Programming Concepts and Methods
M. Broy & C.B. Jones (Editors)
Elsevier Science Publishers B.V. (North-Holland)

On the refinement of non-deterministic recursive routines by transformations

Rudolf Berghammer

Fakultät für Informatik, Universität der Bundeswehr München
Werner-Heisenberg-Weg 39, D-8014 Neubiberg, Fed. Rep. Germany

Herbert Ehler and Bernhard Möller

Institut für Informatik, Technische Universität München
Postfach 20 24 20, Arcisstraße 21, D-8000 München, Fed. Rep. Germany

In this paper we prove a theorem which can be seen as an analogue of Park's well-known theorem in terms of the erratic refinement relation (viz. McCarthy's descendant order). Based on this theorem we demonstrate that the common method of program development by unfold/fold can also be applied in the case of a language with erratic non-determinism and a transformation notion which allows reducing non-determinacy. The proof of the corresponding transformation rule is given in such a way that it is largely independent of particular language features. Rather we give a number of simple requirements on the language which are sufficient to ensure that the analogue of Park's theorem holds.

1. Introduction

Experience has shown that the traditional way of programming is not appropriate for coping with the complex task of writing efficient and correct programs. As a consequence, efforts have been made to replace it by formal programming methods. One of the proposed methods is program development by transformations (see [Feather 87] or [Bauer et al. 89] for a survey). In this approach, the construction of a program is done incrementally by successive applications of semantics-preserving rules.

Within transformational programming the unfold/fold-method (see [Burstall Darlington 77]) is frequently used, e.g., for transforming a non-recursive specification into a first (terminating) recursive solution. In the case of deterministic programs its correctness follows from the simple fact that the semantics of the specification is a fixed point of the functional corresponding to the recursive solution. However, for methodological reasons, the programming language under consideration should also contain non-determinism in order to enable simple specifications of "naturally" ambiguous tasks and to allow the postponing of design decisions. Furthermore, non-determinism is important in connection with parallelism, since certain parallel constructs can transformationally be reduced to non-deterministic sequential constructs (cf. [Broy 80]). If

transformations are only allowed between programs having the same set of possible outcomes, the above correctness argument holds for non-deterministic programs, too. (This approach is e.g., taken in [Furbach 83].) However, it may fail if a transformation notion is used that also allows the reduction of non-determinacy. (If one transformation step strictly reduces non-determinacy, then the semantics of the specification need *not* be a fixed point of the functional corresponding to the recursive solution.) Nevertheless, the unfold/fold-method remains correct provided some very "natural" conditions are fulfilled. This is shown in the paper.

In the literature, one finds three different kinds of non-determinism, viz. angelic, demonic, and erratic non-determinism. Angelic non-determinism corresponds to a "prophetic" choice during computation: Possible termination is equivalent to guaranteed termination. It is useful in those cases where only partial correctness issues are of primary concern. The demonic non-determinism is underlying Dijkstra's wp-calculus. Here possible non-termination is equivalent to guaranteed non-termination. It serves as the basis for refinement calculi at the procedural level, see [Back 88], [Morgan et al. 88], and [Morris 87]. Finally, the erratic non-determinism (also called choice non-determinism) arbitrarily chooses any of the possible computation sequences, be it terminating or non-terminating.

Neither angelic nor demonic non-determinism is problematic wrt. refinements, since in these cases the respective refinement relation coincides with (the converse of) the respective approximation relation. However, in the case of erratic non-determinism, the refinement and approximation relations are quite different so that, for instance, the correctness of unfold/fold needs a separate proof. The following two very simple examples should informally explain the problem.

In the deterministic case the situation is as follows. Given a specification of a routine F with body E, denoted by $F(x) \Leftarrow E$, an unfold/fold development is a sequence of equivalence transformations which typically establishes that F satisfies a fixed point equation $F(x) \equiv A$, where F and x may occur in A. Then it is known (e.g., from Park's theorem, cf. [Park 69]) that under a certain termination condition the original specification is equivalent to the recursive routine given by A. Let's consider a concrete example. Assume a sort "nat" for natural numbers and the specification $F(x) \Leftarrow x * x$ of functionality nat→nat. Then

$$
\begin{aligned}
F(x) &\equiv x * x \\
&\equiv \textbf{if } x=0 \textbf{ then } 0 \\
&\qquad \textbf{else } (x-1)*(x-1)+x+x-1 \textbf{ fi} \\
&\equiv \textbf{if } x=0 \textbf{ then } 0 \\
&\qquad \textbf{else } F(x-1)+x+x-1 \textbf{ fi}
\end{aligned}
$$

is a sequence of equivalence transformations which shows that $F(x) \Leftarrow x * x$ is equivalent to the following recursive routine:

$$
\begin{aligned}
F(x) \Leftarrow\ & \textbf{if } x=0 \textbf{ then } 0 \\
& \textbf{else } F(x-1)+x+x-1 \textbf{ fi}
\end{aligned}
$$

(The transformations show that the function defined by $F(x) \Leftarrow x * x$ is a fixed point of the functional of the recursion; termination implies that this functional has exactly one fixed point.)

Now in the non-deterministic case we would like to have a similar transformation technique. Again, given a specification $F(x) \Leftarrow E$, a transformational development now establishes a refinement relation (denoted by $\gg_D$), i.e., $F(x) \gg_D A$, where again F and x typically occur in A. From this we would like to conclude that the original specification is refined by the recursive routine given by the declaration $F(x) \Leftarrow A$. Let's again consider a concrete example. Assume that the specification $F(x) \Leftarrow$ **some** $y : y \leq x$ of functionality nat→nat is given (the meaning of the **some**-construct is obvious) and let

$$
\begin{aligned}
F(x) \equiv\ & \textbf{some } y : y \leq x \\
\gg_D\ & \textbf{if } x < 2 \textbf{ then } x \\
& \qquad \textbf{else } x \,[]\, \textbf{some } y : y \leq x-2 \textbf{ fi} \\
\equiv\ & \textbf{if } x < 2 \textbf{ then } x \\
& \qquad \textbf{else } x \,[]\, F(x-2) \textbf{ fi}
\end{aligned}
$$

be a sequence of refinement steps. (The symbol [] denotes the finite erratic choice operator, see [Bauer et al. 85]). Then we would like to conclude that $F(x) \Leftarrow$ **some** $y : y \leq x$ can be refined into

$$
\begin{aligned}
F(x) \Leftarrow\ & \textbf{if } x < 2 \textbf{ then } x \\
& \qquad \textbf{else } x \,[]\, F(x-2) \textbf{ fi}
\end{aligned}
$$

under certain conditions.

Therefore, we prove a theorem that can be seen as an analogue of Park's well-known theorem in terms of the erratic refinement relation (viz. McCarthy's descendant order). Based on this theorem we formulate and prove correct a transformation rule enabling the unfold/fold refinement in the erratic case. The proof is given in such a way that it is largely independent of particular language features such as the parameter-passing mechanism (call-time-choice vs. run-time-choice and call-by-value vs. call-by-name) or the presence or absence of unbounded non-determinacy. Rather we give a number of simple requirements on the language in the form of algebraic laws or order relations which are sufficient to ensure that the theorem holds.

2. Totally and partially correct refinements

We model non-deterministic routines by functions that assign to a given input value a *set* of possible output values. The possibility of an aborting or non-terminating computation is indicated by including the pseudo-value $\perp$ in this set. Let therefore N be an arbitrary set of "proper" values. Then we set $N^{\perp} := N \cup \{\perp\}$ (where $\perp \notin N$). By $\mathbf{P}(N^{\perp})$ we denote the set of all non-empty subsets of $N^{\perp}$; these may be considered as sets of possible outcomes of a function application. Non-deterministic routines are then modeled by functions with range $\mathbf{P}(N^{\perp})$.

We can now define our correctness notion for program transformations. Consider functions f, g with the same argument domain and values in the same result domain $\mathbf{P}(N^{\perp})$. We extend set union and set intersection pointwise to functions by defining $(f \cup g)(x) := f(x) \cup g(x)$ and $(f \cap g)(x) := f(x) \cap g(x)$. Now, we call f a **descendant** of g (cf. [McCarthy 63]) and write $f \leq_D g$, if every possible outcome of f is also one of g, i.e., if $f \cup g = g$. Note that the **descendant relation** $\leq_D$ is a partial order.

If we consider g as the specification and f as an implementation, $f \leq_D g$ states that the implementation produces only values allowed by the specification. For this reason, we call f a **totally correct refinement** of g if $f \leq_D g$.

A more liberal relation is the following: Define for a function g with values in a domain $\mathbf{P}(N^{\perp})$ the **$\perp$-extension** $g^{\perp}$ by $g^{\perp} := g \cup \Omega$, where $\Omega(x) := \{\perp\}$ for all x from the argument domain. Then $g^{\perp}$ behaves as g except that it always has the possibility of abortion or nontermination. We call f a **partially correct refinement** of g and write $f \leq_P g$, if $f \leq_D g^{\perp}$, i.e., if whenever f chooses a terminating computation, the outcome is allowed by the specification g. If f and g are **determinate**, i.e., $|f(x)| = |g(x)| = 1$ for all x in the argument domain, then we have $f \leq_P g$ if and only if $f(x) = \{\perp\}$ or $f(x) = g(x)$ for all x.

Some useful properties of $\perp$-extension and partially correct refinement are the following:

2.1 Lemma a) The $\perp$-extension is a closure operator wrt. the descendant relation, i.e.:

$$(1)\ f \leq_D f^{\perp} \qquad (2)\ f \leq_D g \Rightarrow f^{\perp} \leq_D g^{\perp} \qquad (3)\ f^{\perp\perp} = f^{\perp}$$

b) The relation $\leq_P$ is a pre-order, i.e., it is reflexive and transitive.
c) The associated equivalence is equality of $\perp$-extensions, i.e.:

$$(f \leq_P g) \wedge (g \leq_P f) \Leftrightarrow f^{\perp} = g^{\perp}$$

Proof: a) is straightforward.
b) Reflexivity follows from (1). For transitivity assume $f \leq_P g$ and $g \leq_P h$. By (2) and (3) we obtain $g^{\perp} \leq_D h^{\perp\perp} = h^{\perp}$. Now $f \leq_P h$ follows from the transitivity of the relation $\leq_D$.
c) First, we show the direction "$\Rightarrow$". By (2) and (3), from $f \leq_P g$ and $g \leq_P f$ we get $f^{\perp} \leq_D g^{\perp}$ and $g^{\perp} \leq_D f^{\perp}$. Now antisymmetry of the descendant relation shows the desired result. The other direction follows from (1) as $f \leq_D f^{\perp} = g^{\perp}$ and $g \leq_D g^{\perp} = f^{\perp}$. ■

Note that this proof works for arbitrary closure operators.

To establish the connection between partial and total correctness, we need some further definitions. For $X \in \mathbf{P}(N^{\perp})$ we set DEF[X] if $\perp$ is not contained in X. Then, for f and g having the same argument domain and a result domain $\mathbf{P}(N^{\perp})$, we define $f \leq_{DEF} g$ if DEF[f(x)] implies DEF[g(x)] for all x from the argument domain, i.e., if $g \cap \Omega \leq_D f \cap \Omega$. Note that the **definedness relation** $\leq_{DEF}$ is also a pre-order.

2.2 Lemma Total correctness is partial correctness plus preservation of definedness. I.e.:

$$f \leq_D g \Leftrightarrow (f \leq_P g) \wedge (g \leq_{DEF} f)$$

Proof: The direction "⇒" can be shown as follows: By (1) we get $f \leq_D g \leq_D g^{\perp}$, i.e., $f \leq_P g$. Suppose now DEF[g(x)]. By $f(x) \subseteq g(x)$, from DEF[g(x)] it follows that DEF[f(x)], which shows the second assertion.
To prove "⇐", suppose $y \in f(x)$. If $y = \perp$, then $\perp \in g(x)$ follows from $g \leq_{DEF} f$ by contraposition. If $y \neq \perp$, then $y \in g^{\perp}(x)$ is equivalent to $y \in g(x)$. ∎

If one considers also functions that may yield the empty set of values, one has

$$f \leq_{DEF} g \Leftrightarrow g \cap \Omega \leq_D f \cap \Omega.$$

Then Lemma 2.2 can be restated as

$$f \leq_D g \Leftrightarrow (f \cup \Omega \leq_D g \cup \Omega) \wedge (f \cap \Omega \leq_D g \cap \Omega).$$

Our main goal now is to show that the unfold/fold-method leads to a partially correct refinement; by the previous lemma then one only needs to supply a termination argument to establish total correctness.

3. Fixed point semantics of recursion

We assume the reader to be familiar with the following notions: Complete partial order (briefly: cpo), flat domain, domain of monotonic (continuous) functions, and direct (smash) product of cpo's. Details can be found in [Loeckx Sieber 84] and in [Schmidt 86], for example. The cpo of the monotonic (continuous) functions from a cpo D to a cpo W is denoted by $(D \to W)$ (respectively by $[D \to W]$). In both cases the least element is the function Ω mapping every element from D onto the least element $\perp$ of W. By μ_f we denote the least fixed point of a monotonic function f. The direct product of cpo's D_i is denoted by ΠD_i. If the D_i are flat domains, then $\otimes D_i$ is their smash product (cf. [de Roever 72]).

To cover the case of non-determinacy, we have to define an approximation order for sets of values; the appropriate relation is the **Egli-Milner order** $\sqsubseteq_{EM}$ (cf. [Plotkin 76]) on $\mathbf{P}(D)$, where $D := N^{\perp}$, the set of proper and improper result values (cf. Section 2), is ordered as a flat domain. The Egli-Milner order is given by:

$$X \sqsubseteq_{EM} Y :\Leftrightarrow (X \subseteq Y \cup \{\perp\}) \wedge (DEF[X] \Rightarrow X = Y).$$

The pair $(\mathbf{P}(D), \sqsubseteq_{EM})$ is called the **erratic power domain** over the flat domain D. The **Plotkin power domain** is the subdomain consisting of the sets X that satisfy $\perp \in X$ if X is infinite. We set $f \leq_{EM} g$ if and only if $f(x) \sqsubseteq_{EM} g(x)$ for all elements x of the argument domain. Note also that

$$f \leq_P g \Leftrightarrow f^{\perp} \leq_{EM} g.$$

Allowing again functions with empty result set, one has

$$f \leq_{EM} g \Leftrightarrow (f \leq_P g) \wedge (f \setminus g \leq_D \Omega),$$

where $\setminus$ is the pointwise extension of set difference to functions.

Next we recall the principle of computational induction that allows proofs of properties of least fixed points. Assume two recursive routines F and G of the same functionality. Furthermore, let τ and σ be the functionals induced by the bodies of F and G, respectively. To prove that both routines have the same semantics, one must show $\Phi(\mu_\tau, \mu_\sigma)$, where the predicate Φ is defined by $\Phi(f,g) :\Leftrightarrow (f=g)$. The predicate Φ is continuous,*) i.e., whenever it holds for every element of a chain, it holds for the least upper bound of the chain, too. Hence, computational induction may be applied for proving $\Phi(\mu_\tau, \mu_\sigma)$ to be true.

Suppose function domains D_1 and D_2, *monotonic* functionals $\tau_i \in (D_i \to D_i)$, $1 \leq i \leq 2$, and a continuous predicate Φ on $D_1 \times D_2$. Then computational induction is described by the following inference rule:

$$\frac{\Phi(\Omega, \Omega) \qquad \forall f \in D_1, g \in D_2 : \Phi(f,g) \Rightarrow \Phi(\tau_1(f), \tau_2(g))}{\Phi(\mu_{\tau_1}, \mu_{\tau_2})}$$

Now, assume a non-deterministic language with erratic non-determinism and a transformation notion which also allows reducing the set of possible outcomes within a transformation step. To prove that a step from F to G is allowed, a variant of Φ can be used in which the identity relation $=$ is replaced by the descendant relation $\leq_D$.

To establish its correctness, we first note that, defining the **union** $f \cup g$ of two functions f and g by $(f \cup g)(x) := f(x) \cup g(x)$, we have that $f \leq_D g$ if and only if $f \cup g = g$. Moreover, it is well-known (cf. [Plotkin 76]) that set union is continuous wrt. the Egli-Milner order. Hence, we obtain immediately:

3.1 Theorem Let the predicate Φ on $(D_1 \to \mathbf{P}(W_1)) \times (D_2 \to \mathbf{P}(W_2))$ be defined by

$$\Phi(f,g) :\Leftrightarrow (\rho_1(f) \leq_D \rho_2(g)),$$

where $\rho_i \in [(D_i \to \mathbf{P}(W_i)) \to (D \to \mathbf{P}(W))]$, $1 \leq i \leq 2$. Then Φ is continuous. ∎

Assume that the functional ρ_2 of Theorem 3.1 is constant, i.e., that there exists a monotonic function $g_0 \in (D \to \mathbf{P}(W))$ such that $\rho_2(g) = g_0$ for all $g \in (D_2 \to \mathbf{P}(W_2))$. Then ρ_2 is trivially $\leq_{EM}$-continuous. Furthermore, we may interpret Φ as a predicate on $(D_1 \to \mathbf{P}(W_1))$, as it

*) Admissible, cf. [Loeckx Sieber 84].

depends only on the first argument. With this instantiation we have:

3.2 Corollary Let $g_0 \in (D \to \mathbf{P}(W))$ and $\rho \in [(D_1 \to \mathbf{P}(W_1)) \to (D \to \mathbf{P}(W))]$. Then the predicate Φ on $(D_1 \to \mathbf{P}(W_1))$ given by $\Phi(f) :\Leftrightarrow (\rho(f) \leq_D g_0)$ is continuous. ■

4. A formulation of Park's theorem in terms of the descendant relation

Assume a $\leq$-monotonic functional τ on the cpo $(D_1 \to W_1)$ and $f \in (D_1 \to W_1)$. As already mentioned, in the case of deterministic programs the correctness of the unfold/fold-method is based on the simple implication

(4) $\tau(f) = f \Rightarrow \mu_\tau \leq f$

together with a termination condition. The corresponding statement (σ is a $\leq_{EM}$-monotonic functional on $(D_2 \to \mathbf{P}(W_2))$ and g is from $(D_2 \to \mathbf{P}(W_2))$)

(5) $\sigma(g) \leq_D g \Rightarrow \mu_\sigma \leq_D g$

does not hold this way but in a slightly weaker form. This (Theorem 4.4 below) can be seen as an equivalent of Park's theorem (which is (4) with the order relation $\leq$ instead of the identity relation $=$ within the premise) in terms of the descendant order and the $\perp$-extension. We prepare the proof by some technical lemmas. The proofs of the following facts are rather trivial and, therefore, omitted.

4.1 Lemma Assume two functions $f, g \in (D \to \mathbf{P}(W))$. Then:

(6) $f \leq_{EM} g \Rightarrow f \leq_P g$ (7) $f^\perp \leq_{EM} f$ ■

Using these properties, we now obtain:

4.2 Lemma Let $f, g \in (D \to \mathbf{P}(W))$. Then $f \leq_{EM} g$ and $g \leq_D f$ imply $f^\perp = g^\perp$.

Proof: From (6) and the first assumption we get $f \leq_P g$. From the second assumption, formula (1), and the transitivity of the relation $\leq_D$ we obtain $g \leq_P f$. Now Lemma 2.1 c) shows the desired result. ■

The next preparatory result deals with the replacement of functions by their $\perp$-extensions within function applications.

4.3 Lemma Let τ be a $\leq_{EM}$-monotonic and $\leq_D$-monotonic functional on $(D \to \mathbf{P}(W))$ and $f \in (D \to \mathbf{P}(W))$. Then $\tau(f^\perp)^\perp = \tau(f)^\perp$.

Proof: First, we get $f^\perp \leq_{EM} f$ from (7); $\leq_{EM}$-monotonicity of τ shows $\tau(f^\perp) \leq_{EM} \tau(f)$. On the other hand, due to formula (1) we have $f \leq_D f^\perp$ and the $\leq_D$-monotonicity, thus, implies $\tau(f) \leq_D \tau(f^\perp)$. Finally, Lemma 4.2 yields the desired equation. ■

After these preparations we are now able to prove the main theorem of this section, viz. the announced equivalent of Park's theorem in terms of the descendant order and the $\perp$-extension. The proof of the following theorem is an adaptation of the proof given in [Berghammer 90]. (That proof establishes the correctness of the unfold/fold-method for the erratic non-deterministic programming language CIP-L*).)

4.4 Theorem Let τ be a $\leq_{EM}$-monotonic and $\leq_D$-monotonic functional on $(D \to \mathbf{P}(W))$ and $f \in (D \to \mathbf{P}(W))$. If $\tau(f)$ is a totally correct refinement of f, then the least fixed point of τ is a partially correct refinement of f. I.e.:

$$\tau(f) \leq_D f \Rightarrow \mu_\tau \leq_P f$$

Proof: We choose the continuous predicate Φ on the function domain $(D \to \mathbf{P}(W))$ in Corollary 3.2 to be $\Phi(g) :\Leftrightarrow (g \leq_D f^\perp)$.

Induction base: Trivial, as $\Omega(x) = \{\perp\} \subseteq f^\perp(x)$ for all $x \in D$.

Induction step: From the equation $\tau(f^\perp)^\perp = \tau(f)^\perp$ (Lemma 4.3) and the assumption $\tau(f) \leq_D f$ we obtain in connection with (2) of Lemma 2.1 as a first result $\tau(f^\perp)^\perp = \tau(f)^\perp \leq_D f^\perp$. Now we use the induction hypothesis $\Phi(g)$, viz. the property $g \leq_D f^\perp$. By monotonicity of τ wrt. the descendant relation $\leq_D$ we get $\tau(g) \leq_D \tau(f^\perp)$. Thus, (1) of Lemma 2.1 implies $\tau(g) \leq_D \tau(f^\perp)^\perp$ as a second result.
Finally, the combination of both results just proven yields the relation $\tau(g) \leq_D f^\perp$, which is exactly the desired result $\Phi(\tau(g))$. ■

Theorem 4.4 in combination with Lemma 2.2 now immediately gives the following result.

4.5 Corollary Let τ and f be as in Theorem 4.4. If $\tau(f)$ is a totally correct refinement of f and the least fixed point of τ is at least as defined as f, then the least fixed point of τ is a totally correct refinement of f. I.e.:

$$(\tau(f) \leq_D f) \wedge (f \leq_{DEF} \mu_\tau) \Rightarrow \mu_\tau \leq_D f. \qquad \blacksquare$$

Let the function f of Theorem 4.4 be the semantics of a specification and the functional τ of the same theorem be obtained by the interpretation of its recursive refinement. Furthermore, assume that during the derivation process the set of possible outcomes may have been reduced. Then the conclusion of the theorem expresses that every defined result of the recursion is a result of the specification, too, but for some inputs it may happen that the recursion diverges although the specification yields only defined results. Hence, like in the case of (deterministic) unfold/fold for a correctness of (non-deterministic) refinement also a supplementary termination proof is needed. This is formally expressed by the second part of the premise of the corollary: Whenever the specification yields defined values only, its recursive refinement has to terminate.

*) For a formal description of the language CIP-L see the language report [Bauer et al. 85].

5. Non-deterministic refinement of routines: Formulation as a rule

Translating Theorem 4.4 (Corollary 4.5, respectively) into programming language notation, monotonicity of the functional τ wrt. the descendant order corresponds to the so-called substitution-property, saying: If a sub-expression A_1 of an expression E_1 can be refined into an expression A_2 and the expression E_2 is obtained from E_1 by replacing A_1 by A_2, then a refinement of E_1 into E_2 is also correct. The substitution-property plays an essential role as it shows that "local" application of correct refinements also is "globally" correct.

In the sequel, we define the syntax of a simple non-deterministic applicative programming language PL. Then we show that the functionals induced by the bodies of the routines of PL fulfill the properties of Theorem 4.4 (Corollary 4.5, respectively) if certain simple postulates about their semantics hold.

For simplicity, a program in PL consists of an expression or of exactly one declaration of an applicative routine, followed by a semicolon and an expression. Expressions are built up by the use of variables, constants, base-function symbols, routine identifiers, function application, conditional, and finite or comprehensive choice. Our approach can easily be extended to the case of several declarations and/or of systems of mutually recursive routines using an environment and multi-argument functionals in the description of the semantics.

The grammar of the programming language PL is specified by:

prog ::= expr |
routine-id (var ,..., var) ⇐ expr; expr
expr ::= var |
const |
base-fun (expr ,..., expr) |
routine-id (expr ,..., expr) |
if expr **then** expr **else** expr **fi** |
expr [] expr |
some var : expr

Note that the inclusion of the **some**-construct may lead to unbounded non-determinacy.

It will be assumed that each variable and each constant is assigned a sort and each base-function symbol and routine identifier is assigned a functionality. Thus, we can define the sort of an expression in the usual way. If $F(v) \Leftarrow E; A$ is a program and $m_1...m_j \rightarrow n$ is the functionality of F, it is understood that $v = v_1,...,v_j$ is a list of pairwise distinct variables v_i, $1 \leq i \leq j$, each m_i is the sort of v_i, and n is the sort of E. Furthermore, it is supposed that the body E of the routine F can only contain the v_i as variables and F as routine identifier and that the "result expression" A of the program contains at most F as routine identifier and arbitrary variables. In programs without a routine declaration no routine identifiers may occur. Moreover, the expressions following "**if**" as well as "**some** var :" are assumed to be of sort "boolean"; the then and the else branches of the conditional as well as the alternatives of the

choice operator $[]$ are assumed to be of equal sorts.

Rather than giving a specific semantics to PL, we only state some postulates about the semantics which are actually needed in the sequel. These postulates are very natural and do not restrict the possible semantics of PL much.

Let $v = v_1,...,v_k$ denote a list of pairwise distinct variables. Then $EXP^n_{F,v}$ denotes the expressions of sort n which contain at most $v_1,...,v_k$ as variables and F as routine identifier. Furthermore, we assume for every base-sort m a flat domain D_m and for every tuple-sort $(m_1,...,m_k)$ a cpo $D_{m_1...m_k}$ which is built up from the cpo's D_{m_i} in a certain (unspecified) manner. The following table shows possible choices for $D_{m_1...m_k}$ appropriate for various parameter passing mechanisms:

$D_{m_1...m_k}$	call-by-value	call-by-name
call-time-choice	$\otimes D_{m_i}$	ΠD_{m_i}
run-time-choice	$\mathbf{P}(\otimes D_{m_i})$	$\Pi \mathbf{P}(D_{m_i})$

See e.g., [Hennessy Ashcroft 76], [Hennessy 80], and [Bauer et al. 85]. Moreover, we abbreviate function domains like $(D_m \to \mathbf{P}(D_n))$ by $D_{m\to n}$.

As an example for the difference between the four parameter passing mechanism consider the routine declaration $F(x,y) \Leftarrow x+x$. Then the call $F(0[]1,0)$ is equivalent to $0[]2$ with call-time-choice and to $0[]1[]2$ with run-time-choice The call F(0,error) is equivalent to error with call-by-value and to 0 with call-by-name, where error is an expression with $\perp$ as only possible value.

Postulate 1: Let m be a (tuple) sort, F be a routine identifier of functionality $m\to n$, and v be a (list of) variable(s) of (tuple) sort p. We postulate the semantics of the language to be given by a family of mappings

$$[\![.]\!] : EXP^q_{F,v} \to (D_{m\to n} \to D_{p\to q}).$$

This means that $[\![.]\!]$ is supposed to yield $\leq_{EM}$-*monotonic* functionals on $\sqsubseteq_{EM}$-*monotonic* functions for all expressions.

Postulate 2: Next we explain the purpose of the various arguments of the semantic function $[\![.]\!](.)(.)$ in terms of the equation

$$[\![F(v)]\!](f)(x) = f(x),$$

where v is a list of variables $v_1,...,v_j$ and x is a tuple $(x_1,...,x_j)$.

Note that this postulate is only due to the specific form chosen for the semantic function and not of any deep importance. It simply says that a free routine identifier and free variables in an expression are valuated by the second and the third arguments of the semantic function

$[\![.]\!](.)(.)$. For a concrete semantics, Postulate 2 should be an immediate consequence of the semantics of routine application and of variables. E.g., in the case of a call-by-name/run-time-choice semantics it is shown by the following chain of equations:

$$[\![F(v)]\!](f)(x) = f([\![v_1]\!](f)(x),...,[\![v_j]\!](f)(x)) = f(x_1,...,x_j) = f(x)$$

Postulate 3: Let the routine identifier F be of functionality $m \to n$ and let v be a list of variables of sort m. Consider a (possibly recursive) declaration $F(v) \Leftarrow E$ with $E \in EXP^n_{F,v}$. We have, by Postulate 1, $[\![E]\!] \in (D_{m\to n} \to D_{m\to n})$. Thus, according to [Markowsky 76]) the least fixed point $\mu_{[\![E]\!]}$ wrt. the order $\leq_{EM}$ exists. We postulate for the semantics of that declaration

$$[\![F(v) \Leftarrow E]\!] = \mu_{[\![E]\!]},$$

i.e., a least fixed point semantics of recursion.

Postulate 4: Next, we require the semantics of a program $F(v) \Leftarrow E;\ A$ to be compositional, i.e., we postulate the equation

$$[\![F(v) \Leftarrow E;\ A]\!] = [\![A]\!]([\![F(v) \Leftarrow E]\!]).$$

Postulate 5: In the case of program transformation a very fundamental and natural requirement on the semantics is, as already explained above, the substitution property. Hence, we postulate for every expression E the functional $[\![E]\!]$ to be monotonic wrt. the descendant order $\leq_D$.

Next, we introduce the notion of a transformation rule over the programming language PL. Correctness of a rule is defined such that the transition from programs (program parts) to less ambiguous ones is allowed. In order to formalize transformation rules, we need some attributes and relations on expressions, routines, and programs formalizing properties like "the expression E yields only defined values" or "the expression (routine, program) P_1 is less ambiguous than the expression (routine, program) P_2". See also [Broy Partsch Pepper Wirsing 80] and [Bauer et al. 85]. The following definition transfers the descendant order onto the syntactic level.

5.1 Definition a) For two expressions $E_1, E_2 \in EXP^q_{F,v}$, where $m \to n$ is the functionality of the routine identifier F, we define:

$$E_1 \ll_D E_2 :\Leftrightarrow \forall f \in D_{m\to n} : [\![E_1]\!](f) \leq_D [\![E_2]\!](f)$$

b) For two routine declarations $F(v) \Leftarrow E_1$ and $G(w) \Leftarrow E_2$, where the routine identifiers F and G have the same functionality, we define:

$$F(v) \Leftarrow E_1 \ll_D G(w) \Leftarrow E_2 :\Leftrightarrow [\![F(v) \Leftarrow E_1]\!] \leq_D [\![G(w) \Leftarrow E_2]\!]$$

c) Assume two programs $F(v) \Leftarrow E_1;\ A_1$ and $G(w) \Leftarrow E_2;\ A_2$, where $A_1 \in EXP^n_{F,z}$ and $A_2 \in EXP^n_{G,z}$ for some list z of variables. Then we define:

$$F(v) \Leftarrow E_1;\ A_1 \ll_D G(w) \Leftarrow E_2;\ A_2 :\Leftrightarrow [\![F(v) \Leftarrow E_1;\ A_1]\!] \leq_D [\![G(w) \Leftarrow E_2;\ A_2]\!] \quad \blacksquare$$

Let P_1 and P_2 be two expressions (routines, programs). If $P_1 \ll_D P_2$ and $P_2 \ll_D P_1$, then P_1 and P_2 are said to be **equivalent**. We denote this by $P_1 \equiv P_2$. Now, we may define:

5.2 Definition A **transformation rule** over the language PL is an inference rule of the form

$$\frac{>\text{condition}_1< \ \ldots\ >\text{condition}_n<}{>\text{input}< \ \gg\ >\text{output}<}$$

where $n \geq 0$, $\gg \in \{\equiv, \gg_D\}$, $\gg_D$ is the converse of $\ll_D$, the **output template** >output< and the **input template** >input< are both expressions (routines, programs) over the language PL, and the **applicability conditions** $>\text{condition}_i<$, $1 \leq i \leq n$, are formulas over PL built up by attributes and relations over the language PL such as $\gg_D$ and $\equiv$. The rule is said to be **correct**, if >output< and >input< have the same sort (functionality) and if the validity of the conditions $>\text{condition}_i<$, $1 \leq i \leq n$, implies the validity of the relation >input< $\gg$ >output<. $\blacksquare$

In order to improve the readability of the rules presented in the rest of the paper, we omit conditions such as "the expression E is of sort m" and state only the "semantic" applicability conditions explicitly. (In [Partsch 85], conditions of the first kind are called syntactic constraints.)

5.3 Example We transfer the definedness predicate DEF onto the syntactic level and define an expression B of PL to be defined (denoted as DEFINED[B]) if it contains no routine identifier and if $DEF[[\![B]\!](\Omega)(x)]$ holds for every element $x \neq \bot$ (which is the same as $DEF[[\![B]\!](f)(x)]$ for every f and every $x \neq \bot$). The following transformation rule COND is correct for any reasonable semantics of the conditional. It may be used for introducing or removing a conditional.

$$\frac{\text{DEFINED[B]}}{E \equiv \textbf{if } B \textbf{ then } E \textbf{ else } E \textbf{ fi}} \quad \blacksquare$$

If we want to formulate Corollary 4.5 as a transformation rule we also need to transfer the definedness relation $\leq_{DEF}$ onto the syntactic level.

5.4 Definition For two routines $F(v) \Leftarrow E_1$ and $G(w) \Leftarrow E_2$, where F and G have the same functionality, we define:

$$F(v) \Leftarrow E_1 \ll_{DEF} G(w) \Leftarrow E_2 :\Leftrightarrow [\![F(v) \Leftarrow E_1]\!] \leq_{DEF} [\![G(w) \Leftarrow E_2]\!] \quad \blacksquare$$

After these preliminaries we are now able to formalize the refinement of non-deterministic routines as a transformation rule over PL. Note that in the following theorem the result expression A of the program $F(v) \Leftarrow E;\ A$ of the first applicability condition can only contain the variables in v due to the form $F(v) \Leftarrow A$ of the output template.

5.5 Theorem If the programming language PL satisfies Postulate 1 through Postulate 5, then the following inference rule, called REFINE, is a correct transformation rule.

$$\frac{F(v) \Leftarrow E; F(v) \;\gg_D\; F(v) \Leftarrow E; A \qquad F(v) \Leftarrow E \;\ll_{DEF}\; F(v) \Leftarrow A}{F(v) \Leftarrow E \;\gg_D\; F(v) \Leftarrow A}$$

Proof: First, we obtain $[\![A]\!](\mu_{[\![E]\!]}) \leq_D \mu_{[\![E]\!]}$ from the first applicability condition as follows:

$$\begin{array}{lll} & F(v) \Leftarrow E; F(v) \;\gg_D\; F(v) \Leftarrow E; A & \\ \Leftrightarrow & [\![F(v) \Leftarrow E; A]\!] \leq_D [\![F(v) \Leftarrow E; F(v)]\!] & \text{by definition of } \gg_D \\ \Leftrightarrow & [\![A]\!]([\![F(v) \Leftarrow E]\!]) \leq_D [\![F(v)]\!]([\![F(v) \Leftarrow E]\!]) & \text{by Postulate 4} \\ \Leftrightarrow & [\![A]\!](\mu_{[\![E]\!]}) \leq_D [\![F(v)]\!](\mu_{[\![E]\!]}) & \text{by Postulate 3} \\ \Leftrightarrow & [\![A]\!](\mu_{[\![E]\!]}) \leq_D \mu_{[\![E]\!]} & \text{by Postulate 2} \end{array}$$

As a second result we get $\mu_{[\![E]\!]} \leq_{DEF} \mu_{[\![A]\!]}$ from the second applicability condition, since:

$$\begin{array}{lll} & F(v) \Leftarrow E \;\ll_{DEF}\; F(v) \Leftarrow A & \\ \Leftrightarrow & [\![F(v) \Leftarrow E]\!] \leq_{DEF} [\![F(v) \Leftarrow A]\!] & \text{by definition of } \ll_{DEF} \\ \Leftrightarrow & \mu_{[\![E]\!]} \leq_{DEF} \mu_{[\![A]\!]} & \text{by Postulate 3} \end{array}$$

Finally, the functional $[\![A]\!]$ is $\leq_{EM}$-monotonic and $\leq_D$-monotonic due to Postulate 1 and Postulate 5, and the least fixed point $\mu_{[\![E]\!]}$ is a $\sqsubseteq_{EM}$-monotonic function due to Postulate 1. Hence, the assumptions of Corollary 4.5 hold. It shows $\mu_{[\![A]\!]} \leq_D \mu_{[\![E]\!]}$. Now, we get the desired result:

$$\begin{array}{lll} & F(v) \Leftarrow E \;\gg_D\; F(v) \Leftarrow A & \\ \Leftrightarrow & [\![F(v) \Leftarrow A]\!] \leq_D [\![F(v) \Leftarrow E]\!] & \text{by definition of } \gg_D \\ \Leftrightarrow & \mu_{[\![A]\!]} \leq_D \mu_{[\![E]\!]} & \text{by Postulate 3} \end{array}$$ ■

5.6 Remarks a) The rule REFINE expresses, like in the deterministic case, that refinement consists of two steps, viz. the derivation (or verification) of a recursion relation and a subsequent proof that termination is preserved.

b) The first applicability condition of REFINE may be shown by a transformation of the program $F(v) \Leftarrow E; F(v)$ into the program $F(v) \Leftarrow E; A$. Here frequently the unfold/fold-method is used. Its first step is an unfolding-step and replaces the routine call $F(v)$ by the body E of F. A corresponding rule UNFOLD reads:

$$\frac{}{F(v) \Leftarrow E; F(v) \;\equiv\; F(v) \Leftarrow E; E}$$

It is correct for the programming language PL provided Postulate 1 through Postulate 5 hold. The proof essentially consists of the following equations:

$$[\![F(v) \Leftarrow E; E]\!] = [\![E]\!](\mu_{[\![E]\!]}) = \mu_{[\![E]\!]} = [\![F(v)]\!](\mu_{[\![E]\!]}) = [\![F(v) \Leftarrow E; F(v)]\!]$$

Note that the actual argument expression in the call of F must consist of variables only.

Recursion is introduced by folding-steps. Every folding-step interprets a subexpression of an expression as an instance of the body of F and replaces it by a call of F. A formalization of folding as a rule FOLD depends on the specific language semantics. E.g., in the case of a strict call-by-value semantics we have the following rule which, in general, reduces non-determinacy (.[.←.] denotes the substitution operator and A is an expression which contains exactly w as variable):

$$\frac{\text{DEFINED[K]}}{F(v) \Leftarrow E;\ A[E[K \leftarrow v] \leftarrow w] \ \gg_D\ F(v) \Leftarrow E;\ A[F(K) \leftarrow w]}$$

This means: If A contains a subexpression which equals the body E of F in which the expression K is substituted for the variable, then this subexpression can be replaced by a call of F with argument K. The condition DEFINED[K] is sufficient in strict call-by-value semantics: Non-definedness of K would imply that F(K) is also non-defined, whereas it may happen that the expression E[K←v] still is defined viz., if E is not strict wrt. v. This shows, that in the transformation rule FOLD the condition DEFINED[K] may be replaced by the condition of E being strict wrt. v. ■

6. Application: A rule for deriving tail-recursions

As an application of the general rule REFINE we now show the correctness of an analogue of the Hoare-Dijkstra-Gries way of constructing loop programs from specifications. To this end, we start from the specification

some x : R(x,v),

where R is a Boolean base function and v is a sequence $v_1,...,v_j$ of input parameters. We want to derive a program G(y,v) ⇐ ...; G(E,v), where G is a tail-recursive routine implementing this specification. To obtain this, we introduce (cf. [Gries 81]) besides the starting expression E an invariant P and a termination condition B (both given as Boolean base functions) such that

(8) DEFINED[E,B(y,v),P(y,v)] (9) DET[B(y,v),P(y,v)]

(where the predicate DET states that an expression is determinate, i.e., yields exactly one value) and

(10) $P(E,v) \equiv \text{true}$ (11) $P(y,v) \wedge B(y,v) \equiv P(y,v) \wedge B(y,v) \wedge R(y,v)$

as well as an incrementation function F satisfying

(12) $P(y,v) \wedge \neg B(y,v) \equiv P(y,v) \wedge \neg B(y,v) \wedge P(F(y,v),v)$.

Note that (11) and (12) are equational formulations of the requirement that $P(y,v) \wedge B(y,v)$ implies $R(y,v)$ and $P(y,v) \wedge \neg B(y,v)$ implies $P(F(y,v),v)$. For the further development we assume a constant error with $[\![error]\!] = \Omega$ and abbreviate, for Boolean expressions Q and arbitrary expressions A, the conditional **if** Q **then** A **else** error **fi** by $Q \rhd A$ (cf. [Möller 89]). Q is called an assertion for A. The following three transformation rules $ASSERT_i$, $1 \leq i \leq 3$, hold under any reasonable interpretation of the conditional.

a) $ASSERT_1$ states that an assertion which is equivalent to true may be eliminated:

$$\frac{Q \equiv \text{true}}{Q \rhd A \equiv A}$$

b) If Q is a defined and determinate assertion for a conditional, the defined and determinate condition of which is C, then the assertions A_1 and A_2 may be inserted into the **then**-case and the **else**-case, respectively, provided $Q \wedge C$ implies A_1 and $Q \wedge \neg C$ implies A_2. Formalized as a rule $ASSERT_2$ this fact reads:

$$\frac{\begin{array}{c}\text{DEFINED}[C,Q] \\ \text{DET}[C,Q] \\ Q \wedge C \equiv Q \wedge C \wedge A_1 \\ Q \wedge \neg C \equiv Q \wedge \neg C \wedge A_2\end{array}}{Q \rhd \textbf{if } C \textbf{ then } E_1 \textbf{ else } E_2 \textbf{ fi} \equiv Q \rhd \textbf{if } C \textbf{ then } A_1 \rhd E_1 \textbf{ else } A_2 \rhd E_2 \textbf{ fi}}$$

c) Finally, the rule $ASSERT_3$ expresses a property of comprehensive choice, viz. that under the assertion $R(y,v)$ the choice **some** $x : R(x,v)$ can be refined to y:

$$\frac{}{R(y,v) \rhd \textbf{some } x : R(x,v) \;\gg_D\; R(y,v) \rhd y}$$

In the sequel we use a slight generalization of the rule UNFOLD to the case of an argument expression not consisting of variables only.

Let D_F abbreviate the declaration $F(v) \Leftarrow E$. We want to replace a call F(A) by the body E of F in which the actual parameter A is substituted for the formal parameter v. The correctness of this depends on the particular parameter passing mechanism. A corresponding rule UNFSUB (unfolding and substitution) reads

$$\frac{CU[A]}{D_F;\, F(A) \equiv D_F;\, E[A \leftarrow v]}$$

where sufficient conditions CU[A] are the following:

CU[A]	call-by-value	call-by-name
call-time-choice	DEF_DET[A]	DET[A]
run-time-choice	DEFINED[A]	true

Here DET[A] means that A is determinate; DEF_DET[A] means that A is defined and determinate. Note that the conditions again could be weakened using strictness properties of E. Furthermore, in the case of call-by-value and call-time-choice, the condition DEFINED[A] suffices for D_F; E[A←v] $»_D$ D_F; F(A) to hold.

We also use the obvious fact that a routine declaration can be inserted in front of an expression containing no routine identifier. Then condition (10) and UNFSUB show the following implication

(13) CU[E] $\Rightarrow$ **some** x : R(x,v) $\equiv$ D_{spec}; G(E,v),

where D_{spec} abbreviates the declaration G(y,v) $\Leftarrow$ P(y,v) $\triangleright$ **some** x : R(x,v). We calculate (the implicitly used substitution property is a consequence of Postulate 5):

D_{spec};
G(y,v)
$\equiv$ D_{spec};
P(y,v) $\triangleright$ **some** x : R(x,v) — by UNFOLD
$\equiv$ D_{spec};
P(y,v) $\triangleright$ **if** B(y,v) **then some** x : R(x,v)
else some x : R(x,v) **fi** — by (8), COND
$\equiv$ D_{spec};
P(y,v) $\triangleright$ **if** B(y,v) **then** R(y,v) $\triangleright$ **some** x : R(x,v)
else P(F(y,v),v) $\triangleright$ **some** x : R(x,v) **fi** — by (11), (12), $ASSERT_2$
$»_D$ D_{spec};
P(y,v) $\triangleright$ **if** B(y,v) **then** R(y,v) $\triangleright$ y
else P(F(y,v),v) $\triangleright$ **some** x : R(x,v) **fi** — by $ASSERT_3$
$»_D$ D_{spec};
P(y,v) $\triangleright$ **if** B(y,v) **then** R(y,v) $\triangleright$ y
else G(F(y,v),v) **fi** — by FOLD
$\equiv$ D_{spec};
P(y,v) $\triangleright$ **if** B(y,v) **then** R(y,v) $\triangleright$ y
else true $\triangleright$ G(F(y,v),v) **fi** — by $ASSERT_1$
$\equiv$ D_{spec};
P(y,v) $\triangleright$ **if** B(y,v) **then** y
else G(F(y,v),v) **fi** — by (11), $ASSERT_2$

Hence, we have derived a tail-recursion for G, verifying the first applicability condition of the transformation rule REFINE. Let therefore D_{rec} abbreviate the declaration

$$G(y,v) \Leftarrow P(y,v) \triangleright \textbf{if } B(y,v) \textbf{ then } y \textbf{ else } G(F(y,v),v) \textbf{ fi},$$

i.e., the tail-recursive solution of the original specification. Then we have the implication

$$\begin{aligned} D_{spec} \ll_{DEF} D_{rec} &\Rightarrow D_{spec} \gg_D D_{rec} && \text{by REFINE} \\ &\Rightarrow D_{spec}; G(E,v) \gg_D D_{rec}; G(E,v) && \text{by Postulate 4} \\ &\Leftrightarrow \textbf{some } x : R(x,v) \gg_D D_{rec}; G(E,v) && \text{by (13)} \end{aligned}$$

provided CU[E] holds so that we finally obtain the following transformation rule TAILREC:

$$\frac{\begin{array}{c}\text{CU[E]} \\ \text{(8) (9) (10) (11) (12)} \\ D_{spec} \ll_{DEF} D_{rec}\end{array}}{\textbf{some } x : R(x,v) \gg_D D_{rec}; G(E,v)}$$

It should be mentioned, that this rule is the basis for compact rules describing some general algorithm design techniques like augmentation or the greedy approach. For details we refer to [Berghammer 90], where such transformation rules are proved using the language CIP-L (cf. [Bauer et al. 85]) for program notation.

7. Conclusion

In this paper we have demonstrated that the common method of program development by unfold/fold in the deterministic case can be applied in the erratic non-deterministic case as well. We have formulated a very general rule describing the refinement of non-deterministic routines. As a specific case this rule formalizes the unfold/fold method, too. Rather than giving a proof using a specific language semantics, we have only stated some postulates about the semantics which have been actually needed. These postulates turned out to be very natural and do not restrict the possible semantics much. Finally, we demonstrated an application, viz. the proof of a rule describing the derivation of tail-recursions from invariants. The results of this paper show that there is no need to restrict functional programming to the deterministic case.

Acknowledgement

We thank Manfred Broy and Wolfgang Heinle for giving us valuable hints on a draft version of this paper.

References

[Back 88]
Back R. J. R.: A calculus of refinements for program derivations. Acta Informatica 25, 593-624 (1988)

[Bauer et al. 85]
Bauer F. L., Berghammer R., Broy M., Dosch W., Geiselbrechtinger F., Gnatz R., Hangel E., Hesse W., Krieg-Brückner B., Laut A., Matzner T., Möller B., Nickl F., Partsch H., Pepper P., Samelson K., Wirsing M., Wössner H.: The Munich Project CIP, Volume I: The wide spectrum language CIP-L. LNCS 183, Springer: Berlin-Heidelberg-New York (1985)

[Bauer et al. 89]
Bauer F. L., Möller B., Partsch H., Pepper P.: Formal program construction by transformations – Computer-aided, Intuition-guided Programming. IEEE Trans. on Software Engineering, SE-15, 2, 165-180 (1989)

[Berghammer 90]
Berghammer R.: Transformational programming with non-deterministic and higher-order constructs. In preparation

[Broy 80]
Broy M.: Transformational semantics for concurrent programs. Inform. Processing Letters 11, 87-91 (1980)

[Broy Partsch Pepper Wirsing 80]
Broy M., Partsch H., Pepper P., Wirsing M.: Semantic relations in programming languages. In: Lavington S. H. (ed): Information Processing 80, Proc. IFIP Congress 80 - 8th World Computer Congress, North-Holland: Amsterdam, 101-106 (1980)

[Burstall Darlington 77]
Burstall R. M., Darlington J.: A transformation system for developing recursive programs. Journal ACM 24, 44-67 (1977)

[de Roever 72]
de Roever W. P.: A formalization of various parameter mechanisms as products of relations within a calculus of recursive program schemes. Theorie des Algorithmes, des Languages et de la Programmation, Seminaires IRIA, 55-88 (1972)

[Feather 87]
Feather M. S.: A survey and classification of some program transformation techniques. In: Meertens L. G. L. T (ed.): Proc. TC 2 Working Conference on Program Specification and Transformation, April 15-17, 1986, Bad Tölz, F.R.G., North-Holland: Amsterdam, 165-195 (1987)

[Furbach 83]
Furbach U.: Über Transformationsregeln für nichtdeterministische rekursive Funktionsdefinitionen. Dissertation, Fachbereich Informatik, Hochschule der Bundeswehr München (1983)

[Gries 81]
Gries D.: The science of computer programming. Springer: Berlin-Heidelberg-New York (1981)

[Hennessy 80]
Hennessy M.: The semantics of call-by-value and call-by-name in a nondeterministic environment. SIAM J. Comp. 1, 67-84 (1980)

[Hennessy Ashcroft 76]
Hennessy M., Ashcroft E. A.: The semantics of non-determinism. In: Michaelson S., Milner R. (eds.): Proc. 3rd ICALP, Edinburgh, 478-493 (1976)

[Loeckx Sieber 84]
Loeckx J., Sieber K.: The foundations of program verification. Teubner: Stuttgart (1984)

[Markowsky 76]
Markowsky G.: Chain-complete posets and directed sets with applications. Algebra universalis 6, 53-68 (1976)

[McCarthy 63]
McCharty J.: A basis for a mathematical theory of computation. In: Braffort P., Hirschberg D. (eds): Computer programming and formal systems. North-Holland : Amsterdam, 33-69 (1963)

[Morgan et al. 88]
Morgan C. C., Robinson K. A., Gardiner P. H. B.: On the refinement calculus. Technical report PRG-70, Programming Research Group, Oxford University (1988)

[Möller 89]
Möller B.: Applicative assertions. In: van de Snepscheut J. L. A. (ed.): Mathematics of program construction. LNCS 375, Springer: Berlin-Heidelberg-New York, 348-362 (1989)

[Morris 87]
Morris J. M.: A theoretical basis for stepwise refinement and the programming calculus. Science of Computer Programming 9, 298-306 (1987)

[Park 69]
Park D.: Fixpoint induction and proofs of program properties. In: Meltzer B., Michie D. (eds.): Machine Intelligence 5, Edinburgh Univ. Press, 59-78 (1969)

[Partsch 85]
Partsch H.: Transformational program development in a particular problem domain. Habilitationsschrift, Institut für Informatik, TU München (1985). Also in: Science of Computer programming 7, 99-241 (1986)

[Plotkin 76]
Plotkin G.: A powerdomain construction. SIAM Journal on Comp. 5, 452-487 (1976)

[Schmidt 86]
Schmidt D. A.: Denotational semantics - A methodology for language development. Allyn and Bacon: Boston (1986)