where X and Y are integers, $X \geq 1$ and $Y \geq 0$. We are only allowed to use simple arithmetic operations in the program, the exponentiation operation is e.g. not available.

The following is a more formal specification of the program to be constructed, in terms of pre- and postconditions:

Precondition: $X \geq 1$ and $Y \geq 0$.
Postcondition: $z = X^Y$.

Here z is a variable that is to contain the result of the computation.

A first solution can be constructed by introducing two auxiliary variables u and v and a program invariant R, R: $u^v * z = X^Y$ and $u \geq 1$ and $v \geq 0$.

We design a program P_1 in which R is established before entering its loop and is preserved by the loop:

```
P1:     begin   var u,v;
                u:= X; v:= Y; z:= 1;
                while v ≠ 0 do
                        "decrease v so that R is preserved"
                od
        end.
```

A simple implementation of "decrease v so that R is preserved" is by

```
v:= v-1; z:= z * x.
```

This would give us program P_2:

```
P2:     begin   var u,v;
                u:=X; v:= Y; z:= 1;
                while v ≠ 0 do
                        v:= v-1; z := z * x
                od
        end.
```

A more efficient version can be obtained from P_1 by noticing that when v is even, we can half v and square x, without destroying the invariant R. This gives us the refinement P_3:

```
P3:     begin   var u,v;
                u:= X; v:= Y; z:= 1;
                while v ≠ 0 do
                        while even(v) do
                                u:= u * u; v:= v div 2
                        od;
                        v:= v-1; z:= z * x
                od
        end.
```

An alternative form of this program, in which the nested loops are fused into one, is P_4:

```
P4:     begin   var u,v;
                u:= X; v:= Y; z:= 1;
                while v ≠ 0 do
                        if even(v)
                        then    u:= u * u; v:= v div 2
                        else    v:= v - 1; z:= z * x
                        fi
                od
        end.
```

The main features of the stepwise refinement technique are here illustrated. Thus program P_1 contains the abstract statement "decrease v so that R is preserved", which is not an executable statement but is found useful in developing the program. The programs P_2 and P_3 result from P_1 by replacing this abstract statement by a concrete executable statement. This illustrates the use of top-down development, where the original programming problem is decomposed into simpler programming problems with the help of abstract statements. The last version P_4 could again have been constructed from P_3 by applying a general program transformation rule to the iteration part of P_3. This would be a rule allowing nested loops to be fused into one loop, provided certain conditions are met.

Looking at these refinement steps, it is not at all evident whether they are in fact correct, or even what the criterion of correctness for refinements should be. The obvious choice of correctness criterion, requiring that the refined and refining program have exactly the same input-output behaviour, is clearly too restrictive in many cases. We would like to find the weakest possible criterion of correctness which still guarantees that the stepwise refinement technique is sound. This is the problem to be treated in the next section.

3. REFINEMENT BETWEEN PROGRAMS

Assume that a set Prog of _programs_ is given (Prog can be seen as a programming language or as the set of possible meanings of programs). A _correctness criterion_ for Prog is a tuple C = (Spec, sat), where Spec is a set of _specifications_ (a specification language or a set of meanings of specifications) and sat is a relation of _satisfaction_, sat $\subseteq$ Spec $\times$ Prog, S sat P holding if and only if specification S is satisfied by program P. A refinement step, leading from program P to program P′, is intuitively correct if P′ preserves the correctness of P. More precisely, P′ should satisfy any specification which P satisfies. This gives us the following definition of correct refinement:

DEFINITION 1: Let C = (Spec,sat) be a correctness criterion for Prog and let P and P′ be programs in Prog. Then P is said to be _(correctly) refined_ by P′ with respect to C, denoted P ref_C P′, if

$$\forall S \in \mathrm{Spec}(S \text{ sat } P \Rightarrow S \text{ sat } P').$$

This definition provides the basis for our study of the stepwise refinement technique. The first observation about this relation is that it is both reflexive and transitive, i.e. it is a preorder (the proof of this is trivial and is therefore omitted):

PROPOSITION 1: Refinement with respect to C is a preorder in Prog.

The <u>program construction problem</u>, relative to a correctness criterion C = (Spec,sat), can be formulated as follows: Given a specification S in Spec and a set A of <u>acceptable programs</u> in Prog, find a program P in A such that S sat P. With stepwise refinement, this problem is solved in the following way. First one constructs a program P_1 which satisfies specification S. Then we constructs a sequence of programs $P_2,\ldots,P_n$ such that each P_{i+1} is a refinement of P_i, i = 1,...,n-1 and P_n is in A. This gives us

$$S \text{ sat } P_1 \text{ ref}_C\ P_2 \ldots \text{ ref}_C\ P_n \in A.$$

This technique is <u>sound</u>, in the sense that the final program P_n will indeed be a solution to the programming problem. Thus, by transitivity, the above implies that

$$S \text{ sat } P_1 \text{ ref } P_n \in A$$

Using the definition of refinement we have that S' sat $P_1 \Rightarrow S'$ sat P_n for any S' in Spec. Choosing $S' = S$ gives us

$$S \text{ sat } P_n \in A,$$

i.e. P_n is a solution to the program construction problem.

As the above discussion shows, stepwise refinement can also be seen as a constructive technique for proving program correctness. This is in fact the original motivation for the stepwise refinement technique given by DIJKSTRA [9] (see [8] for another exposition of this idea). The refinement relation induces an equivalence relation between programs in the obvious way:

DEFINITION 2. Let $C = (Spec, sat)$ be a correctness criterion for Prog and let P and P' be two programs in Prog. Then P and P' are <u>equivalent</u> with respect to C, denoted $P \; eq_C \; P'$, if

$$\forall S \in Spec(S \; sat \; P \Leftrightarrow S \; sat \; P').$$

We obviously have that $P \; eq_C \; P'$ if and only if $P \; ref_C \; P'$ and $P' \; ref_C \; P$. Essentially $P \; eq_C \; P'$ says that P and P' are indistinguishable, as far as the correctness criterion C goes, i.e. one will be correct whenever the other is correct. An immediate consequence of the definition is the following fact:

PROPOSITION 2: Equivalence with respect to C is an equivalence relation in Prog.

4. CONSTRUCTING PROGRAM REFINEMENTS

The previous section introduced the notion of refinement as a relation between programs, but did not give any hints as to how one actually is to find a refinement of a given program. Here we will briefly discuss some techniques employed to this end, commenting on their soundness in light of the definition of correct refinement adopted above. Soundness will here mean that when a refinement P' of a program P is constructed with such a technique, $P \; ref_C \; P'$ will hold, for the chosen correctness criterion C.

Program transformation rules

One of the important approaches to constructing a refinement of a program consists in using a predefined set of program transformation rules. This approach has gained considerable success since its introduction by Burstall and Darlington [7] and is treated in e.g. [4,14 and 20], just to mention a few. A <u>program transformation rule</u> can be considered as a function

$$t:\mathrm{Prog} \to \mathrm{Prog},$$

which assigns to each program P in Prog a suitably transformed program t(P). For the correctness of such a rule, we give the following definition:

DEFINITION 3. The program transformation rule $t:\mathrm{Prog} \to \mathrm{Prog}$ is <u>correct</u> with respect to the correctness criterion C, if

$$\forall P \in \mathrm{Prog}(P \ \mathrm{ref}_C \ t(P)).$$

A program transformation rule t would usually not be defined for every possible program P in Prog, but only for a subset of programs satisfying certain conditions. We can model this by defining $t(P) = P$ for programs P which do not satisfy the conditions associated with t. As ref_C is reflexive, $P \ \mathrm{ref}_C \ t(P)$ will always hold for such programs P, so correctness is determined only by the value of t on programs which do satisfy the given conditions.

If t and t' are two correct program transformation rules, then their composition $t \circ t'$ is also a correct program transformation rule. This follows immediately from the definition of correctness for such rules and the transitivity of refinement. Therefore, any program derived from an initial program by a sequence of correct program transformation rules will be a refinement of the initial program. Use of correct program transformation rules is thus a sound technique for constructing refinements of given programs.

Selective refinement

Program transformation rules would not be very useful if we only were to apply them to a program as a whole. To make efficient use of these rules, one needs to apply a rule selectively to some specific component of a program. Consider a program P which contains a component P_1, i.e. $P = P[P_1]$. Applying a transformation t to this component yields $P' = P[t(P_1)]$, i.e. P' is constructed from P by replacing P_1 in P by $t(P_1)$. We obviously want this way of selectively refining a part of a program to be sound, i.e. P ref_C P' should hold. More generally, we require that

$$P[P_1] \ \mathrm{ref}_C \ P[P_2]$$

holds whenever P_1 ref_C P_2 holds.

Programs are usually built up from basic constructs such as assignment statements using different kinds of program constructors, such as composition, conditional statements, iteration and/or recursion. More abstractly, this means that Prog is not just a set but an algebra, generated by the constructors from the basic constructs (the constants of the algebra) in Prog. A program constructor g can be considered as a function

$$g:\mathrm{Prog}^n \to \mathrm{Prog},$$

yielding a new program $g(P_1,\ldots,P_n)$ from given programs $P_1,\ldots,P_n$. (More generally, Prog could be one of the sorts in a many-sorted algebra, which also would include other sorts necessary for the construction of programs. Also, a distinction between syntax and semantics should be made in this context. We take the simplistic view above in order not to be deflected from our main topic, the study of the refinement relation itself. We refer to [13] for more details on the algebraic approach to programs and to [1] for a precise treatment of selective refinement.)

A sufficient condition for the refinement step above to be sound is that each program constructor is monotone with respect to ref_C, i.e. if P_i ref_C P_i' for $i=1,\ldots,n$, then

$$g(P_1,\dots,P_n) \text{ ref}_C\ g(P_1',\dots,P_n'),$$

for any P_i, P'_i in Prog, $i = 1,\dots,n$.

DEFINITION 4. Let Prog be the set of programs generated by the constructors G from a set of basic constructs. Then Prog is said to <u>admit selective refinement</u> with respect to the correctness criterion C if each constructor in G is monotone with respect to ref_C.

Abstraction

The use of abstraction, in the form of abstract statements, is a characteristic feature of the stepwise refinement technique. Its use is emphasized in [1] and is also central to some of the work done on designing program development languages [4,15]. In section 2 we saw that using the abstract statement "decrease v so that R is preserved" in program P_1 makes it easier to find an initial solution to the programming problem. The use of this abstraction reduces the original problem to a simpler one, that of finding a program satisfying a specification corresponding to the abstract statement. The corresponding specification could e.g. be expressed as

precondition: $R \,\&\, v > 0$
postcondition: $R \,\&\, v < v'$

where R is the assertion defined in section 2 and v' refers to the initial value of v. In general, abstraction is used to decompose a given problem into a number of independent subproblems, with abstract statements serving as specifications of these subproblems.

Let C = (Spec, sat) be a correctness criterion for Prog. <u>Abstract statements</u> can be seen as a subset Spec* of Prog, such that there is a one-to-one correspondence between elements of Spec and Spec*. For a specification S in Spec, let S* denote the corresponding program in

Spec*. The satisfaction relation sat <u>induces</u> a corresponding relation sat* between Spec* and Prog, by

$$S^* \text{ sat}^* \ P \quad \text{iff} \quad S \text{ sat } P,$$

for every S in Spec and P in Prog. Let C* = (Spec*,sat*) be the corresponding correctness criterion. Then it is easy to see that

$$P \text{ ref}_C \ P' \quad \text{iff} \quad P \text{ ref}_{C^*} \ P',$$

for every P and P´ in Prog.

This construction gives us a correctness criterion C* = (Spec*, sat*) for Prog, where $\text{Spec}^* \subseteq \text{Prog}$. Satisfaction is now a relation between programs, in the same way as refinement, so we may ask for the relationship between these two relations. This is clarified by the following two simple observations. First, assume that sat* is transitive in Prog. Then

$$S \text{ sat}^* \ P \Rightarrow S \text{ ref}_{C^*} \ P$$

for any $S \in \text{Spec}^*$ and $P \in \text{Prog}$. To see this, assume that S sat* P and let S´ be an arbitrary specification such that S´ sat* S. By transitivity, this means that S´ sat* P, i.e. S ref_{C^*} P holds, as S´ was arbitrarily chosen.

Next, assume that sat* is reflexive in Spec, i.e. S sat* S holds for any $S \in \text{Spec}^*$. Then

$$S \text{ ref}_{C^*} \ P \Rightarrow S \text{ sat}^* \ P,$$

for any $S \in \text{Spec}^*$ and $P \in \text{Prog}$. This is also easy to see. Thus, assume that S ref_{C^*} P holds. By reflexivity, S sat* S holds, and by the definition of refinement this means that S sat* P also holds. Combining these two observations gives us the following result:

PROPOSITION 3: Let $C = (\text{Spec},\text{sat})$ be a correctness criterion for Prog, where Spec $\subseteq$ Prog. Then

$$\forall S \in \text{Spec}\ \ \forall P \in \text{Prog}.(S \text{ sat } P \Leftrightarrow S \text{ ref}_C\ P)$$

if and only if sat is reflexive in Spec and transitive in Prog.

The only if part of the proposition is a result of ref_C being reflexive and transitive in Prog. In the special case when Spec = Prog, we have that sat = ref_C if and only if sat is a preorder in Prog. This result shows that satisfaction, which is a relation between specifications and programs, is a special case of the more general relation of refinement between programs, when the assumptions stated above are fulfilled.

The above analysis leads up to the following definition.

DEFINITION 5. The set of programs Prog admits abstraction with respect to the correctness criterion $C = (\text{Spec},\text{sat})$, if there is a one-to-one correspondence between Spec and a subset Spec* of Prog, such that the induced satisfaction relation sat* is reflexive in Spec* and transitive in Prog.

When a programming language admits abstraction, there is only one kind of objects to consider, (abstract) programs, and only one relation to consider, the refinement relation. The specifications form a subset of the programs and the satisfaction relation is a restriction of the refinement relation. The stepwise refinement technique then simplifies to: Given an abstract program P_0 and a set A of acceptable programs find a sequence of programs $P_1,\ldots,P_n$ such that

$$P_0 \text{ ref } P_1 \text{ ref } \ldots \text{ ref } P_n \in A.$$

Top-down development

The top-down program development technique derives its strength from the use of abstraction in combination with selective refinement. Let P be a program of the form $P = P[S_1,\dots,S_n]$, i.e. P contains the abstract statements $S_1,\dots,S_n$ as components. With top-down development, we try to construct programs $P_1,\dots,P_n$ such that S_i sat P_i, for $i = 1,\dots,n$. These new programs may contain abstract statements as components in their turn. The abstract statements in P are then replaced with these new programs, giving a program $P' = P[P_1,\dots,P_n]$. Obviously P ref_C P′ will then hold, provided Prog admits selective refinement and abstraction, i.e. the top-down method is sound.

The above discussion should be sufficient to indicate some of the necessary conditions for a successful use of stepwise refinement in program construction. Essentially the programming language should admit selective refinement and abstraction, and the program transformation rules used should be correct in the sense defined above.

5. REFINEMENT OF DETERMINISTIC PROGRAMS

The previous sections discussed the refinement relation in abstracto, without any specific commitments to the choice of programs studied or to the correctness criterion used. From now on we will be more specific, studying the refinement relation for certain important choices of values for the parameters Prog, Spec and sat. In this section we consider deterministic programs with respect to the criterions of total and partial correctness. In the next section we extend the study to nondeterministic programs with respect to these same correctness criterions.

As already remarked in the introduction, we want to treat refinement independently of any specific choice of programming language. We achieve this by taking a semantic point of view, regarding programs as state transformations. For this purpose, let Σ_0 be a set of <u>proper states</u>, and let $\perp$ be a special element not occurring in Σ_0 (the undefined state). The set of states is $\Sigma = \Sigma_0 \cup \{\perp\}$. A state transformation on Σ is a function $f:\Sigma \rightarrow \Sigma$ such that $f(\perp) = \perp$. Let F_Σ be the set of state transformations on Σ.

Consider a program interpreted as a state transformation f in F_Σ. Then $f(\sigma) = \sigma'$, σ and σ' in Σ, means that the program, started in initial state σ will either terminate in the final state σ' (when $\sigma' \neq \perp$) or not terminate (when $\sigma' = \perp$). The association of a state transformation with a program described in a specific programming language is determined by a meaning function, giving the semantics of the programming language used (see e.g. DE BAKKER [3] or TENNENT [19] for details).

Total and partial correctness of a program is usually defined with respect to an entry and an exit condition. The entry condition describes the set of initial states for which the program is required to work properly, while the exit condition describes for each such initial states the set of final states allowed for the program. An entry - exit pair is thus a specification for a program. Partial and total correctness differ only in the way satisfaction of such a specification is defined. A specification will thus be a pair (D,R), where $D \subseteq \Sigma_0$ is the entry condition (the specified domain) and $R:D \rightarrow P(\Sigma_0)$ is the exit condition (the specified result). The set of all such (D,R) pairs is denoted H_Σ.

Let f be a state transformation in F_Σ and let (D,R) be a specification in H_Σ. Then f is said to be totally correct with respect to (D,R), denoted (D,R) tot f, if

$$\forall\sigma \in D.\ f(\sigma) \in R(\sigma).$$

As $\perp \notin R(\sigma)$, this implies that $f(\sigma) \neq \perp$ for each $\sigma \in D$. We say that f is partially correct with respect to (D,R), denoted (D,R) par f, if

$$\forall\sigma \in D.\ f(\sigma) \in R(\sigma) \cup \{\perp\}.$$

We now choose Prog = F_Σ, Spec = H_Σ and consider the two choices for sat, sat = tot and sat = par. This gives us two correctness criterions, total correctness $T = (H_\Sigma, \text{tot})$ and partial correctness $P = (H_\Sigma, \text{par})$. These correctness criterions determine two different notions of refinement between state transformations in F_Σ. By definition 1, f ref_T f′ if

$$\forall (D,R) \in H_\Sigma . [(D,R) \text{ tot } f \Rightarrow (D,R) \text{ tot } f']$$

and similarly for f ref_P f′. The relation ref_T will be called total refinement and the relation ref_P partial refinement.

To find a simpler characterization of these relations, we need the approximation ordering of fixpoint semantics, defined as follows. Let σ and σ' be two states in Σ. Then σ approximates σ', denoted $\sigma \sqsubseteq \sigma'$, if

$$\sigma = \bot \text{ or } \sigma = \sigma'.$$

Let f and f′ be two state transformations in F_Σ. Then f approximates f′, denoted $f \sqsubseteq f'$, if

$$\forall \sigma \in \Sigma .\ f(\sigma) \sqsubseteq f(\sigma').$$

Obviously we have that $f \sqsubseteq f'$ if and only if $f(\sigma) = \bot$ or $f(\sigma) = f'(\sigma)$ for every σ in Σ. We now have the following characterizations of total and partial refinement.

PROPOSITION 4. Let f and f′ be elements in F_Σ. Then

$$f \text{ ref}_T f' \text{ if and only if } f \sqsubseteq f'.$$

Proof: Assume first that f ref_T f′. Choose an arbitrary $\sigma \in \Sigma$ such that $f(\sigma) = \sigma' \neq \bot$. Then (D,R) tot f, where $D = \{\sigma\}$ and $R = \lambda \tau \in D.\ \{\sigma'\}$. By the definition of refinement, this gives (D,R) tot f', i.e. $f'(\sigma) \in R(\sigma) = \{\sigma'\}$. Thus $f(\sigma) = f'(\sigma)$, so we may conclude that $f \sqsubseteq f'$.

For the converse, assume that $f \sqsubseteq f'$. Let (D,R) be a specification such that (D,R) tot f. Then $f(\sigma) \neq \perp$ for $\sigma \in D$, so $f(\sigma) = f'(\sigma)$. Thus $f'(\sigma) \in R(\sigma)$ for $\sigma \in D$, i.e. (D,R) tot f'. We may thus conclude that f ref_T f'. []

PROPOSITION 5. Let f and f' be elements in F_Σ. Then

f ref_P f' if and only if $f' \sqsubseteq f$.

Proof: Assume first that f ref_P f'. Let (D,R) be a specification, where $D = \{\sigma\}$ and $R = \lambda\tau \in D.(\{f(\sigma)\} - \{\perp\})$. Then (D,R) par f. By assumption, this gives (D,R) par f', i.e. $f'(\sigma) \in R(\sigma) \cup \{\perp\}$, so $f'(\sigma) = f(\sigma)$ or $f'(\sigma) = \perp$. We thus conclude that $f' \sqsubseteq f$.

For the converse, assume that $f' \sqsubseteq f$. Let (D,R) be a specification such that (D,R) par f. This means that $f(\sigma) \in R(\sigma) \cup \{\perp\}$ for each $\sigma \in D$. Now $f'(\sigma) = f(\sigma)$ or $f'(\sigma) = \perp$ by assumption. In both cases do we have that $f'(\sigma) \in R(\sigma) \cup \{\perp\}$ for $\sigma \in D$, i.e. (D,R) par f'. We may thus conclude that f ref_P f' holds. [].

These characterizations show a nice and somewhat surprising connection between the proof theoretically motivated refinement relations and the information increasing approximation relation used in Scott-like definitions of programming language semantics.

Some immediate consequences of the two propositions are worth mentioning. First, total and partial refinement are each others inverses, i.e. f ref_T f' if and only if f' ref_P f. Intuitively, total refinement only allows the domain of termination to be increased while partial refinement only allows this domain to be decreased. Another consequence is that equivalence with respect to total and partial correctness (<u>total</u>

and <u>partial equivalence</u>) coincide, both being reduced to functional equality, i.e.

$$f\ eq_T\ f' \text{ iff } f\ eq_P\ f' \text{ iff } f = f'.$$

We could also have made some other choices for the set of specifications. One possible choice would be to take as specifications all pairs (D,R), where D and R would both be subsets of Σ_0. However, an inspection of the proofs above show that this choice of specifications would yield the same characterization of total and partial refinement (this observation is due to G. Plotkin). Another possibility would be to choose only deterministic specifications, i.e. pairs (D,R) where $R(\sigma)$ is a singleton for each $\sigma \in D$. This would also yield the same characterization of refinement (in the case of partial refinement, we would need to assume that there are at least two elements in Σ_0.)

A more serious change would be to choose Spec to be a subset of H_Σ rather than H_Σ itself. This choice would be more realistic in some sense, as it would correspond to a situation in which a fixed specification language is given and Spec is the set of meanings of specifications in this language. Not all pairs (D,R) would then necessarily be expressible in the language, so Spec would be a proper subset of H_Σ. In this case we would not usually get the same characterizations of total and partial refinement (this would depend on the subsets of H_Σ chosen). However, total and partial refinement, as we define it above, would still be unique, in that they would be the strongest refinement relation for the respective correctness criterions. In other words, for any C = (Spec,tot), with Spec a subset of H_Σ, we would have that

$$f\ ref_T\ f' \Rightarrow f\ ref_C\ f'$$

for every f and f' in F_Σ (similarly for partial refinement). With respect to equivalence, this means that total (partial) equivalence gives the finest possible partitioning of the state transformations with respect to total (partial) correctness.

6. REFINEMENT OF NONDETERMINISTIC PROGRAMS

Let us now turn to nondeterministic state transformations. These are functions of the form $f:\Sigma \to P(\Sigma)$, where $f(\sigma) \neq \emptyset$ for each $\sigma \in \Sigma$ and $f(\perp) = \{\perp\}$. We denote the set of all nondeterministic state transformations by G_Σ. If $f \in G_\Sigma$ is the interpretation of a nondeterministic program, then $f(\sigma) = W$, W a subset of Σ, means that if the program is started in the initial state σ, each state in W will be a possible final state of the program. If W contains $\perp$, then it is also possible that the program will not terminate for this initial state.

Total and partial correctness will again be specified with respect to an entry and an exit condition. We use the same set H_Σ for specifications as we used in the deterministic case. However, the satisfaction relations have to be redefined. Let f be an element in G_Σ and let (D,R) be a specification in H_Σ. Then f is said to be _totally correct_ with respect to (D,R), denoted (D,R) tot f, if

$$\forall \sigma \in D.\ f(\sigma) \subseteq R(\sigma)$$

As $\perp \notin R(\sigma)$, this implies that $\perp \notin f(\sigma)$, for each $\sigma \in D$. We say that f is _partially correct_ with respect to (D,R), denoted (D,R) par f, if

$$\forall \sigma \in D.\ f(\sigma) \subseteq R(\sigma) \cup \{\perp\}.$$

This gives us the _total correctness_ criterion $T = (H_\Sigma, \text{tot})$ and the _partial correctness_ criterion $P = (H_\Sigma, \text{par})$ for G_Σ. We use the same notations here as in the previous section, because in the deterministic case, when $f(\sigma)$ is a singleton for each $\sigma \in \Sigma$, the relations tot and par agree with the previously defined satisfaction relations. The correctness criterions T and P determine corresponding refinement relations ref_T and ref_P, by definition 1. To find simpler characterizations of these, we again turn to the approximation ordering.

The approximation ordering used between nondeterministic state transformations is the so-called <u>Egli-Milner ordering</u> [16]. Let the approximation ordering between elements of Σ be defined as before. We define the following two relations between nonempty subsets W and W$'$ of Σ:

$$W \sqsubseteq_1 W' \text{ iff } \forall\sigma \in W\ \exists\sigma' \in W'.\ \sigma \sqsubseteq \sigma' \quad \text{and}$$

$$W \sqsubseteq_2 W' \text{ iff } \forall\sigma' \in W'\ \exists\sigma \in W.\ \sigma \sqsubseteq \sigma'.$$

An alternative characterization of these relations is

$$W \sqsubseteq_1 W' \text{ iff } W \subseteq W' \cup \{\bot\} \quad \text{and}$$

$$W \sqsubseteq_2 W' \text{ iff } \bot \in W \text{ or } W \supseteq W'.$$

The Egli-Milner approximation ordering is then defined by

$$W \sqsubseteq W' \text{ iff } W \sqsubseteq_1 W' \text{ and } W \sqsubseteq_2 W'.$$

An alternative characterization is

$$\begin{aligned} W \sqsubseteq W' \text{ iff } & \text{ either } \bot \in W \text{ and } W \subseteq W' \cup \{\bot\} \\ & \text{ or } \quad \bot \notin W \text{ and } W = W'. \end{aligned}$$

These relations are extended to nondeterministic state transformations in the usual way, i.e. for f and f$'$ elements of G_Σ, we define

$$f \propto f' \text{ iff } \forall\sigma \in \Sigma.\ f(\sigma) \propto f'(\sigma),$$

where $\propto$ is any one of the relations $\sqsubseteq_1$, $\sqsubseteq_2$ or $\sqsubseteq$.

A denotational semantics for nondeterministic programming languages can now be defined using the Egli-Milner ordering. (Actually the relation $_2$ alone can be used in defining such a semantics, as shown by SMYTH [18]). We are now able to characterize the refinement relations introduced above in terms of these approximation relations. We have the following results.

PROPOSITION 6. Let f and f′ be elements of G_Σ. Then

$$f \ \mathrm{ref}_T \ f' \quad \text{iff} \quad f \sqsubseteq_2 f'.$$

Proof: Assume first that $f \ \mathrm{ref}_T \ f'$. Let $\sigma \in \Sigma$ be such that $\perp \notin f(\sigma)$. Let $D = \{\sigma\}$ and $R = \lambda\tau \in D.\ f(\sigma)$. Then (D,R) tot f. The assumption gives us that (D,R) tot f′, i.e. $f'(\sigma) \subseteq f(\sigma)$. Thus we may conclude that $f \sqsubseteq_2 f'$.

For the converse, assume that $f \sqsubseteq_2 f'$. Let (D,R) be a specification such that (D,R) tot f. Thus $\perp \notin f(\sigma)$ for $\sigma \in D$. This means that $f'(\sigma) \subseteq f(\sigma) \subseteq R(\sigma)$, by the assumption, so (D,R) tot f′. Thus $f \ \mathrm{ref}_T \ f'$ holds. []

PROPOSITION 7. Let f and f′ be elements of G_Σ. Then

$$f \ \mathrm{ref}_P \ f' \quad \text{iff} \quad f' \sqsubseteq_1 f.$$

Proof: Assume first that $f \ \mathrm{ref}_P \ f'$ holds. Let σ be an element of Σ. Let $D = \{\sigma\}$ and $R = \lambda\tau \in D.(f(\sigma) - \{\perp\})$. Then (D,R) par f. Using the assumption we get (D,R) par f′, i.e. $f'(\sigma) \subseteq R(\sigma) \cup \{\perp\} = f(\sigma) \cup \{\perp\}$. Thus $f' \sqsubseteq_1 f$.

For the converse, assume that $f' \sqsubseteq f$. Assume that (D,R) par f. Then $f(\sigma) \subseteq R(\sigma) \cup \{\perp\}$, for every $\sigma \in D$. This means that $f(\sigma) \cup \{\perp\} \subseteq R(\sigma) \cup \{\perp\}$ and as by assumption $f'(\sigma) \subseteq f(\sigma) \cup \{\perp\}$, we have that (D,R) par f′, i.e. $f \ \mathrm{ref}_P \ f'$. []

Combining these two observations also gives us a characterization of the Egli-Milner ordering in terms refinement relations:

PROPOSITION 8. Let f and f′ be elements in G_Σ. Then

$$f \sqsubseteq f' \text{ iff } f \text{ ref}_T\ f' \text{ and } f' \text{ ref}_P\ f.$$

This again shows the close connection between the proof theoretically motivated refinement relations and the approximation ordering of fixpoint semantics.

Both relations $\sqsubseteq_1$ and $\sqsubseteq_2$ are preorders, but neither one is a partial order, i.e. neither one is antisymmetric. The equivalence relations induced by these can be characterized as follows. Let W and W′ be two nonempty subsets of Σ. Then

$$W \equiv_1 W' \text{ iff } W \cup \{\perp\} = W' \cup \{\perp\} \text{ and}$$

$$W \equiv_2 W' \text{ iff } \perp \in W \cap W' \text{ or } W = W'.$$

Equivalence between state transformations is defined in the same way as approximation, taking $\propto$ above to be $\equiv_1$ or $\equiv_2$. These relations are also the equivalence relations with respect to total and partial correctness. Thus

$$f \text{ eq}_T\ f' \text{ iff } f \equiv_2 f' \text{ and}$$

$$f \text{ eq}_P\ f' \text{ iff } f \equiv_1 f'.$$

There are also other relations which can be defined between nondeterministic state transformations (see e.g. [5]). The relations ref_P and eq_P are actually included in the list of interesting relations between programs presented in [6], although ref_T and eq_T seem to be missing.

7. REFINEMENT AND PREDICATE TRANSFORMERS

In order to give proof rules by which refinement between programs can be shown, we need to connect the refinement relation to more familiar concepts in program correctness. We will therefore show here how to characterize total and partial refinement in terms of predicate transformers. The characterization of total refinement in terms of weakest preconditions forms the basis for a general proof rule for total refinement, studied in detail in [1]. A similar proof rule can also be given for partial refinement.

Let Q be a subset of Σ_0 and let f be an element of G_Σ. The <u>weakest precondition</u> of f for Q, denoted wp(f,Q), is defined as

$$wp(f,Q) = \{\sigma \in \Sigma_0 \mid f(\sigma) \subseteq Q\}.$$

The definition implies that $f(\sigma)$ does not contain $\perp$ when $\sigma \in wp(f,Q)$. A characterization of total refinement can be given in terms of weakest preconditions, as follows:

PROPOSITION 9. Let f and f′ be elements of G_Σ. Then

$$f \text{ ref}_T \ f' \quad \text{iff} \quad \forall Q \subseteq \Sigma_0 .\ wp(f,Q) \subseteq wp(f',Q)$$

Proof: Assume first that f ref_T f′ holds. Let $Q \subseteq \Sigma_0$ and let $\sigma \in wp(f,Q)$. This means that $\perp \notin f(\sigma)$ so using the assumption and proposition 6 we have that $f'(\sigma) \subseteq f(\sigma) \subseteq Q$. Thus $\sigma \in wp(f',Q)$.

For the converse, assume that $wp(f,Q) \subseteq wp(f',Q)$ for any $Q \subseteq \Sigma_0$. Let σ be an element of Σ_0 such that $\perp \notin f(\sigma)$. Choose $Q = f(\sigma)$. Then $\sigma \in wp(f,Q)$, so by the assumption, $\sigma \in wp(f',Q)$, i.e. $f'(\sigma) \subseteq Q = f(\sigma)$. This means that $f \sqsubseteq_2 f'$ and so, by proposition 6, that f ref_T f′. []

This connection between the weakest preconditions and the Smyth ordering (relation $\sqsubseteq_2$ or ref_T) was independently found by Plotkin. A detailed discussion of the consequences of this, in showing the isomorphism between a predicate transformer semantics and a semantic based on nondeterministic state transformations is given in [17].

Let f again be an element of G_Σ and let $Q \subseteq \Sigma_0$. The <u>strongest postcondition</u> of f for Q, denoted sp(f,Q), is defined as

$$sp(f,Q) = \{\sigma' \in \Sigma_0 \mid \sigma' \in f(\sigma) \text{ for some } \sigma \in Q\}.$$

This gives us the following characterization of partial refinement:

PROPOSITION 10. Let f and f′ be elements of G_Σ. Then

$$f \ \mathrm{ref}_P \ f' \quad \text{iff} \quad \forall Q \subseteq \Sigma_0 .\ sp(f',Q) \subseteq sp(f,Q).$$

Proof: Assume first that f ref_P f′. Let $Q \subseteq \Sigma_0$ and let $\sigma' \in sp(f',Q)$, i.e. for some $\sigma \in Q$, $\sigma' \in f'(\sigma)$. By proposition 7, we then have that $f'(\sigma) \subseteq f(\sigma) \cup \{\perp\}$ and as $\sigma \neq \perp$, $\sigma' \in f(\sigma)$. Thus $\sigma' \in sp(f,Q)$.

For the converse, assume that $sp(f',Q) \subseteq sp(f,Q)$, for any $Q \subseteq \Sigma_0$. Let $Q = \{\sigma\}$ and assume that $\sigma' \in f'(\sigma)$. If $\sigma' \neq \perp$, this means that $\sigma' \in sp(f',Q)$, so by the assumption, $\sigma' \in sp(f,Q)$, i.e. $\sigma' \in f(\sigma)$. Thus $\sigma' \in f(\sigma) \cup \{\perp\}$. Using proposition 7, this gives that f ref_P f′. []

The importance of these results rests on the fact that the weakest preconditions and strongest postconditions can be computed syntactically for given programs, at least in the case of simple iterative programs. This gives us a syntactic characterization of refinement, on which a proof theory can be built.

8. PROGRAM CONSTRUCTION WITH TOTAL AND PARTIAL REFINEMENT

In this final section we will study the techniques for constructing refinements of programs described in section 4, with respect to the total and partial refinement relations defined in the previous sections. We consider first total and partial refinement in the case of deterministic programs.

There will be two kinds of correctness preserving transformation rules, depending on whether we wish to preserve total or partial correctness. In the first case, a transformation rule $t: F_\Sigma \to F_\Sigma$ will be correct if

$$\forall f \in F_\Sigma \,.\, f \sqsubseteq t(f)$$

while in the second case we require that

$$\forall f \in F_\Sigma .\, t(f) \sqsubseteq f.$$

In other words, in the first case a transformation rule is only allowed to increase the domain of termination, while in the second case it only is allowed to decrease the domain of termination.

These criterions for correctness of program transformation rules are not really new. The first criterion has e.g. been formulated by WEGBREIT [20], while the second criterion can be found in e.g. Burstall´s and Darlington´s article [7].

The requirement that Prog admits selective refinement boils down to requiring that the program constructors used in defining the denotational semantics of programming languages with composition, conditionals and iteration are monotone with respect to the approximation relation. As this is one of the basic requirements of this approach to semantics, simple programming languages of this kind will admit selective refinement, both with respect to total and partial correctness.

To analyze abstraction, let us denote by H_Σ' the set of deterministic specifications, i.e. specifications of the form (D,R), where $R(\sigma)$ is a singleton for each $\sigma \in D$. Let tot′ (par′) be the restriction of tot (par) to deterministic specifications. Let $T' = (H_\Sigma'$, tot′) and $P' = (H_\Sigma'$, par′). As remarked in section 5, the refinement relation determined by T' and P' are the same as those determined by T and P. We now have the following two results.

PROPOSITION 11. F_Σ admits abstraction with respect to T'.

Proof: For each deterministic specification (D,R), define a corresponding state transformation $(D,R)^*$ in F_Σ, by

$$(D,R)^*(\sigma) = \begin{cases} \sigma' & \text{when } \sigma \in D \text{ and } R(\sigma) = \{\sigma'\} \\ \perp & \text{otherwise} \end{cases}$$

This gives a one-to-one correspondence between deterministic specifications and deterministic state transformations. The induced satisfaction relation tot* is defined by

$$(D,R)^* \text{ tot}^* f \text{ iff } (D,R) \text{ tot}' f.$$

The relation tot* is easily shown to be both reflexive and transitive. Thus F_Σ admits abstraction with respect to T'. []

Thus, if we restrict ourselves to deterministic programs and deterministic specifications only, the approximation relation serves both as the relation of satisfaction and as the relation of refinement in program construction, provided we are interested in establishing and preserving total correctness of programs.

The situation with respect to partial correctness of deterministic programs is quite different. With the same embedding of deterministic specifications into F_Σ, the induced satisfaction relation par* turns out not to be transitive. In fact, the following proposition shows that, except for trivial cases, partial correctness and abstraction cannot be combined.

PROPOSITION 12. F_Σ does not admit abstraction with respect to P', if there are at least two elements in Σ_0.

Proof: Assume that there was an embedding of H_Σ' into F_Σ such that the induced satisfaction relation par*, defined by

$$(D,R)^* \text{ par}^* f \text{ iff } (D,R) \text{ par}' f,$$

was reflexive in the image of H_Σ' and transitive in F_Σ. Let us chose $D = \emptyset$ and let R be the empty function. Then (D,R) par$'$ f holds for any f in F_Σ. Consequently, $(D,R)^*$ par* f must hold for any f in F_Σ too. Now choose f and f' in F_Σ such that $f(\sigma)$, $f'(\sigma) \neq \perp$ and also $f(\sigma) \neq f'(\sigma)$ (this is possible, as Σ_0 is assumed to have at least two elements). There can obviously be no element f'' in F_Σ such that $f \sqsubseteq f''$ and $f' \sqsubseteq f''$. However, by assumption $(D,R)^*$ par* f and $(D,R)^*$ par* f'. Using proposition 3, this gives us that $f \sqsubseteq (D,R)^*$ and $f' \sqsubseteq (D,R)^*$. This is a contradiction, so there can be no element $(D,R)^*$ in F_Σ which corresponds to the chosen specification (D,R), i.e. F_Σ does not admit abstraction with respect to P'. [ ]

Next we consider the nondeterministic case. As with deterministic programs, we have two different notions of correctness of program transformation rules, depending on whether we wish to preserve total or partial correctness of programs. In the first case, we require that a program transformation rule $t:F_\Sigma \to F_\Sigma$ satisfies

$$\forall f \in F_\Sigma .\ f \sqsubseteq_2 t(f)$$

while in the second case we require

$$\forall f \in F_\Sigma .\ t(f) \sqsubseteq_1 f.$$

The requirement that Prog admits decomposition means in the nondeterministic case that the usual program constructors, such as composition, conditionals and iteration should be monotone with respect to the preorders $\sqsubseteq_1$ and $\sqsubseteq_2$. That this in fact is the case for the preorder $\sqsubseteq_2$ follows from results proved in BACK [1]. These program constructors can also be shown to be monotone with respect to the preorder $\sqsubseteq_1$.

To study abstraction, let H_Σ'' be the set of specifications which are satisfiable with respect to total correctness. This is the set of all pairs (D,R) such that $R(\sigma) \neq \emptyset$ for each $\sigma \in D$. Let tot" be the restriction of tot to this set of specifications, and let T" = (H_Σ'', tot"). The refinement relation determined by T" is the same as the refinement relation determined by T, as unsatisfiable specifications cannot affect the refinement relation. We have the following result.

PROPOSITION 13. G_Σ admits abstraction with respect to T".

Proof: Let the embedding of the satisfiable specifications into G_Σ be given as follows. For each (D,R) in H_Σ'', define

$$(D,R)^*(\sigma) = \begin{cases} R(\sigma) & \text{when } \sigma \in D \\ \{\perp\} & \text{otherwise} \end{cases}$$

The induced satisfaction relation tot*, defined by

$$(D,R)^* \text{ tot}^* f \text{ iff } (D,R) \text{ tot}'' f$$

is then easily shown to be reflexive in the image of H_Σ'' and transitive in G_Σ, thus proving the proposition.[]

Abstraction can be combined with partial correctness in the nondeterministic case, as shown by the following result.

PROPOSITION 14. G_Σ admits abstraction with respect to P.

Proof: In this case we choose the embedding as follows. Let (D,R) be a specification in H_Σ. Define the corresponding state transformation in G_Σ by

$$(D,R)^*(\sigma) = \begin{cases} R(\sigma) \cup \{\perp\}, \text{ if } \sigma \in D \\ \Sigma \quad \text{otherwise} \end{cases}$$

Also in this case is the induced satisfaction relation par* easily shown to be reflexive in in the image of H_Σ and transitive in G_Σ, thus proving the proposition. []

ACKNOWLEDGEMENTS. I would like to thank Gordon Plotkin for his careful reading of a previous version of this paper and for the many important comments he made, and Lambert Meertens for clarifying discussions on the topics treated here.

REFERENCES

[1] BACK, R.J.R., Correctness preserving program refinements: Proof theory and applications. Mathematical Center Tracts 131, Mathematisch Centrum, Amsterdam, 1980.

[2] BACK,R.J.R., On the notion of correct refinement of programs. In Proc. 5th Scandinavian Logic Symposium (F.V. Jensen, B.H. Mayoh, K.K. Moller, eds.), Aalborg University Press, 1979

[3] DE BAKKER, J.W., Mathematical Theory of Program Correctness. Prentice-Hall, 1980.

[4] BAUER, F.L., M. BROY, H. PARTSCH, P. PEPPER & H. WOSSNER, Systematics of transformation rules. In Program Construction (F.L. Bauer & M. Broy, eds.), pp.273-289, Lecture Notes in Computer Science 69, Springer-Verlag, 1979.

[5] BROY, M., R. GNATZ & M. WIRSING, Semantics of nondeterministic and noncontinuous constructs. In Program Construction (F.L. Bauer & M. Broy, eds.), pp. 553-592, Lecture Notes in Computer Science 69, Springer-Verlag, 1979.

[6] BROY, M., P. PEPPER & M. WIRSING, On relations between programs. In International Symposium on Programming (B. Robinet, ed.), pp. 59-78, Lecture Notes in Computer Science 83, Springer-Verlag 1980.

[7] BURSTALL,R.M. & J.DARLINGTON, Some transformations for developing recursive programs. Journal of ACM 24, 1, pp. 44 -67, 1977.

[8] CORRELL, C.H., Proving programs correct trough refinement. Acta Informatica 9, pp. 121-132, 1978.

[9] DIJKSTRA, E.W., A constructive approach to the problem of program correctness. BIT 8, pp. 174-186, 1968.

[10] DIJKSTRA, E.W., Notes on structured programming. In Dahl, O.J., E.W. Dijkstra & C.A.R. Hoare: Structured Programming, Academic Press, 1971.

[11] DIJKSTRA, E.W. A Discipline of Programming. Prentice-Hall, 1976.

[12] GERHART, S.L., Correctness preserving program transformations. In Proc. Second ACM Conference on Principles of Programming Languages, pp. 54-66, 1975.

[13] GOGUEN, J.A., J.W. THATCHER, E.G. WAGNER & J.B. WRIGHT, Initial algebra semantics and continuous algebras. Journal of ACM 24 (1977) 68-95.

[14] LOVEMAN, D.B., Program improvement by source-to-source transformations. Journal of ACM 24, 1, pp.121-145, 1977.

[15] MEERTENS, L.G.L.T., Abstracto 84: The next generation. Report IW 120/79, Mathematisch Centrum, 1979.

[16] PLOTKIN, G.D., A power domain construction. SIAM Journal of Computing 5, 3, pp. 452-487, 1976.

[17] PLOTKIN, G.D., Dijkstra's weakest preconditions and Smyth's power domains. In Abstract Software Specifications (D. Bjorner, ed.), Lecture Notes on Computer Science 86, Springer-Verlag 1979.

[18] SMYTH, M.B., Power domains. Journal of Computer & System Sciences 16, pp. 23-36, 1978.

[19] TENNENT, R.D., The denotational semantics of programming languages. Comm. ACM, vol. 19, no. 8, 1976, pp. 437-452.

[20] WEGBREIT, B., Goal directed program transformations. IEEE Trans. on Software Engineering SE-2, 2, pp. 69-80, 1976.

[21] WIRTH, N., Program development by stepwise refinement. Comm. ACM 14, 4, pp.221-227, 1971.

[22] WIRTH, N., Systematic Programming. Prentice-Hall, 1973.