Assertions and Recursions

Bernhard Möller

Institut für Mathematik, Universität Augsburg, D-86135 Augsburg, Germany,
e-mail: moeller@uni-augsburg.de

Abstract. We provide an algebraic description of subtypes and the way they propagate through recursive functions. By abstracting from the concrete domain of functions or relations we obtain a framework which is independent of strict or non-strict, deterministic or non-deterministic semantics. Applications include efficiency increasing simplification of recursions as well as proofs about recursions by noetherian induction, such as termination proofs.

1 Introduction

The paper [12] presented a way of introducing and manipulating assertions in an applicative language, mainly for use in transformational program development. The treatment was based one particular functional language with strict call-time choice semantics, and the essential rule for strengthening and weakening assertions was proved only for the case of linear recursion. Although this was already a generalization of corresponding techniques from the refinement calculus (see e.g. [15]), one would have liked a rule that works for arbitrary recursions. Related investigations are reported in [17] for the case of a non-strict functional semantics.

In the present paper we abstract from the particular semantic framework; the central requirements on the objects involved lead to the notion of a composition algebra. Concrete instances of this are usual functions over domains, binary relations, but also formal languages and sets of paths in graphs. We come up with an algebraic characterization of invariants and a general rule for strengthening and weakening them in recursions over the respective composition algebra. The rule is a simple instantiation of the fusion rule for least fixpoints.

We extend the technique to cover proofs by noetherian induction, in particular, termination proofs, for general recursions.

2 Assertions

2.1 Representation

We want to model that at a certain occurrence of an expression E an assertion A is satisfied. To this end we replace the occurrence of E under consideration by another expression $A \triangleright E$ where $A \triangleright E$ is equivalent to E if A holds at that occurrence and to error otherwise, where error represents the semantical pseudo-value

$\perp$ or "undefined". This construct corresponds to the assert-statement in CIP-L [3, 2] and the refinement calculus (see e.g. [15]); its properties were investigated in [12] for a strict semantics.

As assertions we allow arbitrary logical formulas over expressions of the language as terms and relations such as semantic equivalence $\equiv$ or approximation $\sqsubseteq$ as predicates. The semantics of the assertion expression is then more precisely defined, using an evaluation function

$$[\![\_]\!] : \mathit{Expr} \to \mathit{Env} \to \mathit{Val} \ ,$$

by the equation

$$[\![A \rhd E]\!]\rho \overset{\text{def}}{=} \begin{cases} [\![E]\!] \text{ if } \rho \models A \ , \\ \perp \quad \text{otherwise} \ . \end{cases}$$

Note that this may lead to non-monotonic expressions. However, as long as for a recursively defined function the assertions do not contain recursive calls, the associated functional will still be monotonic.

It is convenient to admit also Boolean expressions B of the programming language as assertions; the expression $B \rhd E$ is then defined to be an abbreviation for $(B \equiv \mathit{true}) \rhd E$. This yields monotonic expressions; such an expression is equivalent to

$$\mathsf{if}\ B\ \mathsf{then}\ E\ \mathsf{else}\ \mathsf{error}\ .$$

The use of Boolean assertions admits assertions as first-class objects [17], in particular, recursively defined assertions.

While the above semantics draws on a conventional domain-theoretic approach, there is another possibility if one considers only strict operations. In this case $\perp$ is not needed and one can use simple argument-value-pairs as semantics for (partial) functions. This approach was used e.g. in [10]. In this case assertions can simply be represented as subsets of the domain of discourse, namely as the set of all values satisfying the assertions. The theory in this case becomes much smoother if one embeds the partial functions into the set of arbitrary binary relations between domain and co-domain. This approach is followed in [14, 1, 5]; it also allows a convenient treatment of non-determinacy. Expressions are then set-valued, giving the set of all possible results; the empty set $\emptyset$ plays the role of $\perp$.

The assertion expression can then be defined analogously. We want to avoid the associated technical details (see [12, 14]), since they are not needed in the sequel. Rather, we present a treatment that is independent of the particular representation of assertions and hence works for both of the above approaches. In any of them the following algebraic properties hold:

$$\mathit{TRUE} \rhd E \;=\; E \tag{1}$$

$$A \rhd (B \rhd E) \;=\; (A \wedge B) \rhd E \ , \tag{2}$$

$$A \rhd f(E) \;=\; A \rhd f(A \rhd E) \ , \tag{3}$$

where f is an arbitrary function. Moreover,

$$\begin{aligned} \mathit{true} \rhd E &= E \ , \\ \mathit{false} \rhd E &= \mathsf{error} \ , \\ \mathsf{error} \rhd E &= \mathsf{error} \ , \end{aligned}$$

and, for determinate Boolean expression B,

$$\begin{aligned} & \mathsf{if}\ B\,\mathsf{then}\,E\,\mathsf{else}\,F \\ =\ & \mathsf{if}\ B\,\mathsf{then}\,B \rhd E\,\mathsf{else}\,F \\ =\ & \mathsf{if}\ B\,\mathsf{then}\,E\,\mathsf{else}\,\neg B \rhd F \ . \end{aligned} \tag{4}$$

Of course there are useful combinations of (3) and (4).

2.2 Assertions as Restrictions

We mainly use assertions as parameter restrictions for functions (cf.[3]).

Definition 2.1 Let A be an assertion possibly involving the identifier x. The *partial identity* belonging to A is

$$id_A \overset{\text{def}}{=} \lambda x.\, A \rhd x \ .$$

A function with its parameter *restricted* by A is then

$$\lambda x : A.\, E \overset{\text{def}}{=} (\lambda x.\, E) \circ id_A \ .$$

Here, $\circ$ is the usual function composition given by $(f \circ g)(x) = f(g(x))$. The definition using the composition operator encapsulates the particular underlying semantics. In case of a strict semantics we have for $f \overset{\text{def}}{=} \lambda x : A.\, E$ that

$$f \ = \ \lambda x.\, A \rhd E \ . \tag{5}$$

This means that f is undefined for all arguments x that violate the restriction A, or, in other words, that A is a precondition for definedness of f. In case of a non-strict semantics, the assertion A is checked lazily, i.e., only if parameter x is actually used critically in evaluating E.

Characteristic properties of the restricting partial identities are, first, that they are weaker than the full identity and that they are idempotent under composition by (2). Moreover, the elements that satisfy the assertion are precisely the fixpoints of the corresponding restriction function.

Following [17], we shall call such functions *subtypes* and prove our laws about assertions for general subtypes in an algebraic style in Section 3.

2.3 Invariants

How can we now describe that an assertion holds for all recursive calls of a function and hence is "invariant" for that function? The key idea is to compare the behaviour of functions with and without the assertion. If it makes no difference whether we plug in one or the other in the place of recursive calls, then for each of them the assertion must be satisfied or the result of the call is irrelevant for the overall computation anyhow.

We introduce a higher-order function that provides a function with an assertion. We shall assume that the assertion is expressed by a unary predicate C.

Definition 2.2 The *partial identity* belonging to C is

$$id_C \stackrel{\text{def}}{=} \lambda x.\ C(x) \rhd x\ .$$

while the *restriction functional* for C is

$$F_C \stackrel{\text{def}}{=} \lambda f.\ f \circ id_C\ .$$

Consider now a declaration of the form

$$f \stackrel{\text{def}}{=} \mu G\ ,$$

where μ is the least fixpoint operator and G is the functional corresponding to the body of the recursively defined routine. Semantically, G "records" all the places where recursive calls occur. Hence we can describe a modified recursion where a check for assertion C is inserted for the argument of every recursive call by the functional $G \circ F_C$.

Checking assertion C for the argument of the original call of a function f amounts to evaluating $F_C(f)$. Informally, C is an invariant of a recursion when C holds for all recursive calls provided it holds for the initial call. Using functional notation, we therefore say

Definition 2.3 Assertion C is an *invariant* of functional G if

$$F_C \circ G \ = \ F_C \circ G \circ F_C\ .$$

This will be taken up at a more abstract level in the following section.

3 An Algebraic View of Subtypes and Assertions

3.1 Composition Algebras

We have shown how an assertion can be modeled by a particular object that represents the set of elements for which the assertion holds. Enforcing an assertion is done by composing with that object. The assertion holds for the range of a function or relation, if composing with that object on the range side does not change the function or relation; a dual remark applies to the domain side.

Since composition plays such an essential role, we define the notion of a composition algebra. We briefly repeat the necessary order-theoretic concepts. Let $(M, \leq)$ be a partial order. A subset $D \subseteq M$ is *directed* if it is non-empty and any two elements of D have a common upper bound in D. The order $(M, \leq)$ is called *complete* ("complete partial order", briefly *cpo*), if M has a least element and every directed subset of M has a supremum in M. A *complete lattice* is a partial order in which *every* subset has a supremum (and hence also an infimum). A complete lattice also is a cpo.

Let $(M, \leq)$ and $(N, \leq)$ be partial orders. A (total) map $f : M \to N$ is *monotonic*, if for all $x, y \in M$ with $x \leq y$ also $f(x) \leq f(y)$. Assume now that $(M, \leq)$ and $(N, \leq)$ are cpos. Then f is *strict* if it preserves the least element, and *continuous* if it is monotonic and preserves the suprema of all directed subsets of M. If $(M, \leq)$ and $(N, \leq)$ are complete lattices then map $F : M \to N$ is *universally disjunctive* if it preserves the suprema of all subsets of M. A universally disjunctive map is both strict and continuous. Now we are ready for

Definition 3.1 A *composition algebra* is a quintuple $C = (M, \cdot, 1, \leq, 0)$ such that
– $(M, \cdot, 1)$ is a monoid; the operation $\cdot$ is called *composition*;
– $(M, \leq)$ is a cpo with least element 0;
– the operation $\cdot$ is monotonic in both arguments.
The algebra C is called *left-continuous* if $\cdot$ is continuous in its left argument. *Right-continuity* is defined analogously. If C is both left-continuous and right-continuous it is called *continuous*.

A prominent example of a composition algebra is the set of all monotonic functions between two cpos under function composition and the usual pointwise extension of the approximation ordering. This algebra is left-continuous but not right-continuous. A continuous algebra is obtained by considering continuous functions only. However, a number of specification constructs, such as the quantifiers in CIP-L [3], lead to functions that are monotonic but not continuous, and so the more general notion has its use.

Prompted by this example, we view in all composition algebras the range of an element as being "on its left", its domain as being "on its right".

Note that, by monotonicity, multiplication with a subidentity is strict:

$$x \leq 1 \Rightarrow x \cdot 0 = 0 = 0 \cdot x \ . \tag{6}$$

A particular and important class of composition algebras is introduced by (see e.g. [6])

Definition 3.2 A composition algebra $C = (M, \cdot, 1, \leq, 0)$ is called a *Kleene algebra* if $(M, \leq)$ is even a complete lattice and $\cdot$ is universally disjunctive. In Kleene algebras the general supremum operator is usually denoted by Σ, the binary one by $+$.

All Kleene algebras are continuous. Important examples are

1. the power set of a set under intersection or union and the inclusion ordering;
2. the set of all binary relations on a set under relational composition and the inclusion ordering;
3. the set of all formal languages over some alphabet under concatenation and the inclusion ordering;
4. the Kleene algebra of all sets of paths in a directed graph under the pointwise extension of path concatenation and the inclusion ordering;
5. all sequential algebras [9].

It turns out that the following relations are of interest for many properties; they reflect a kind of subsumption. In the power set Kleene algebra they coincide with inclusion.

Definition 3.3 The relations $\preceq_d, \preceq_r \subseteq M \times M$ are defined by

$$\begin{aligned} x \preceq_d y &\overset{\text{def}}{\Leftrightarrow} x \cdot y = x \ , \\ x \preceq_r y &\overset{\text{def}}{\Leftrightarrow} y \cdot x = x \ . \end{aligned}$$

Hence $x \preceq_d y$ means that the domain of x is stable under y; it implies that y is idempotent on the domain of x. Analogous remarks hold for $\preceq_r$. The following properties are immediate:

Lemma 3.4 1. For all $x \in M$ we have $x \preceq_d 1$ and $x \preceq_r 1$.
2. $\preceq_d$ and $\preceq_r$ are transitive.
3. $x \preceq_d x$ iff $x \preceq_r x$ iff x is idempotent, i.e., $x \cdot x = x$.
4. Suppose x and y commute, i.e., $x \cdot y = y \cdot x$. Then $\preceq_d$ and $\preceq_r$ are antisymmetric on x, y, i.e., $x \preceq_d y \wedge y \preceq_d x \Rightarrow x = y$ and $x \preceq_r y \wedge y \preceq_r x \Rightarrow x = y$.
5. If $x \preceq_d y$ then also $z \cdot x \preceq_d y$ for all $z \in M$. If $x \preceq_r y$ then also $x \cdot z \preceq_r y$ for all $z \in M$.

Next, we introduce ternary relations that express weak commutativity properties:

Definition 3.5 For $a, b, c \in M$ we set

$$\begin{aligned} x : y \rightharpoonup z &\overset{\text{def}}{\Leftrightarrow} x \cdot y \leq z \cdot x \ , \\ x : y \leftharpoondown z &\overset{\text{def}}{\Leftrightarrow} z \cdot x \leq x \cdot y \ , \\ x : y \rightleftharpoons z &\overset{\text{def}}{\Leftrightarrow} x : y \rightharpoonup z \wedge x : y \leftharpoondown z \\ &\Leftrightarrow x \cdot y = z \cdot x \ . \end{aligned}$$

Roughly, the first relation says that the range of the restriction of x to y is contained in z. Analogous remarks apply to the other relations. A similar definition occurs in [7]. Connections between these relations and $\preceq_d, \preceq_r$ will be established in a later section. Their use is illustrated in the following fusion rules for least fixpoints. Consider monotonic functions $f, g, h : M \to M$, where $(M, \leq)$ is a cpo, under function composition $\circ$. As usual, we denote the least fixpoint of a function k by μk. We have

$$\text{(Right Subfusion)} \quad \frac{g : h \leftharpoondown f}{\mu f \leq g(\mu h)}$$

$$\text{(Left Subfusion)} \quad \frac{\begin{array}{l} f \text{ continuous and strict} \\ f : g \rightharpoonup h \end{array}}{f(\mu g) \leq \mu h}$$

$$\text{(Fusion)} \quad \frac{\begin{array}{l} f \text{ continuous and strict} \\ f : g \rightleftharpoons h \end{array}}{f(\mu g) = \mu h}$$

The right subfusion rule is immediate from the fact that μf is also the least element x with $f(x) \leq x$. Contrarily, the proof of the left subfusion rule needs fixpoint induction. For the case of $(M, \leq)$ being a complete lattice, a different derivation using the theory of Galois connections is given in [11]; then f has to be universally disjunctive. The fusion rule, finally, is immediate from the two subfusion rules.

3.2 Pretypes

We now return to the topic of assertions. As already stated, it is characteristic of restriction functions representing assertions that they are idempotent subidentities (see also [17]). Idempotence is also characteristic of so-called retractions and used centrally in Scott's classical paper [19] for the definition of data types: each such type is viewed as the range or, equivalently, the set of fixpoints of a retraction. We shall call idempotent subidentities pretypes, since under additional assumptions they will play the role of subtypes.

Definition 3.6 The set of *pretypes* of composition algebra C is

$$PT(C) \overset{\text{def}}{=} \{x \in M : x \leq 1 \wedge x \leq x \cdot x\} \ .$$

We list the pretypes for some of the Kleene algebras mentioned above:

1. for binary relations the pretypes are exactly the subrelations of the identity, called monotypes in [1, 7, 8];
2. for formal languages we only have ε, the language consisting of the empty word, and $\emptyset$, i.e., just 1 and 0;
3. for sets of paths in a graph with vertex set V the pretypes are exactly the subsets of $\varepsilon \cup V$, i.e., sets of paths with at most one vertex.

The following properties are immediate from the definition, (6), monotonicity of composition, neutrality of 1 and antisymmetry of $\leq$:

$$0, 1 \ \in \ PT(C) \ , \tag{7}$$

$$x \in PT(C) \wedge y \in M \ \Rightarrow \ x \cdot y \leq y \wedge y \cdot x \leq y \ , \tag{8}$$

$$x \in PT(C) \ \Rightarrow \ x \cdot x \ = \ x \ . \tag{9}$$

In the relation or function model, a pretype is characterized by its set of fixpoints which coincides with its range. We call multiplication by a pretype on the right *restriction* and multiplication on the left *corestriction* by that pretype. Hence for pretype y the relation $x \preceq_r y$ means that corestriction of x by y does not change x, so that the range of x is contained in the characteristic set of y. A dual remark holds for the $\preceq_d$ relation.

It turns out that for pretypes the subsumption relations coincide with the ordering $\leq$:

Lemma 3.7 For $x, y \in PT(C)$ we have

$$x \preceq_d y \ \Leftrightarrow \ x \leq y \ \Leftrightarrow \ x \preceq_r y \ .$$

Proof: We show the second equivalence, the first one being dual.

$$\begin{array}{ll} & x \leq y \\ \Rightarrow & \quad \{\!\![\text{ monotonicity of composition }]\!\!\} \\ & x \cdot x \leq y \cdot x \\ \Rightarrow & \quad \{\!\![\ x \in PT(C) \text{ and transitivity }]\!\!\} \\ & x \leq y \cdot x \\ \Rightarrow & \quad \{\!\![\text{ by (8) and antisymmetry }]\!\!\} \\ & x = y \cdot x \\ \Rightarrow & \quad \{\!\![\text{ reflexivity }]\!\!\} \\ & x \leq y \cdot x \\ \Rightarrow & \quad \{\!\![\text{ by (8) and transitivity }]\!\!\} \\ & x \leq y \ . \end{array}$$

■

The composition of two pretypes is their infimum, provided it is a pretype:

Lemma 3.8 Suppose $X \subseteq PT(C)$ and $x, y \in X$ such that $x \cdot y \in X$. Then $x \cdot y \ = \ x \sqcap_X y$.

Proof: From (8) we get $x \cdot y \leq x, y$. Assume now $z \leq x, y$ for some $z \in X$. Then $z \ = \ z \cdot z \leq x \cdot y$ by (9) and monotonicity of composition. ■

The infimum, i.e., the composition, of two pretypes plays the role of the conjunction of the corresponding assertions. We have the property

Corollary 3.9 The composition of two pretypes is a pretype iff they commute under composition.

Proof: ($\Rightarrow$) follows from the previous lemma and the commutativity of the binary infimum operator.
($\Leftarrow$) Assume that $x, y \in PT(C)$ commute. By monotonicity of $\cdot$ and idempotence of 1 we get $x \cdot y \leq 1$. Moreover, since x and y commute, we have $x \cdot y \cdot x \cdot y = x \cdot x \cdot y \cdot y = x \cdot y \geq x \cdot y$. ■

The property that the composition of two pretypes is their infimum if they commute can already be found in [19], p. 541.

3.3 Pretype Propagation

We investigate now the relations $\rightharpoonup$, $\leftharpoondown$ and $\rightleftharpoons$ in the particular case of subtypes. For $x, y \in PT(C)$ and $a \in M$ the relation $a : x \rightharpoonup y$ means that the range of the restriction of a to x is included in y. A dual remark applies to $\leftharpoondown$. This is also shown by

Lemma 3.10 For $x, y \in PT(C)$ and $a \in M$ we have

$$\begin{aligned} a : x \leftharpoondown y &\Leftrightarrow y \cdot a \preceq_d x \ , \\ a : x \rightharpoonup y &\Leftrightarrow a \cdot x \preceq_r y \ . \end{aligned}$$

Proof: We show the second equivalence, the first one being dual.

$$\begin{array}{ll} & a \cdot x \leq y \cdot a \\ \Rightarrow & \quad \{\![\text{ monotonicity }]\!\} \\ & a \cdot x \cdot x \leq y \cdot a \cdot x \\ \Rightarrow & \quad \{\![\text{ idempotence of } x \ ]\!\} \\ & a \cdot x \leq y \cdot a \cdot x \\ \Rightarrow & \quad \{\![\text{ by (8) }]\!\} \\ & y \cdot a \cdot x = a \cdot x \\ \Rightarrow & \quad \{\![\text{ by (8) }]\!\} \\ & a \cdot x \leq y \cdot a \ . \end{array}$$

■

Assume now $x, y, u, v, x \cdot u, y \cdot v \in PT(C)$. Then we have the following immediate inference rules for forming conjunctions:

$$\text{(Pretype Conjunction I)} \ \ \frac{a : x \rightharpoonup y \qquad a : u \rightharpoonup v}{a : x \cdot u \rightharpoonup y \cdot v}$$

$$\text{(Pretype Conjunction II)} \quad \frac{a : x \leftharpoondown y \qquad a : u \leftharpoondown v}{a : x \cdot u \leftharpoondown y \cdot v}$$

With the help of the $\rightharpoonup$ and $\leftharpoondown$ relations it is straightforward to characterize invariants:

Definition 3.11 A pretype $x \in PT(C)$ is an *invariant* of $a \in M$ if $a : x \rightharpoonup x$; it is a *co-invariant* of a if $a : x \leftharpoondown x$.

Corollary 3.12 A type $x \in PT(C)$ is an invariant of $a \in M$ iff $a \cdot x = x \cdot a \cdot x$ iff $x : a \cdot x \rightleftharpoons a$. Similarly, x is a co-invariant of a iff $x \cdot a = x \cdot a \cdot x$ iff $x : a \rightleftharpoons x \cdot a$.

Proof: immediate from Lemma 3.10. ■

3.4 Subtypes

The above inferences have shown us the need for sets of pretypes that are closed under composition, i.e., under conjunction.

Definition 3.13 We call $T \subseteq PT(C)$ a *set of subtypes* if $1 \in T$ and T is closed under composition.

Hence a set of subtypes is a submonoid of $(M, \cdot, 1)$, and $\{1\}$ is the smallest set of subtypes. Moreover, by Corollary 3.9, all elements of a set of subtypes commute.

How can we distinguish sets of subtypes? A sufficient criterion is the following, which in the case of the function algebra was given in [17] at the point level.

Definition 3.14 A pretype $x \in PT(C)$ is called *downward closed* if for all $a \in M$ we have $a \leq x \Rightarrow a \preceq_r x$.

In the function algebra this means that for every fixpoint of x also all smaller elements are fixpoints of x. Note that 1 is downward closed. Since the characteristic set of a pretype is considered to be its range, we do not bother to define a dual notion involving domains.

Lemma 3.15 Let x, y be downward closed pretypes. Then $z \overset{\text{def}}{=} x \cdot y$ is a downward closed pretype as well.

Proof: By (8) we have $z \leq x, y$ and hence also $z \leq 1$. Moreover, by downward closedness of x, y we have $z \preceq_r x, y$. Now

$$\begin{aligned}
& z \cdot z \\
= \quad & \{\![\text{ definition }]\!\} \\
& x \cdot y \cdot z \\
= \quad & \{\![\text{ since } z \preceq_r y \ ]\!\} \\
& x \cdot z \\
= \quad & \{\![\text{ definition }]\!\} \\
& x \cdot x \cdot y \\
= \quad & \{\![\text{ idempotence of } x \ ]\!\} \\
& x \cdot y \\
= \quad & \{\![\text{ definition }]\!\} \\
& z \ .
\end{aligned}$$

So z is a pretype. Assume now $a \leq z$. Then also $a \leq x, y$ and

$$\begin{aligned}
& z \cdot a \\
= \quad & \{\![\text{ definition }]\!\} \\
& x \cdot y \cdot a \\
= \quad & \{\![\ y \text{ downward closed }]\!\} \\
& x \cdot a \\
= \quad & \{\![\ x \text{ downward closed }]\!\} \\
& a \ .
\end{aligned}$$

■

Corollary 3.16 The set of all downward closed pretypes is a set of subtypes.

From now on we assume a fixed set $T \subseteq PT(C)$ of subtypes.

3.5 Algebras of Functionals

We now lift our notions to the level of functions over composition algebras. Since for the functional and relational case these functions are of second order, we speak of algebras of functionals.

Definition 3.17 Given a composition algebra $C = (M, \cdot, 1, \leq, 0)$, we define the *algebra of functionals* over C as $F(C) = ([M \to M], \circ, id_M, \leq, \Omega)$ where $[M \to M]$ is the set of all monotonic functions from M into itself, $\circ$ is the usual function composition, id_M is the identity on M, $\leq$ is the pointwise function ordering and Ω is the constant 0-valued function.

It is clear that this defines a left-continuous composition algebra. We now prepare the lifting of subtypes to the level of functionals:

Definition 3.18 For $a \in M$ we define $F_a \in [M \to M]$ as the restriction functional $F_a(b) \overset{\text{def}}{=} b \cdot a$.

In the discussion about assertions in Section 2 only restriction was used. For that reason we only deal with that operation here; a dual treatment for co-restriction is obvious.

Lemma 3.19 For all $a, b \in M$ we have $F_{a \cdot b} = F_b \circ F_a$ and $F_a \leq F_b \Leftrightarrow a \leq b$. If C is left-continuous then all F_a are continuous. If $x \in PT(C)$ then $F_x \in PT(F(C))$ and F_x is strict. Finally, for a set T of subtypes over C the set $F(T) \overset{\text{def}}{=} \{F_x : x \in T\}$ is a set of subtypes over $F(C)$.

Proof: The first two claims are obvious. For $x \in PT(C)$ the first claim implies $F_x \in PT(F(C))$, whereas strictness follows from (6). The last claim follows from the first one and the definitions. ■

Invariants are now also lifted to the functional level by generalising Definition 2.3:

Definition 3.20 We call a subtype $x \in T$ an *invariant* of $G \in [M \to M]$ if

$$F_x \circ G = F_x \circ G \circ F_x \ .$$

A more concise statement of the invariance property is given in

Corollary 3.21 x is an invariant of G iff $F_x : G \rightleftharpoons F_x \circ G$ iff $G : F_x \leftharpoondown F_x$, i.e., if F_x is a co-invariant of G.

Proof: Immediate from Corollary 3.12. ■

This odd reversal seems unavoidable.

3.6 Invariants and Recursion

The semantics of a recursive definition over a composition algebra is, as usual, given as the least fixpoint of the associated functional in the respective algebra of functionals. We investigate in this section how invariants influence least fixpoints.

Frequently, one has to strengthen an invariant to achieve a certain simplification. This is done by showing that the conjunction of the original invariant with another assertion is an invariant, too. One then passes to a recursion with the stronger invariant and performs the simplification exploiting it. However, after that, one may want to get rid of the assertion again, in particular, when assertions are implemented as actually executed checks. Over a left-continuous composition algebra, a tool for this is the following inference rule which is an instance of the fusion rule of Section 3.1:

$$\text{(Invariant)} \quad \frac{x \in PT(C) \text{ invariant for } G}{F_x(\mu G) \;=\; \mu(F_x \circ G)}$$

The first premise of the fusion rule is satisfied by Corollary 3.21 and strictness and continuity of F_x (see Lemma 3.19). Let us interpret this rule in the algebra of functionals over monotonic functions: on the left hand side of the conclusion, x is only checked for the argument of the first call to μG, whereas on the right it is also checked for all recursive calls. So passing from left to right means introduction of an invariant, the reverse direction its removal.

Conjunctions of assertions can be introduced by iterated application of this rule using the property

$$F_x \circ F_y \;=\; F_{x \sqcap_T y}$$

for subtypes x, y, which follows from Lemma 3.8 and Lemma 3.19. This allows then weakening and strengthening assertions.

As an application we show a property of Kleene algebras. To this end we recall that for $a, b \in M$ we have

$$a \cdot b^* \;=\; \mu G_{ab} \;, \tag{10}$$

where $G_{ab}(x) \overset{\text{def}}{=} a + x \cdot b$. In particular, $b^* \overset{\text{def}}{=} \mu G_{1b}$.

Lemma 3.22 Let C be a Kleene algebra and assume that $a \in M$ is an invariant of $b \in M$. Then

$$b^* \cdot a = a \cdot (b \cdot a)^* \;.$$

Proof: We first show that a is an invariant of G_{1b}:

$$\begin{array}{ll}
 & (F_a \circ G_{1b} \circ F_a)(x) \\
= & \quad \{\!\![\text{ definitions }]\!\!\} \\
 & (1 + x \cdot a \cdot b) \cdot a \\
= & \quad \{\!\![\cdot \text{ universally disjunctive, 1 neutral }]\!\!\} \\
 & a + x \cdot a \cdot b \cdot a \\
= & \quad \{\!\![\; a \text{ invariant of } b \;]\!\!\} \\
 & a + x \cdot b \cdot a \\
= & \quad \{\!\![\cdot \text{ universally disjunctive, 1 neutral }]\!\!\} \\
 & (1 + x \cdot b) \cdot a \\
= & \quad \{\!\![\text{ definitions }]\!\!\} \\
 & (F_a \circ G_{1b})(x) \;.
\end{array}$$

The last three steps also show $F_a \circ G_{1b} \;=\; G_{a,(b \cdot a)}$. Now the claim is immediate from the previous lemma and (10). ■

Corollary 3.23 Under the above assumptions,

$$b^* \cdot a = a \cdot b^* \cdot a\ .$$

Proof:

$$\begin{array}{ll} & b^* \cdot a \\ = & \quad \{\!\![\ \text{above lemma}\ ]\!\!\} \\ & a \cdot (b \cdot a)^* \\ = & \quad \{\!\![\ a \text{ idempotent}\ ]\!\!\} \\ & a \cdot a \cdot (b \cdot a)^* \\ = & \quad \{\!\![\ \text{above lemma}\ ]\!\!\} \\ & a \cdot b^* \cdot a\ . \end{array}$$

■

In the path Kleene algebra this means for a set R of edges and a set S of vertices, viewed as singleton paths, the following: If all edges of R that start in S also end in S, then all paths using only edges in R that start in S will pass only through vertices in S and end in S. This is used in [18] for the simplification of a general scheme for layer-oriented graph traversal.

4 Assertions Revisited

We now apply the general results of the previous section to the particular case of subtypes generated by assertions. In particular, we are interested in invariants. For a very simple case we can give a sufficient criterion:

Lemma 4.1 Suppose G is a functional in f such that, for fixed binary g and unary h, k, we have $G(f) = g \circ [k, f \circ h]$, where $[f_1, f_2] \overset{\text{def}}{=} \lambda x.\,(f_1(x), f_2(x))$. Moreover, let C be a unary predicate. If id_C is an invariant of h, then C is an invariant of G.

Proof: We calculate

$$\begin{array}{ll} & (F_C \circ G)(f) \\ = & \quad \{\!\![\ \text{definition } F_C \text{ and } G\ ]\!\!\} \\ & g \circ [k, f \circ h] \circ id_C \\ = & \quad \{\!\![\ \text{distributivity}\ ]\!\!\} \\ & g \circ [k \circ id_C, f \circ h \circ id_C] \\ = & \quad \{\!\![\ id_C \text{ invariant of } h\ ]\!\!\} \\ & g \circ [k \circ id_C, f \circ id_C \circ h \circ id_C] \\ = & \quad \{\!\![\ \text{distributivity}\ ]\!\!\} \\ & g \circ [k, f \circ id_C \circ h] \circ id_C \end{array}$$

$$\begin{array}{ll} = & \{\![\ \text{definition } F_C \text{ and } G\ ]\!\} \\ & (F_C \circ G \circ F_C)(f)(x)\ . \end{array}$$

■

Example 4.2 Consider the recursion

$$\mathsf{letrec}\ f(x)\ =\ \mathsf{if}\ x \leq 1\, \mathsf{then}\, 0\, \mathsf{else}\, f(x-2)\ .$$

The associated functional is

$$G \overset{\text{def}}{=} \lambda f.\, \lambda x.\, \mathsf{if}\ x \leq 1\, \mathsf{then}\, 0\, \mathsf{else}\, f(x-2)\ .$$

Taking $g(x,y) \overset{\text{def}}{=} \mathsf{if}\ x \leq 1\, \mathsf{then}\, 0\, \mathsf{else}\, y$ and $h(x) \overset{\text{def}}{=} x-2$ we see that by the previous lemma both the predicates *even* and *odd* are invariants of G. ■

This criterion generalizes in a straightforward way to an $n+1$-ary function g and n functions h_i:

Lemma 4.3 Suppose $G(f)\ =\ g \circ [k, f \circ h_1, \ldots, f \circ h_n]$ such that id_C is an invariant of h_i for all $i \in \{1, \ldots, n\}$. Then C also is an invariant of G.

Example 4.4 Consider the recursion

$$\mathsf{letrec}\ \mathit{oddtree}(i)\ =\ \langle \mathit{oddtree}(i+2), i, \mathit{oddtree}(i-2)\rangle\ ,$$

where $\langle ., ., . \rangle$ is a constructor for node-labelled binary trees. It builds an infinite tree. The predicates *even* and *odd* also are invariants of the associated functional *ODD* with

$$\mathit{ODD}(f)(i)\ =\ \langle f(i+2), i, f(i-2)\rangle\ .$$

■

For examples where proving an invariant also involves rule (4) see [12].

Example 4.5 As an example of the use of the invariant rule we choose the improvement of a shortest path algorithm (cf. [13] for the derivation). We start from the version

```
let $\mathit{shortestpath}\ =\ \lambda x, y.\ \mathit{sp}(\{x\}, \emptyset, y)$ where

letrec $\mathit{sp}\ =\ \lambda S, V, y : S \cap V\ =\ \emptyset \wedge y \notin V\ .$
         if $y \in S$
         then 0
         else if $\mathit{succ}(S) \cup \mathit{succ}(V)\ \subseteq\ S \cup V$
                then $\infty$
                else $1 + \mathit{sp}((\mathit{succ}(S) \cup \mathit{succ}(V)) \backslash (S \cup V)),\ S \cup V,\ y)$ .
```

To calculate the length of a shortest path from vertex x to vertex y, where each edge has cost 1, we embed the problem into the function *sp*. It computes the length of a shortest path from vertex set S, the set of vertices to be inspected in the current round, to y along paths not leading through V, the set of vertices already visited. The function *succ* calculates the set of immediate successors of a set of vertices. The algorithm is very inefficient, since it always forms the union $succ(S) \cup succ(V)$. However, a simple analysis shows that the invariant of *sp* can be strengthened by the conjunct $succ(V) \subseteq S \cup V$. Using this, the body can be simplified considerably and we obtain

```
let $shortestpath \;=\; \lambda x, y.\, sp(\{x\}, \emptyset, y)$  where

letrec  $sp \;=\; \lambda S, V, y : S \cap V \;=\; \emptyset \wedge y \notin V \wedge succ(V) \subseteq S \cup V$ .
        if $y \in S$
        then 0
        else  if $succ(S) \subseteq S \cup V$
              then $\infty$
              else  $1 + ssp(succ(S) \backslash (S \cup V),\, S \cup V,\, y)$ .
```

Afterwards, for passing to a loop we may remove the invariant again. ■

Next we give an example for the case of non-strict semantics.

Example 4.6 Consider a function for constructing the infinite list of squares of integers starting from an integer x:

$$squares = \lambda x.\, [x^2] \mathbin{+\!\!+} squares(x+1) \; .$$

In this recursion the square of each number is computed separately. We can speed up generation of the numbers in the list using the technique of finite differencing (see e.g. [16]). To this end we define a version of *squares* with an additional parameter and an assertion:

$$esquares = \lambda x, s : s = x^2 \; . \; squares(x) \; .$$

A simple unfold/fold transformation shows that we have the recursion

$$esquares = \lambda x, s : s = x^2 \; . \; [s] \mathbin{+\!\!+} esquares(x+1, s+2*x+1) \; ,$$

in which the squares are now computed incrementally. ■

5 Enforcing Relations and Recurrence

Invariants are unary predicates that treat the parameters of recursions in isolation. In some circumstances, however, one wants to relate the parameters of the recursive calls with that of the parent call using a binary relation $R \subseteq T \times T$. For R and $x \in T$ we denote by Rx the residual predicate which holds for $y \in T$ iff $(y, x) \in R$.

Definition 5.1 The *restricting functional* associated with R is

$$F_R \overset{\text{def}}{=} \lambda f, x.\, f \circ id_{Rx} \ .$$

We say that a functional $H : [T \to T] \to [T \to T]$ *enforces* R if

$$\forall\ f : \forall\ x : H(f)(x) \ = \ H(F_R(f, x), x) \ .$$

So H enforces R if the arguments of all recursive calls are in relation R with the argument of the parent call.

Example 5.2 Consider the recursion

$$\mathsf{letrec}\ \mathit{heaptree}(n) \ = \ \langle \mathit{heaptree}(2 * n), n, \mathit{heaptree}(2 * n + 1) \rangle \ .$$

It builds an infinite tree where the nodes carry the indices of a heap-organized infinite array, starting with index n. The associated functional HEAP with

$$\mathit{HEAP}(f, n) \ = \ \langle f(2 * n), n, f(2 * n + 1) \rangle$$

enforces the relation R given by

$$yRx \overset{\text{def}}{\Leftrightarrow} x = y\ \mathit{div}\ 2 \ .$$

■

As another application, we consider termination proofs using noetherian relations.

Definition 5.3 A relation $\ll \subseteq T \times T$ is *noetherian* if there is no infinite sequence $(x_i)_{i \in \mathbb{N}}$ with $x_{i+1} \ll x_i$ for all $i \in \mathbb{N}$. We call a functional H $\ll$*-recurrent* if H enforces $\ll$.

For a noetherian relation $\ll$ and a unary predicate P on T one has the following proof principle:

$$\text{(Noetherian Induction)} \quad \frac{\forall\ u \in T : (\forall\ w \in T : w \ll u \ \Rightarrow \ P(w)) \ \Rightarrow \ P(u)}{\forall\ u \in T : P(u)}$$

The restricting functional associated with a noetherian relation $\ll$ is

$$N_{\ll} \overset{\text{def}}{\Leftrightarrow} \lambda f, x.\ f \circ id_{\ll x} \ ,$$

and so H is $\ll$-recurrent iff

$$\forall\ f : \forall\ x : H(f)(x)\ =\ H(N_{\ll}(f,x))(x)\ .$$

A related notion is discussed in [7, 8].

Example 5.4 Consider Ackermann's function

$$\begin{array}{l}
\mathsf{letrec}\ ack(x,y) = \mathsf{if}\ x = 0\,\mathsf{then}\, y+1 \\
\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ \mathsf{if}\ y = 0\,\mathsf{then}\, ack(x-1,1) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ \ ack(x-1, ack(x, y-1))
\end{array}$$

of type $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$. The corresponding functional is ACK with

$$\begin{array}{l}
ACK(f)(x,y) = \mathsf{if}\ x = 0\,\mathsf{then}\, y+1 \\
\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ \mathsf{if}\ y = 0\,\mathsf{then}\, f(x-1,1) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ \ f(x-1, f(x, y-1))\ .
\end{array}$$

We want to show that ACK is recurrent. We choose for $\ll$ the lexicographic order on $\mathbb{N} \times \mathbb{N}$, viz.

$$(x_1, y_1) \ll (x_2, y_2) \overset{\text{def}}{\Leftrightarrow} x_1 < x_2 \vee (x_1 = x_2 \wedge y_1 < y_2)\ .$$

We calculate, assuming a strict semantics,

$$\begin{array}{ll}
 & ACK(N_{\ll}(f, (x,y)))(x,y) \\
= & \quad \{\![\ \text{definition of } ACK \text{ and } N_{\ll} \text{ and } (5)\ ]\!\} \\
 & \mathsf{if}\ x = 0 \\
 & \quad \mathsf{then}\, y+1 \\
 & \quad \mathsf{else}\ \mathsf{if}\ y = 0 \\
 & \qquad\quad \mathsf{then}\, (x-1,1) \ll (x,y) \triangleright f(x-1,1) \\
 & \qquad\quad \mathsf{else}\ \ (x-1, f(x,y-1)) \ll (x,y) \triangleright \\
 & \qquad\qquad\quad f(x-1,\ (x,y-1) \ll (x,y) \triangleright f(x,y-1)) \\
= & \quad \{\![\ \text{definition of } \ll\ ]\!\} \\
 & \mathsf{if}\ x = 0\,\mathsf{then}\, y+1 \\
 & \qquad\quad \mathsf{else}\ \mathsf{if}\ y = 0\,\mathsf{then}\, \mathit{TRUE} \triangleright f(x-1,1) \\
 & \qquad\qquad\qquad\qquad \mathsf{else}\ \ \mathit{TRUE} \triangleright f(x-1,\ \mathit{TRUE} \triangleright f(x,y-1)) \\
= & \quad \{\![\ \text{by } (1)\ ]\!\} \\
 & \mathsf{if}\ x = 0\,\mathsf{then}\, y+1 \\
 & \qquad\quad \mathsf{else}\ \mathsf{if}\ y = 0\,\mathsf{then}\, f(x-1,1) \\
 & \qquad\qquad\qquad\qquad \mathsf{else}\ \ f(x-1, f(x,y-1)) \\
= & \quad \{\![\ \text{definition of } ACK\ ]\!\} \\
 & ACK(f)(x,y)\ .
\end{array}$$

■

6 Properties of Recurrent Recursions

We now want to analyse how certain relations propagate through formation of least fixpoints. To this end, consider a transitive binary relation $\leadsto$ (e.g., refinement of one element by another) on the target domain of the function space under consideration, such that $\bot \leadsto \bot$.

Definition 6.1 We lift $\leadsto$ pointwise to functions by setting

$$f \leadsto g \overset{\text{def}}{\Leftrightarrow} \forall\, x : f(x) \leadsto g(x)\ .$$

A functional F is *$\leadsto$-monotonic* if

$$\forall\, f, g : f \leadsto g \;\Rightarrow\; F(f) \leadsto F(g)\ .$$

Now we can show

Theorem 6.2 Let H be a functional that is $\leadsto$-monotonic and $\ll$-recurrent. If, for some function f we have $f \leadsto H(f)$ and g is a fixpoint of H then also $f \leadsto g$.

Proof: We perform a noetherian induction on the predicate

$$P(x) \overset{\text{def}}{\Leftrightarrow} f(x) \leadsto g(x)\ .$$

Suppose $\forall\, y : y \ll x \;\Rightarrow\; P(y)$. By $\bot \leadsto \bot$ this implies

$$N_{\ll}(f, x) \;\leadsto\; N_{\ll}(g, x)\ . \tag{11}$$

Now,

$$\begin{array}{ll}
 & f(x) \\
\leadsto & \quad \{\![\ \text{by the assumption}\ ]\!\} \\
 & H(f)(x) \\
= & \quad \{\![\ H \ll\text{-recurrent}\ ]\!\} \\
 & H(N_{\ll}(f, x))(x) \\
\leadsto & \quad \{\![\ \text{by (11) and } \leadsto\text{-monotonicity of } H\ ]\!\} \\
 & H(N_{\ll}(g, x))(x) \\
= & \quad \{\![\ H \ll\text{-recurrent}\ ]\!\} \\
 & H(g)(x) \\
= & \quad \{\![\ g \text{ a fixpoint of } H\ ]\!\} \\
 & g(x)\ .
\end{array}$$

■

This has been used in [4] to prove a declarative analogue of the usual inference rule for while-loops.

Corollary 6.3 (Unique Fixpoint) Let H be a $\ll$-recurrent functional. Then H has at most one fixpoint.

Proof: Take $=$ for $\leadsto$ in the above theorem. ∎

Next, we want to talk about preservation of certain properties. For instance, a function is total at an argument x if it preserves definedness of argument x:

$$TOT(f,x) \overset{\text{def}}{\Leftrightarrow} (DEF(x) \Rightarrow DEF(f(x))) \ .$$

In a deterministic setting the definedness predicate DEF may be defined as

$$DEF(x) \overset{\text{def}}{\Leftrightarrow} x \neq \mathsf{error} \ ;$$

in a non-deterministic setting the precise definition depends on the particular model of choice (erratic, angelic, demonic). Based on TOT we define

$$TOTAL(f) \overset{\text{def}}{\Leftrightarrow} \forall\ x : TOT(f,x) \ .$$

Consider now, more generally a unary predicate P. The predicate

$$PRES_P(f,x) \overset{\text{def}}{\Leftrightarrow} (P(x) \Rightarrow P(f(x)))$$

then expresses that f preserves P at argument x. In particular, $TOT = PRES_{DEF}$.

Lemma 6.4 Let H be a functional and f a fixpoint of H. From

$$\forall\ g : \forall\ x : (\forall\ y : y \ll x \Rightarrow PRES_P(g,y)) \Rightarrow PRES_P(H(g),x) \qquad (12)$$

we may infer

$$\forall\ x : PRES_P(f,x) \ .$$

Proof: Straightforward noetherian induction on the predicate $Q(x) \overset{\text{def}}{\Leftrightarrow} PRES_P(f,x)$ using the fixpoint property of f. ∎

From this we obtain

Theorem 6.5 (Termination Criterion) Let $H : [T \to T] \to [T \to T]$ for flat domain T be $\ll$-recurrent and f be a fixpoint of H. Then from

$$\forall\ g : TOTAL(g) \Rightarrow TOTAL(H(g)) \qquad (13)$$

we may infer

$$TOTAL(f) \ .$$

Proof: We show that (12) in the previous lemma is satisfied for $TOT = PRES_{DEF}$. Consider an arbitrary x and assume $\forall\, y : y \ll x \Rightarrow TOT(g,y)$. Choose $u = g(\bot)$ if $g(\bot) \neq \bot$ and arbitrary in $T\backslash\{\bot\}$ otherwise, and define h by

$$h(y) \overset{\text{def}}{=} \begin{cases} g(y) \text{ if } y \ll x \vee y = \bot \ , \\ u \quad \text{otherwise.} \end{cases}$$

By flatness of T then h is monotonic. Moreover, it satisfies $\forall\, y : TOT(h,y)$ and

$$N_{\ll}(h,x) \;=\; N_{\ll}(g,x) \ . \tag{14}$$

Now

$$\begin{array}{ll} & TRUE \\ \Leftrightarrow & \quad \{\!\![\text{ by (13) }]\!\!\} \\ & TOT(H(h),x) \\ \Leftrightarrow & \quad \{\!\![\; H \ll\text{-recurrent }]\!\!\} \\ & TOT(H(N_{\ll}(h,x)),x) \\ \Leftrightarrow & \quad \{\!\![\text{ by (14) }]\!\!\} \\ & TOT(H(N_{\ll}(g,x)),x) \\ \Leftrightarrow & \quad \{\!\![\; H \ll\text{-recurrent }]\!\!\} \\ & TOT(H(g),x) \ . \end{array}$$

■

As applications of Corollary 6.3 and Theorem 6.5 we obtain that the functional ACK associated with Ackermann's function has a unique fixpoint which is a total function.

7 Conclusion

We have given an algebraic characterization of invariants and a general rule for strengthening and weakening them in recursions. The rule covers arbitrary types of recursion, not just linear ones, and is independent of the particular semantic framework used. It applies not only at the level of recursive programs, but also to other recursively defined entities, such as closures in Kleene algebras, which may be seen as data.

The particular approach to assertions leads to a nice calculational framework. Both the proofs of the basic properties and the actual manipulation of assertions largely become elegant (in)equational reasoning. The technique covers also proofs by noetherian induction, in particular, termination proofs, for general recursions.

Acknowledgements. Section 3 was greatly stimulated by [17]. The idea of the functional $N_{\ll}$ and the definition of the notion of being recurrent are due to U. Berger who, in turn, attributes inspiration to H. Schwichtenberg. Helpful comments were provided by R. Backhouse, R. Bird, M. Russling and the anonymous referees.

References

1. R.C. Backhouse, P. de Bruin, G. Malcolm, T.S. Voermans, J. van der Woude: Relational catamorphisms. In: B. Möller (ed.): Constructing programs from specifications. Proc. IFIP TC2/WG 2.1 Working Conference on Constructing Programs from Specifications, Pacific Grove, CA, USA, 13–16 May 1991. Amsterdam: North-Holland 1991, 287–318
2. F.L. Bauer, B. Möller, H. Partsch, P. Pepper: Formal program construction by transformations — Computer-aided, Intuition-guided Programming. IEEE Transactions on Software Engineering **15**, 165–180 (1989)
3. F.L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, H. Wössner: The Munich project CIP. Volume I: The wide spectrum language CIP-L. Lecture Notes in Computer Science **183**. Berlin: Springer 1985
4. R. Berghammer, H. Ehler, B. Möller: On the refinement of nondeterministic recursive routines by transformation. In: M. Broy, C.B. Jones (eds.): Programming concepts and methods. Amsterdam: North-Holland 1990, 53–71
5. R.S. Bird, O. de Moor: The algebra of programming. Prentice-Hall (to appear)
6. J.H. Conway: Regular algebra and finite machines. London: Chapman and Hall 1971
7. H. Doornbos, R. Backhouse: Induction and recursion on datatypes. In: B. Möller (ed.): Mathematics of Program Construction. Lecture Notes in Computer Science **947**. Berlin: Springer 1995, 242–281
8. H. Doornbos, R. Backhouse: Reductivity. Science of Computer Programming (to appear)
9. B. von Karger, C.A.R. Hoare: Sequential calculus. Information Processing Letters **53**, 123–130 (1995)
10. S.C. Kleene: Introduction to metamathematics. New York: van Nostrand 1952
11. Mathematics of Program Construction Group: Fixed-point calculus. Information Processing Letters **53**, 131-136 (1995)
12. B. Möller: Applicative assertions. In: J.L.A. van de Snepscheut (ed.): Mathematics of Program Construction. Lecture Notes in Computer Science **375**. Berlin: Springer 1989, 348–362
13. B. Möller, M. Russling: Shorter paths to graph algorithms. Science of Computer Programming **22**, 157–180 (1994)
14. B. Möller: Relations as a program development language. In: B. Möller (ed.): Constructing programs from specifications. Proc. IFIP TC2/WG 2.1 Working Conference on Constructing Programs from Specifications, Pacific Grove, CA, USA, 13–16 May 1991. Amsterdam: North-Holland 1991, 373–397
15. C.C. Morgan: Programming from Specifications. Prentice-Hall, 1990
16. H.A. Partsch: Specification and transformation of programs — A formal approach to software development. Berlin: Springer 1990

17. C. Runciman: Subtype constraints as first-class values. Talk given at 48th meeting of IFIP WG 2.1, Günzburg, Germany, October 1995. Paper in preparation
18. M. Russling: A general scheme for breadth-first graph traversal. In: B. Möller (ed.): Mathematics of Program Construction. Lecture Notes in Computer Science **947**. Berlin: Springer 1995, 380–398
19. D. Scott: Data types as lattices. SIAM J. Comp. **5**, 522–587 (1976)