Program Development in an Algebraic Setting*

Peter Pepper

Technische Universität Berlin
Fachbereich Informatik
e-mail: pepper@cs.tu-berlin.de

Abstract

We present a framework for program development, which is based on concepts from algebra and logic. In this framework, programming is viewed as a process that successively extends the program under consideration by adding new axioms or theorems to it. In this setting, axioms constitute design decisions, whereas theorems merely make deducible knowledge explicit.

Technically, this is achieved by combining concepts from two related areas: Algebraic specifications are used to represent programs, and Gentzen-style rules of natural deduction are used to represent derivation processes.

The paper presents the algebraic framework and a collection of characteristic derivation rules. A number of small examples shall illustrate the underlying methodology.

1 Instead of an Introduction: An Example

We start by presenting an example. Even though we have to rely on the reader's intuition throughout its presentation, such an example can give a good first impression of the basic concepts and principles of the overall approach. The disadvantage is, of course, that such an introductory example is by necessity very small. But toy examples can illustrate notations and techniques just as well as intricate programming problems.

The example: ("Binary lorgarithm":) Suppose that we have a positive integer $x > 0$ and that we want to know, how many digits will be required for its binary representation. In the sequel we will go through the derivation of a program that solves this little problem.

1. *Specification*: Our first task is to describe the problem precisely. This is achieved by the following specifications.

   Fun $blog : nat \to nat$
   Spc $blog(x) = n$
   Pre $x > 0$
   Post $2^{n-1} \leq x < 2^n$

* This work was partially sponsored by the German Ministry of Research and Technology (BMFT) as part of the project "KORSO – Korrekte Software".

The essential meaning of this notation should be self-explaining; details will be given in Sections 2 and 4.6.

Note: In the sequel we develop an implementation for this specification. Whether the specification itself actually characterizes our original problem "number of digits in the binary representation of x" is beyond the realm of our discussion here. In other words: If the specification fails to properly describe the given problem, then the implementation will not solve the problem. All that we ensure is the compatibility of specification and implementation.

2. *Deriving recurrence relations by use of induction*: The decisive step in our development is the derivation of a number of recurrence relations that together establish a recursive function declaration. To this end, we have to make a suitable case distinction for x.
   (a) *Let $x = 1$:* Then we can deduce:
   $2^{n-1} \leq 1 < 2^n$ [by postcondition]
   $\vdash n = 1$ [by arithmetic]
   (b) *Let $x \geq 2$:* Since we work in the context of binary number systems, the idea comes to mind to make an even/odd case distinction.
      i. *Let $x = 2y$:* Then we can calculate:

$$\begin{array}{ll} 2^{n-1} \leq 2y < 2^n & \text{[by postcondition]} \\ \vdash 2^{(n-1)-1} \leq y < 2^{n-1} & \text{[since } n \geq 2\text{]} \\ \vdash 2^{m-1} \leq x \div 2 < 2^m \wedge m = n-1 & \text{[since } x = 2y\text{]} \\ \vdash m = blog(x \div 2) \wedge n = m+1 & \text{[by postcondition]} \\ \vdash n = blog(x \div 2) + 1 & \end{array}$$

      ii. *Let $x = 2y + 1$:* Then we can calculate

$$\begin{array}{ll} 2^{n-1} \leq 2y+1 < 2^n & \text{[by postcondition]} \\ \vdash 2^{n-1} \leq 2y < 2^n & \text{[by arithmetic]} \\ \vdots & \text{[analogously to 2(b)i]} \\ \vdash n = blog(x \div 2) + 1 & \end{array}$$

   Since both cases 2(b)i and 2(b)ii yield the same result, the case distinction is not really necessary. Therefore we have
   $\vdash n = blog(x \div 2) + 1$
   Note: Some of the above derivation steps are only legal, because certain conditions are met due to the specification of *blog*. This will be elaborated in greater detail in Section 4.6.
3. *Recursive solution*: The case distinction from the previous step is complete. Moreover, the recurrence relations also meet the termination requirement (see below). Therefore we obtain the following recursive function declaration.

$$\begin{array}{lll} \textbf{Def } blog(x) = \textbf{If } x = 1 \textbf{ Then} & 1 & \\ \qquad\qquad\qquad\qquad\quad \textbf{Else} & blog(x \div 2) + 1 & \textbf{Fi} \end{array}$$

   The function $x \div 2$ on the argument position of the recursive call establishes a strictly decreasing sequence of values, ending in the termination case $x = 1$. Hence our function is totally correct.

4. *Improving the recursion*: The above function exhibits a so-called "linear recursion", which ultimately leads to a stack implementation. Therefore we now change the type of recursion. To this end we introduce an auxiliary function.

   Fun $bl : nat \times nat \to nat$
   Def $bl(x, k) = blog(x) + k$.

   This function *generalizes blog* in the sense that it has a parameter k at the point, where (the expression around the recursive call of) *blog* has the constant 1. For this function we can now calculate:

$$\begin{aligned} bl(x,k) &= blog(x) + k \\ &= (\textbf{If } x = 1 \textbf{ Then } 1 \\ &\qquad \textbf{Else } blog(x \div 2) + 1 \textbf{ Fi }) + k \\ &= \textbf{If } x = 1 \textbf{ Then } 1 + k \\ &\qquad \textbf{Else } blog(x \div 2) + 1 + k \textbf{ Fi} \\ &= \textbf{If } x = 1 \textbf{ Then } 1 + k \\ &\qquad \textbf{Else } bl(x \div 2, 1 + k) \textbf{ Fi} \end{aligned}$$

   Note that this derivation depends on the *associativity* of $+$. The last equation – which still meets the termination requirement – establishes a new version of the definition of *bl*. (For reasons of readability we also apply the commutativity of $+$, even though this has no relevant effects for the algorithm.)

$$\begin{aligned} \textbf{Def } bl(x,k) = \textbf{If } & x = 1 \textbf{ Then } k + 1 \\ & \textbf{Else } bl(x \div 2, k + 1) \textbf{ Fi} \end{aligned}$$

   The connection between this new function *bl* and our original function *blog* is given by the following equation, which is a direct consequence of the initial definition of *bl*.

$$blog(x) = blog(x) + 0 = bl(x, 0).$$

   Hence we obtain the new declaration

$$\textbf{Def } blog(x) = bl(x, 0).$$

   Note that this only works, because $+$ has a neutral element 0.

5. *An iterative solution*: The new function *bl* exhibits a recursion structure which is called "tail recursion". This kind of iteration is equivalent to traditional loop constructs of imperative languages. (As a matter of fact, all modern compilers realize that this kind of recursion can be implemented without stacks.) Hence, we obtain our final version

$$\begin{aligned} \textbf{Def } blog(x) = \ & \textbf{Begin} \\ & \textbf{Var } v, k : nat; \\ & v, k := x, 0; \\ & \textbf{While } x > 1 \textbf{ Do } x, k := x \div 2, k + 1 \textbf{ Od }; \\ & \textbf{Return } k + 1 \qquad\qquad\qquad\qquad \textbf{End} \end{aligned}$$

   It now depends on the kind of programming language that we have at our disposal, whether this has to be transformed further into procedures, or whether we can stop at this level (except for some syntactic sugaring).

Discussion: This example – small as it may be – already illustrates some principal points:

- ▷ A program derivation proceeds through several stages, which we will call **milestones** in the sequel.
- ▷ The successive milestones are related to each other through algebraic calculations that establish their equivalence.
- ▷ The overall development process – the *strategy* – is guided by the programmer's ideas ("insights", "intuition").
- ▷ Experience that has been gained over the years with this approach indicates that the essential concepts for solving the given problem are best expressed on the functional level. The transition to an imperative language level is in most cases very technical and can even be left to a good compiler.

In the remainder of this paper we will elaborate the theoretical and methodological foundations of this approach in greater detail. In addition, we will illustrate its principles further by presenting some more examples.

2 An Algebraic View of Programming Concepts

We take a uniform algebraic view for our language concepts. (This even pertains to concepts from imperative languages.) In order to be more precise, we at least want to sketch here the notational and semantical framework within which we perform our deductions

2.1 Signatures

Our specifications and programs consist of collections of functions that operate on certain sets of data elements. These functions and data sets have to be appropriately named. We do this here in the following way:

- ▷ A **sort** is a name for a data set. Well-known examples are
  Sort $bool, nat, int, real, char, text$ etc.
  But we do not only have such plain sorts, but also *generic sorts*. Examples are
  Sort $seq[\alpha], set[\alpha], map[\alpha, \beta]$ etc.
  Such generic sorts can be instantiated such as
  Sort $seq[int], map[nat, real], set[seq[char]]$ etc.
  It is also possible to combine these last two concepts such as in $seq[set[\alpha]]$.
- ▷ A **function symbol** is an identifier together with a *functionality*. A functionality is a type expression, built up from sorts using direct product and function space. Examples are
  Fun $+$: $nat \times nat \to nat$
  Fun *top* : $stack[\alpha] \to \alpha$
  Fun *topop* : $stack[\alpha] \to \alpha \times stack[\alpha]$
  Fun *filter* : $(\alpha \to bool) \to seq[\alpha] \to seq[\alpha]$
  The last one of these examples shows that we also admit *higher-order functions*, that is, functions that have other functions as arguments and/or results.
  Notational conventions: We are very liberal with respect to overloading of symbols, use of infix notations, etc.

▷ A **signature** consists of a set S of sorts and a set F of function symbols, the functionalities of which are expressed in terms of the sorts from S.

Given a signature , we immediately obtain the notions of *terms* and *formulas.*

▷ A **term** is a well-formed expression built up from the symbols of a given signature, possibly including free variables. Examples:
$(x + y) * 2$, $push(s, top(s) \div 3)$, ...
The *sort of a term* is the result sort of its outermost function symbol.

▷ A **formula** is a term of sort *bool.* Atomic formulas are applications of predicates such as
$x \geq y$, $push(s, x + 1) = pop(r)$, ...
More complex formulas are then constructed with the help of boolean operations. Examples:
$x \geq y \wedge y \geq z$, $x = 0 \Rightarrow x * y = 0$, ...
Finally, we can use quantifiers to bind variables. Examples (where $\hat{}$ stands for concatenation):
$(\forall x : seq)\ s \neq empty \Rightarrow (\exists r, t : seq,\ x : data)\ s = r \hat{} x \hat{} t$
$(\forall s : s \neq empty)\ (\exists r, t : seq,\ x : data)\ s = r \hat{} x \hat{} t$
Note that these two formulas are just notational variants of each other. Note also that we automatically deduce the sorts of variables from the context whenever possible.

2.2 Specifications

In order to associate meaning to the symbols from a signature we provide specifications, which usually are in the form of predicate-logic formulas. But for methodological as well as for semantical reasons we distinguish several kinds of formulas.

▷ **Axioms** provide the basic constraints that determine the meaning of the sorts and function symbols. Examples are
Axm $(\forall x : nat)\quad x + 0 = x$
Axm $(\forall x, y : nat)\ x + suc(y) = suc(x + y)$

▷ **Theorems** are derivable from the axioms in the specification. For example, the above axioms entail – together with some further knowledge about 0 and *suc* – the following theorems:
Thm $(\forall x, y : nat)\quad x + y = y + x$
Thm $(\forall x, y, z : nat)\ (x + y) + z = x + (y + z)$

▷ **Specifications** of functions are a notational variant of certain axiomatizations that we encounter frequently. Example:
Spc $blog(x) = n$ **Pre** $x > 0$ **Post** $2^{n-1} \leq x < 2^n$
The relationship between these kinds of specifications and standard axioms will be discussed in Section 4.6.

▷ **Definitions** of functions determine a least fixed-point semantics. Example:
Def $blog(x) =$ **If** $x = 1$ **Then** 1
Else $blog(x \div 2) + 1$ **Fi**
The impacts of such a fixed-point semantics will be discussed in Section 2.4.

- ▷ **Subsort relations** express the fact that the data elements of one sort are also members of another sort. Examples:
  Axm $nat \subseteq int$
  Thm $seq[nat] \subseteq seq[int]$
- ▷ **Generation constraints** express the fact that all data elements of a sort can be constructed with a certain set of operations. Examples:
  Axm *bool* **Generated by** *true false*
  Axm *nat* **Generated by** 0 *suc*
  Axm *seq* **Generated by** *empty append*
  These kinds of formulas lead to certain induction principles that will be discussed further in Section 4.6
- ▷ **Observability** is an important concept in connection with implementation. Two objects of a given sort s are considered equivalent, if they cannot be distinguished by "observing" them only "from the outside". Example:
  Axm $seq[\alpha]$ **Observable by** α *bool nat*
  This means that the elements of sort $seq[\alpha]$ may only be distinguished by applying operations to them that lead into the sorts α, *nat*, or *bool*. If we now write equality formulas for elements of sort $seq[\alpha]$, they actually mean this observable equivalence.

Notational conventions: Again, we are very liberal in our notations: As long as the meaning is clear from the context, we will omit explicit quantifications. Note also that we allow all kinds of predicate formulas, not only equations.

2.3 Modularization and Parameterization

So far we have only considered means for *programming in the small.* But we also need ways to structure programs into larger units. In this paper we do not want to adhere to a specific language[2]. Therefore we employ a slightly more abstract notation by using "operations" such as **Signature**(...) that allow us to talk about the corresponding language features without going into syntactic details.

- ▷ **Classes** are our highest-level structuring concept.[3] They correspond to what is often called *specification, structure, encapsulation, module*, etc. These classes consist of signatures and specifications (axioms and theorems) and possibly some additional information. Example:
  Class Set = **Signature**(Set) ∪ **Specification**(Set) ∪ **Import**(Set)
  This states that the class *Set* consists of three sets of items. (For more details see section 2.5.)
- ▷ **Imports** make certain classes available in the definition of other classes. Example:
  Import(Set) = $Bool \cup Nat$
  This means that all sorts, operations, and specifications from the two classes *Bool* and *Nat* can be used in the definition of the class *Set*.

[2] ...even though we are influenced by the language SPECTRUM [10].

[3] There are certain relationships to the class concept of object-oriented programming; but there are also differences. We have chosen the name, because on the semantical level a "class" represents a *class of algebras* (which some people prefer to view as a *category* of algebras).

▷ **Exports** state, which parts of classes are made available to the environment. This way, certain sorts and operations of a class can be *hidden*, which is useful e.g. in connection with internal auxiliary functions. Example:

Export(*Set*) = {*set*, ∈}

This means that only the sort *set* and the element test ∈ can be imported by other classes.

▷ **Renaming** can be used to avoid *name clashes* between identifiers in different classes. Example:

Set **Renamed** [∅ **To** *empty*, ∈ **To** *has*]

The result of this operation is the class *Set*, but with the operations ∅ and ∈ renamed to *empty* and *has*, respectively.

▷ **Parameters** of classes are other classes. That is, we designate certain subclasses as parameters, very much like in the case of imports. Example:

$$\begin{array}{lll} \textbf{Parameter}(\mathit{OrderedSet}) = \textbf{Sort} & \alpha \\ & \textbf{Fun} & . \preceq . : \alpha \times \alpha \to \mathit{bool} \\ & \textbf{Axm} & (\forall x : \alpha)\ x \preceq x \\ & & \vdots \end{array}$$

But we could also write

Parameter(*OrderedSet*) = *LinOrd*

provided that there exists a class *LinOrd* defined as

$$\begin{array}{lll} \textbf{Class}\ \mathit{LinOrd} = \textbf{Sort} & \alpha \\ & \textbf{Fun} & . \preceq . : \alpha \times \alpha \to \mathit{bool} \\ & \textbf{Axm} & (\forall x : \alpha)\ x \preceq x \\ & & \vdots \end{array}$$

Given a parameterized class like *OrderedSet* above, we can create *instances*:

OrderedSet **Instantiated** [α **To** *nat*, $\preceq$ **To** $\leq$]

Alternatively, we could also write

OrderedSet **Instantiated By** *Nat*

provided that we have established the property

Nat **Isa** (*LinOrd* **Renamed** [α **To** *nat*, $\preceq$ **To** $\leq$])

The advantage of this latter variant is that we can establish this property once and for all for the specification of *Nat* and then reuse it wherever needed.

This short overview of possible constructs shall suffice for our purposes. A thorough elaboration can be found in the paper of Wirsing [37] and in the textbooks of Ehrig and Mahr [11]. In these references many subtleties are elaborated that we had to skip here.

2.4 Semantics

The aforementioned paper of Wirsing [37] also provides an excellent discussion of possible semantics for algebraic specifications. Therefore we content ourselves with a brief sketch here. As in the previous section, we have to skip many of the subtleties, challenging as they may be.

The fundamental notion in this context is that of an algebra. An **algebra** is a family of carrier sets together with a family of functions on these carrier sets. Such an algebra

$\mathcal{A}$ is called a *Sig*-**algebra** for a given signature *Sig*, if every sort of *Sig* is interpreted by a carrier set of $\mathcal{A}$ and every function symbol of *Sig* is interpreted by a function of $\mathcal{A}$.

A *Sig*-algebra $\mathcal{A}$ is called a **member** (or a **model**) of a class C, if **Signature**(C) = *Sig* and if all properties in **Specification**(C) hold in $\mathcal{A}$. We denote the category of all models of C by $\mathcal{MOD}(C)$. This concept is termed *loose semantics* (sometimes even *ultra-loose* or *hyper-loose* in the literature; see e.g. [9] or [26]).

In the literature there are also approaches where one specific algebra is designated as the semantics of a given specification, usually the so-called *initial* algebra. But taking the whole class of *all* models has decisive methodological advantages, in particular in connection with a stepwise development process. It is now possible to start with a very abstract specification, where no premature commitments have been made. Then we can gradually add design decisions, which are reflected in additional axioms and thus constrain the class of admissible models. In the extreme case we end up with a completely unambiguous specification, which has only one model left. Usually this final specification will be in the form of executable algorithms.

There is one additional feature that we have to mention here. Since we also work with recursive function declarations, we have to provide a suitable semantics for them as well. We do this in the traditional form by means of **least fixpoints**. We refer to the standard literature, e.g. Manna [17], Scott [33], Gordon [13], or Schmidt [32]. Section 4.6 contains additional information.

2.5 An Example: Bags

Since we will need it in some of our examples, we use the structure of "bags" to illustrate our algebraic specification concepts. *Bags*, sometimes also called *multisets*, are essentially like sets, the only difference being that multiple occurrences of elements are permitted. For example, the collection of numbers $\{1, 7, 5, 2, 5\}$ is a bag, but not a set.

We first give the signature of the class *Bag*. Note that we allow great notational freedom here, such as overloading of function symbols or infix and distfix notations.

Signature(Bag) =
Parameter (**Sort** α)
Sort $bag[\alpha]$
Let $bag = bag[\alpha]$ **In**

Fun $\emptyset$	$: bag$	– – empty bag
Fun $\{.\}$	$: \alpha \to bag$	– – singleton bag
Fun $. \uplus .$	$: bag \times bag \to bag$	– – union
Fun $. \oplus .$	$: bag \times \alpha \to bag$	– – addition of element
Fun $. \oplus .$	$: \alpha \times bag \to bag$	– – addition of element
Fun $. \ominus .$	$: bag \times \alpha \to bag$	– – deletion of element
Fun $. \in .$	$: \alpha \times bag \to bool$	– – element test
Fun $.in.$	$: \alpha \times bag \to nat$	– – number of occurrences
Fun $card$	$: bag \to nat$	– – cardinality

Obviously, this specification is based on Boolean values and natural numbers:

Import$(Bag) = Bool \cup Nat$

The specification of bags requires at least the following kinds of axioms:

Specification(Bag) =
Axm *bag* **Generated by** $\emptyset$ $\{.\}$ $.\uplus.$
Axm *bag* **Generated by** $\emptyset$ $.\oplus.$
Axm *bag* **Observable by** *nat*

Axm $(\forall B : bag,\ x : \alpha)$
$B \oplus x = B \uplus \{x\}$
$x \oplus B = \{x\} \uplus B$
Axm $(\forall A, B : bag,\ x, y : \alpha)$
$x\ in\ \emptyset = 0$
$x\ in\ \{y\} =$ **If** $x = y$ **Then** 1 **Else** 0 **Fi**
$x\ in\ A \uplus B = (x\ in\ A) + (x\ in\ B)$
$x \in B = ((x\ in\ B) > 0)$

Axm $(\forall B : bag,\ x : \alpha)$
$card(\emptyset) = 0$
$card(B \oplus x) = card(B) + 1$
Axm $(\forall B : bag,\ x, y : \alpha)$
$\emptyset \ominus x = \emptyset$
$(B \oplus x) \ominus x = B$
$(B \oplus x) \ominus y = (B \ominus y) \oplus x$ **If** $x \neq y$

Note that – in analogy to the language SPECTRUM – we presuppose the existence of a built-in equality. Therefore we only need the sort α as a parameter.

2.6 Using Higher-order Functions

There is a whole subculture emphasizing the use of higher-order functions for achieving very concise program derivations. Today, main proponents of this methodology are R. Bird and L. Meertens (and we refer the reader to their corresponding contributions in this volume, as well as to [4, 5, 6]). We will briefly sketch, how this paradigm can be integrated in our framework. First of all, we enrich *Bag* into an extended class:

Class *ExtendedBag* **Enriches** *Bag*

Then we add the desired functions. From the abundant wealth of operations considered by Bird and Meertens we only select the three most fundamental ones for illustration purposes;

Signature($ExtendedBag$) =
Fun $.\lhd.$: $(\alpha \to bool) \times bag[\alpha] \to bag[\alpha]$ – – filter
Fun $.\star.$: $(\alpha \to \beta) \times bag[\alpha] \to bag[\beta]$ – – map
Fun $./.$: $(\alpha \times \alpha \to \alpha) \times bag[\alpha] \to bag[\alpha]$ – – reduce

For these functions we have among others the following properties:

$$
\begin{array}{l}
\textbf{Specification}(\mathit{ExtendedBag}) = \\
\quad \textbf{Axm} \quad (\forall A, B : \mathit{bag}[\alpha],\ x : \alpha,\ p : \alpha \to \mathit{bool}) \\
\qquad\qquad p \lhd \emptyset \qquad\quad = \emptyset \\
\qquad\qquad p \lhd \{x\} \qquad = \textbf{If } p(x) \textbf{ Then } \{x\} \textbf{ Else } \emptyset \textbf{ Fi} \\
\qquad\qquad p \lhd (A \uplus B) = (p \lhd A) \uplus (p \lhd B) \\
\\
\quad \textbf{Axm} \quad (\forall A, B : \mathit{bag}[\alpha],\ x : \alpha,\ f : \alpha \to \alpha) \\
\qquad\qquad f \star \emptyset \qquad\quad = \emptyset \\
\qquad\qquad f \star \{x\} \qquad = \{f(x)\} \\
\qquad\qquad f \star (A \uplus B) = (f \star A) \uplus (f \star B) \\
\\
\quad \textbf{Axm} \quad (\forall A, B : \mathit{bag}[\alpha],\ x : \alpha,\ . \odot . : \alpha \times \alpha \to \alpha) \\
\qquad\qquad \odot/\emptyset \qquad\quad = \textbf{Neutral}(\odot) \\
\qquad\qquad \odot/\{x\} \qquad = \{x\} \\
\qquad\qquad \odot/(A \uplus B) = (\odot/A) \odot (\odot/B) \\
\qquad\qquad \textbf{Pre}(\odot/B) = \textbf{Commutative}(\odot)
\end{array}
$$

Here, **Neutral**($\odot$) denotes the neutral element of the operation $\odot$, the existence of which we have to presuppose. Similarly, **Commutative**($\odot$) expresses the requirement that $\odot$ be a commutative operation. Without these constraints the definition of the higher-level functions would be in contradiction to the corresponding properties of $\uplus$. (The deeper reason behind this problem is that / essentially is a homomorphism from the *Bag*-algebras to the α-algebras, and that this homomorphism maps $\uplus$ to $\odot$; hence both operations must posses the same algebraic properties.)
On the basis of these axioms we can now derive further theorems that ultimately enable the compact forms of reasoning advocated in the aforementioned papers. (Note: By $\circ$ we denote function composition.)

$$
\begin{array}{lll}
\textbf{Specification}(\mathit{ExtendedBag}) \vdash & \textbf{Thm} & (p\lhd) \circ (q\lhd) = (q\lhd) \circ (p\lhd) \\
& \textbf{Thm} & f \star (g \star B) \ = (f \circ g) \star B \\
& & \vdots
\end{array}
$$

Since there are other contributions in this volume that deal extensively with these techniques, we will not pursue the issue further here.

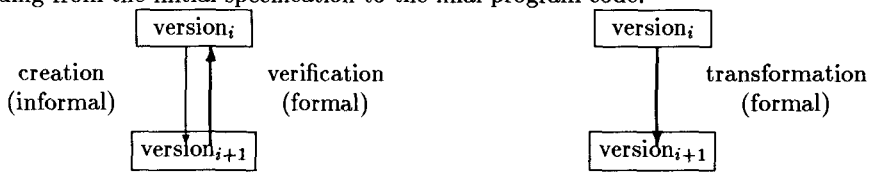
3 Programming By Transformations

It is unquestioned that programming is a stepwise process. No reasonably large or complex piece of software is produced in one go. But the decisive question is: *How* are the individiual steps performed?

- The *traditional approach* – which still predominates in industrial software engineering – is a more or less disciplined process of stepwise refinement. Starting from an abstract requirements document one proceeds through more and more concrete versions of the program, mostly written in some kind of pseudo code, until one finally produces the actual program code. The main weakness of this approach is that all these activities are *highly informal* such that quality assurance must solely rely on managerial and organizational means like "code walk-through", "structured testing", and the like.

▷ The *verification approach* strives for quality assurance through rigorous proof procedures: Any two successive stages in the development process have to be proved equivalent. The prerequisite for this way of proceeding is, of course, a fully formalized representation of all intermediate stages of the program, since formalization is at the heart of any sound mathematical proof system. (Depending on the area of application the most common formalisms are variations of Hoare logic or temporal logic.)
By the way, early hopes that it would suffice to verify only the final program against the initial specification have been shattered: The great gap between these two versions generally made the proofs too complex. Therefore, the evolutionary proceeding through a series of intermediate stages had to be adopted in the verification approach as well in order to keep the individual proofs manageable.

▷ The *transformational approach* combines the evolutionary concept of a stepwise development process with the mathematical rigour of the verification approach. But it takes a slightly different view: In a verification-oriented development step one first constructs the next version of the program "on speculation", and then performs an "a posteriori" verification of its equivalence to the previous version. By contrast, the transformation idea is to *derive* the next version from the actual one according to pre-verified, formal rules. In other words, one performs a *constructive* derivation, leading from the initial specification to the final program code.



verification-oriented development step

transformational development step

▷ We pursue here a mixed concept, which we baptize – for lack of better terminology – simply **deductive approach**. This is motivated by the following experience: In general the evolutionary concept of transformational programming is most valuable; but sometimes the compulsion of a strictly constructive, forward-oriented process turns into a straitjacket. On those occasions we want to freely switch to a verification-oriented way of proceeding.
In other words, we still adopt a mainly transformation-based style of programming. But whenever necessary, we employ verification methods as well. Indeed, as will be seen in the remainder of the paper, the division of the two concepts is fuzzy anyway.

4 A Calculus For Deductive Programming

The technical realization of a transformational programming style can be effected in various forms. We choose here the framework of logic and algebra. More precisely, our concepts are oriented at the so-called *calculus of natural deduction*, which goes back to Gentzen [12]. The advantages of this choice are evident: The foundations have been thoroughly elaborated and are accepted for many years by now. The concept has proved

worthwhile in many areas of mathematics. The notations and techniques are clear and easy to comprehend, in spite of their absolute formal rigour. For these reasons, this kind of logic has been used by many computer scientists (such as Manna [17], Gries [14], Kahn [16]). And it will be seen below that the natural-deduction style is very well suited for our approach as well.

It is, however, not our intention to explain the underlying principles of natural deduction here. We will just employ them in our work.

4.1 Rules

The central concept of our method are **deduction rules**. Basically, such a rule consists of several premises and a conclusion. But we also allow the combination of several rules with identical premises into a single rule, which then has several conclusions.

$$\frac{\begin{array}{l}\vdash premise_1\\ \quad\vdots\\ \vdash premise_n\end{array}}{\vdash conclusion} \qquad \frac{\begin{array}{l}\vdash premise_1\\ \quad\vdots\\ \vdash premise_n\end{array}}{\begin{array}{l}\vdash conclusion_1\\ \quad\vdots\\ \vdash conclusion_k\end{array}}$$

Such a rule expresses the following fact: *Whenever the premises are valid, the conclusion is valid as well.* Hence, a rule is to be viewed as a *judgement*, that is, as a statement of fact; it must not be confused with an implication formula, which can be either true or false. A most trivial example of a rule is given by

$$\frac{\vdash \neg B}{\vdash \textbf{If } B \textbf{ Then } X \textbf{ Else } Y \textbf{ Fi} \;=\; Y}$$

As premises and conclusions we usually have formulas. However, for reasons of uniformity we also consider the other constructs of our language as a kind of special predicates here:

- ▷ **Fun** $f : s \to r$ is viewed as a predicate stating that the functionality of f is $s \to r$.
- ▷ **Axm** F is viewed as a predicate stating that the formula F is an axiom (in the given context). Analogously for **Thm** .
- ▷ **Signature**$(S) = Sig$ is viewed as a predicate stating that the signature of S is (the text) Sig. Analogously for **Import**(S), **Specification**(S), etc.

4.2 Contexts ("Local Theories")

Any program derivation takes place within a certain *context*. Graph algorithms are developed within the context of graph theory, compilers are developed within the context of language theory, matrix algorithms are developed within the context of linear algebra and numerical analysis, and so on. Each of these contexts constitutes a **local theory** that comprises the knowledge which is available from the relevant context. We respect such local theories by extending our notion of "rule" to the following form:

$$\frac{\begin{array}{ll}\Gamma_1 & \vdash premise_1 \\ & \vdots \\ \Gamma_n & \vdash premise_n\end{array}}{\Gamma_{n+1} \vdash conclusion}$$

The meaning of such a rule is the following judgement: If the premises hold in their respective contexts Γ_i, then the conclusion holds in the context Γ_{i+1}. (Mostly, all Γ_i will be identical.) Example:

$$\frac{Nat, Set \vdash card(s \cup \{x\}) > 0}{Nat, Set \vdash \textbf{If } card(s \cup \{x\}) = 0 \textbf{ Then } A \textbf{ Else } B \textbf{ Fi } = B}$$

This means: If we can show in the context of the specifications *Nat* and *Set* that the cardinality of the set $s \cup \{x\}$ is greater 0, then the same context ensures that the given conditional reduces to its **Else**-branch.

- ▷ Our contexts will usually be collections of specifications such as *Nat*, *Seq*, *Set*, etc., but we also allow single formulas such as $x \geq 0$, $s \neq empty$, etc.
- ▷ We may write **Post**$(f) \vdash \ldots$ or **Spc**$(f) \vdash \ldots$ to express the fact that some conclusion can be drawn from the specification of f or from the postcondition of f.

Conventions: To increase readability we employ a number of notational conventions.

- When the context or parts of the context in a rule are identical or evident, we omit them.
- When a context remains invariant over a longer development period, then we will only state it at the very beginning.
- We sometimes state a requirement like "... where x is a new variable". This means that x does not occur freely in any formula of the given context.

4.3 Trivial Rules For Specifications

There are some fundamental rules that are so obvious that we will usually not even mention them explicitly in our derivations. But since they form the basis for our operations on specifications and contexts, we will at least exemplify them by a few selected rules here.

1. It is self-evident that any property, which is explicitly written down in a specification, also holds in the context of this specification. But technically we need to express this fact in a rule: It transfers written specification text into the basic formulas of a deduction.

Rule 1 (elementary deduction)
$\dfrac{\vdash P \in Spec}{Spec \vdash P}$

The notation $P \in Spec$ means that P is an actual part of the text *Spec*.

2. If a property P is deducible from a specification *Spec* we can add it to *Spec* as a theorem:

Rule 2 (theorem introduction)
$\Gamma, Spec \vdash P$
$\Gamma \qquad \vdash Spec \equiv Spec \cup \{\mathbf{Thm}\ P\}$

Again, the notation $Spec \cup \{\mathbf{Thm}\ P\}$ means that the law **Thm** P is added to the specification text *Spec*. And '$\equiv$' means semantic equivalence.

To emphasize this again: Such trivialities will not be mentioned in the sequel, but from a technical point of view they have to be part of our calculus.

4.4 Design Decisions And Modalities

In the course of a program development we usually have to make design decisions. This means that we *strengthen the specification* in the sense that the class of possible models is reduced.[4] Obviously, such strengthening properties are not *derivable* in the calculus. All we can expect is that they are *compatible* with the remainder of the specification.

From a *practical* point of view this simply means that we need *operations* that perform the pertinent modifications of the specification under consideration — provided that the necessary premises are fulfilled. However, this would mean that the conclusions of our rules are no longer predicates but rather actions. And this would be in contrast with a clean notion of "calculus".

Therefore we introduce a notation that from a *pragmatic* point of view may be interpreted as "permission" to apply such an operation. But fortunately there is also a clear *conceptual* view of this notation, based on principles from modal logic. In the sequel we will briefly elaborate this viewpoint.

We are confronted with the following situation: At any given stage of our development process we are dealing with a certain *theory* $\mathcal{T}_i$ (which specifies a class of models $\mathcal{MOD}(\mathcal{T}_i)$, where – intuitively speaking – the various models represent the possible "implementations" of our current specification). A true design decision usually excludes some of these possible implementations and only leaves the remaining ones as candidates. That is, we pass on to a *new theory* $\mathcal{T}_{i+1}$, which has the properties

$$\emptyset \neq \mathcal{MOD}(\mathcal{T}_{i+1}) \subseteq \mathcal{MOD}(\mathcal{T}_i)$$
that is, $\mathcal{T}_{i+1} \vdash \mathcal{T}_i$

In other words, the development process $\mathcal{T}_0 \rightarrow \cdots \rightarrow \mathcal{T}_i \rightarrow \mathcal{T}_{i+1} \rightarrow \cdots$ proceeds in the converse direction of the entailment relation '$\vdash$'. But this means that the main transition steps of our derivation process, viz. the transitions $\mathcal{T} \rightarrow \mathcal{T}'$, cannot be expressed within the calculus. All we can do is the following:

- ▷ First, apply the changes leading from $\mathcal{T}$ to $\mathcal{T}'$ on speculation.
- ▷ Then verify the entailment $\mathcal{T}' \vdash \mathcal{T}$.

[4] There are also design decisions which are true alterations of the specification (reflecting e.g. a change of the requirements); but these are not considered here.

Since this is technically unpleasant, we may use the following trick[5]: We *derive within the old theory* $\mathcal{T}$ "admissible" modifications, that is, modifications that guarantee $\mathcal{T}' \vdash \mathcal{T}$. However, since these modifications are in general true strengthenings of the theory $\mathcal{T}$, they cannot be directly "derived". All we can say is that they are "compatible" with $\mathcal{T}$.

This could be expressed in a modal logic of "necessity and possibility" [23]. In such a logic we could write, for instance, a formula like

$$\textbf{possibly}\ (x > 0 \Rightarrow f(x) = f(x \div 2) + 1).$$

for which we use the short-hand notation (that is usually employed in the area of modal or temporal logic)

$$\Diamond\,(x > 0 \Rightarrow f(x) = f(x \div 2) + 1).$$

This means that the decision to define for $x > 0$ the function $f(x)$ as $f(x \div 2) + 1$ is compatible with the rest of the specification, even though it does not necessarily follow from it.

We need this notation in particular in connection with the introduction of new axioms or definitions, that is, whenever we have to make a true *design decision*. Then we write e.g.

$$\frac{\cdots}{\mathcal{T} \vdash \Diamond\ \textbf{Axm}\ F}$$

This means that the formula F is compatible with the rest of the specification and therefore *can be added* as an axiom. (By contrast, a theorem is actually deducible from the specification.) The same principle applies to other constructs that reflect design decisions, such as

$$\vdash \quad \Diamond\ \textbf{Def}\ f(x) = \cdots$$
$$\vdash \quad \Diamond\ \textbf{Spc}\ f(x) = \cdots$$

Pragmatically this simply means that the old definition or specification, respectively, of f shall be replaced by the new one.

What do we gain from this concept *theoretically*? Suppose we have a current theory $\mathcal{T}$ and we are able to derive

$$\mathcal{T} \vdash \Diamond F$$

This means that there is at least one model in $\mathcal{MOD}(\mathcal{T})$, in which F holds. Hence, we can pass to

$$\mathcal{T}' \stackrel{\text{def}}{=} \mathcal{T} \cup \{F\}$$

and we know that $\mathcal{MOD}(\mathcal{T}') \neq \emptyset$ and $\mathcal{T}' \vdash \mathcal{T}$. In other words, the necessary meta-level step from one theory to the next has been completely prepared *within* the calculus.

[5] This is an experimental idea that we pursue for the first time in this paper. Its practical feasibility still has to be investigated.

Note, however, that this concept is in a certain sense "non-monotonic". To see this, suppose that we had also established the additional property $T \vdash \Diamond E$. Then it need not be the case that T' still entails $\Diamond E$. Hence, the commitment to one possible design decision requires a reassessment of all other possibilities.

To round off the discussion we should at least sketch the essential semantical and logical properties of the $\Diamond$-operator.

The *semantics* of '$\Diamond$' is explained as follows: Let T be a specification and $\mathcal{MOD}(T)$ its class of models. Then we define a "reachability relation" ϱ between algebras by

$$\mathcal{A}\varrho\mathcal{B} \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad \mathcal{A} \in \mathcal{MOD}(T) \wedge \mathcal{B} \in \mathcal{MOD}(T)$$

This trivial relation is reflexive, symmetric, and transitive. (Therefore we obtain – in the terminology of modal logic – a so-called system S5.) Now we define the *validity* of a formula $\Diamond F$ in an algebra $\mathcal{A}$ by

$$(\mathcal{A} \models \Diamond F) \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad (\exists \mathcal{B}, \mathcal{A}\varrho\mathcal{B} : \ \mathcal{B} \models F)$$

This operator obeys among others the following axioms (as described e.g. in [15]).

$$\begin{array}{ll} F & \Rightarrow \Diamond F \\ \Diamond\Diamond F & \Leftrightarrow \Diamond F \\ \Diamond(E \vee F) & \Leftrightarrow (\Diamond E) \vee (\Diamond F) \\ \Diamond(E \wedge F) & \Rightarrow (\Diamond E) \wedge (\Diamond F) \end{array}$$

Note, however, that the converse direction of the last implication does in general not hold.

4.5 Standard Rules For Predicate Logic

The Gentzen calculus of natural deduction originally has been developed for propositional logic and predicate logic. It is documented in textbooks such as those of Manna [17] and Gries [14]. The calculus is organized in such a way that for each logical symbol there is one rule for its introduction and one rule for its elimination. Traditionally, this entails the following rules:

- The rules $\neg$-*introduction* and $\neg$-*elimination*.
- The rules $\vee$-*introduction* and $\vee$-*elimination*.
- The rules $\wedge$-*introduction* and $\wedge$-*elimination*.
- The rules $\Rightarrow$-*introduction* and $\Rightarrow$-*elimination*.
- The rules $\Leftrightarrow$-*introduction* and $\Leftrightarrow$-*elimination*.
- The rules $\forall$-*introduction* and $\forall$-*elimination*.
- The rules $\exists$-*introduction* and $\exists$-*elimination*.

We employ all rules from this calculus, but we will usually not mention them explicitly, because they are again trivial and well-known. But there are a few exceptions, which are more relevant from a methodological point of view:

1. We often argue by case distinctions. This is formalized by the rule

Rule 3 ($\vee$-elimination, "proof by case distinction")
$A \vdash P$ $B \vdash P$ $\vdash A \vee B$ $\vdash P$

2. A classical proof technique proceeds by contradiction:

Rule 4 ($\neg$-introduction, "proof by contradiction")
$A \vdash \mathit{false}$ $\vdash \neg A$

3. The connection between rules and implication formulas is established by the following pair of rules.

Rule 5 ($\Rightarrow$-elimination, "modus ponens")
$\vdash A \Rightarrow B$ $\vdash A$ $\vdash B$

Rule 6 ($\Rightarrow$-introduction, "deduction theorem")
$A \vdash B$ $\vdash A \Rightarrow B$

4.6 Rules For Specific Language Constructs

The constructs of a programming language can be viewed as operations in an algebra. Therefore they possess properties like operations in any other algebra. In the sequel we present some illustrative examples for such properties.

Rules for conditional expressions. A conditional expression can be introduced according to the following rule, which is closely related to the rule 3, because **Defined**(C) — which means that the evaluation of the expression C does neither abort nor diverge — is the same as $C \vee \neg C$.

Rule 7 (If-introduction)
$C \vdash x = E_1$ $\neg C \vdash x = E_2$ $\vdash$ **Defined**(C)
$\vdash x =$ **If** C **Then** E_1 **Else** E_2 **Fi**

The converse direction allows us to eliminate a conditional expression.

Rule 8 (If-elimination)
$\vdash x =$ **If** C **Then** E_1 **Else** E_2 **Fi** $\vdash C$
$\vdash x = E_1$

Note that this rule has an important effect: It allows us to deduce facts about the **Then**-branch in an extended context, reflecting the knowledge that the condition is true. The analogous rule is available for $\neg C$. And it is evident, how these rules can be generalized to nested conditionals and to conditionals with more than two cases.

Rules for Let-clauses. Often we want to introduce an auxiliary identifier that abbreviates the value of an expression. The following two rules formalize this activity. Note that this introduces an abbreviating identifier into the *development*, and not yet into the program texts.

Rule 9 (Let-introduction)
$\vdash (\exists x)\ x = A$
$\vdash$ **Let** $x = A$
where x is a new identifier

This rule is closely related to the classical Gentzen rules for the existential quantifier. It says: When the expression A has a well-defined value (that is, its evaluation does not abort or diverge), then we can name this value by x (as long as no name clashes occur).

Rule 10 (Let-elimination)
$\vdash$ **Let** $x = A$
$\vdash x = A$

This rule merely performs the necessary "theory propagation" that allows us to make use of the declaration in the remainder of the development.

We can bring such **Let**-clauses from the development level into the program level by the following rule:

Rule 11 (Let-clause)
$\vdash$ **Let** $x = A$ $\vdash E = F$ $\vdash E =$ **Let** $x = A$ **In** F
x must not occur freely in E

Rules for function specifications. The effect of our specification construct, which is based on pre- and postconditions, can also be obtained with standard axiomatic notations. But then we have to talk about the complete set of all axioms in many derivations, which is technically unpleasant. Therefore we add this concept to our calculus in order to be able to localize full axiomatizations in one syntactic construct.

There are several ways in which the roles of pre- and postconditions can be defined. We choose here the view of "total correctness"; that is, the precondition holds if and only if the function terminates with a well-defined value. This is made precise in the following rule.

Notational convention: We use the notations $P[x]$ and $R[x, y]$ to denote the following conventions: The expression $R[x, y]$ may contain the identifiers x and y, whereas $P[x]$ may only contain x, but not y.

Rule 12 (deductions from function specifications)
$\Gamma \vdash$ **Spc** $f(x) = y$ **Pre** $P[x]$ **Post** $R[x, y]$
$\Gamma,\ P[x] \vdash (\exists y)\ y = f(x)$ $\Gamma,\ (\exists y)\ y = f(x) \vdash P[x]$ $\Gamma,\ y = f(x) \vdash R[x, y]$
x and y must not occur freely in the given context Γ

The first two conclusions state that the precondition $P[x]$ holds if and only if the application $f(x)$ is defined. The last conclusion states that the result meets the postcondition.

The converse direction of this rule means to extract a pre/postcondition-based specification of f from the given algebraic axiomatization. To do this we have to express the fact that any property $Q[f]$ that follows from the axiomatization Γ also follows from the postcondition R. This necessitates a premise of the kind

$$\ll \text{from} \quad \Gamma \vdash Q[x, f(x)] \quad \text{infer} \quad \Gamma \vdash R[x, y] \Rightarrow Q[x, y] \gg,$$

which, unfortunately, would add a third level to our two-level calculus. Since it is much easier to express this collection of all f-related axioms algorithmically, we refrain from this complication of the calculus.

From the above definitions we obtain a rule that reflects the activity of making design decisions in the course of a development: We can always strengthen a given postcondition,

thus restricting the degrees of freedom that are permitted by the specification. In doing so we have, of course, to ensure that the specification does not become inconsistent.

Rule 13 (strengthening the postcondition)

$$\frac{\begin{array}{l}\Gamma \vdash \textbf{Spec } f(x) = y \textbf{ Pre } P[x] \textbf{ Post } R[x,y] \\ \Gamma,\ P[x] \vdash (\exists y)\ Q[x,y] \\ \Gamma,\ P[x] \vdash Q[x,y] \Rightarrow R[x,y]\end{array}}{\Gamma \vdash \Diamond\ \textbf{Spc } f(x) = y\ \textbf{ Pre } P[x]\ \textbf{ Post } Q[x,y]}$$

x and y must not occur freely in the given context Γ

An analogous rule can be given for weakening the precondition.

Rules for fixpoint equations. Of particular interest are possibilities for passing from function specifications to function definitions. We split this process into (at least) two kinds of rules: The first kind introduces fixpoint equations from specifications, and the second one passes from fixpoint equations to least-fixpoint definitions.

For the first class of rules we give four variants. (Note that they are all design decisions, which in general requires the possibility operator '$\Diamond$' from Section 4.4).

The first rule reflects the classical verification-oriented way of proceeding: We "guess" a solution, that is, a possible function body $E[x, f]$ and prove that this design meets the specification.

Rule 14 (introduction of fixpoint equations)

$$\frac{\begin{array}{l}\Gamma \vdash \textbf{Spc } f(x) = y\ \textbf{ Pre } P[x]\ \textbf{ Post } R[x,y] \\ \Gamma,\ P[x] \vdash (\exists y)\ y = E[x,f] \\ \Gamma,\ P[x] \vdash y = E[x,f] \Rightarrow R[x,y]\end{array}}{\Gamma \vdash \Diamond\ \textbf{Axm } P[x] \Rightarrow f(x) = E[x,f]}$$

x and y must not occur freely in the given context Γ

The justification of this rule is simple: It is a corollary to rule 13 above.

The second rule is more "constructive" in the sense that it deduces the recursion structure of f from certain algebraic properties of expressions E and K — which, however, still have to be invented by the programmer.

Rule 15 (introduction of fixpoint equations)
$\Gamma \vdash \mathbf{Spc}\ f(x) = y\ \mathbf{Pre}\ P[x]\ \mathbf{Post}\ R[x,y]$ $\Gamma, P[x] \vdash R[K[x], y] \Rightarrow R[x, E[x,y]]$ $\Gamma,\ P[x],\ \neg P[K[x]] \vdash R[x, E[x, \bot]]$
$\Gamma \vdash \Diamond\ \mathbf{Axm}\ P[x] \Rightarrow f(x) = E[x, f(K[x])]$
x and y must not occur freely in the given context Γ

Let us briefly consider the justification of this rule. To this end, recall that a specification generally admits many implementations. And the above rule says that among these solutions there is at least one, which obeys the given recursive equation. Formally, this rule is a simple corollary to rule 13 above, since we can consider the additional fixpoint property as a strengthening of the postcondition:

$$\mathbf{Spc}\ f(x) = y\ \mathbf{Pre}\ P[x]\ \mathbf{Post}\ R[x,y] \wedge y = E[x,\ f(K[x])]$$

For this strengthening the first and third premise of rule 13 are trivially fulfilled. Therefore it remains to establish the second premise, that is,

$$P[x] \vdash (\exists y)\ R[x,y] \wedge y = E[x, f(K[x])].$$

We distinguish two cases:

1. $P[K[x]]$ holds. Then we can deduce:
   $\vdash R[K[x], f(K[x])]$ [by spec. of f]
   $\vdash R[x, E[x, f(K[x])]]$ [by 2nd premise of rule 15]
   Hence, $y \stackrel{\text{def}}{=} E[x, f(K[x])]$ is a "witness" for the validity of the existential formula above.
2. $\neg P[K[x]]$. Then we can deduce:
   $\vdash f(K[x]) = \bot$ [by spec. of f]
   $\vdash E[x, f(K[x])] = E[x, \bot]$
   $\vdash R[x, E[x, \bot]]$ [by 3rd premise of rule 15]
   Hence, $y \stackrel{\text{def}}{=} E[x, \bot]$ is a witness.

The possibility operator is needed, since in general the postcondition $R[x,y]$ is compatible with several different fixpoint equations. If, however, R determines the function uniquely, we obtain the equation as a theorem:

Rule 16 (introduction of fixpoint equations)

$$\frac{\begin{array}{r}\Gamma \vdash \textbf{Spc}\ f(x) = y\ \textbf{Pre}\ P[x]\ \textbf{Post}\ R[x,y] \\ \Gamma, P[x] \vdash R[K[x], y] \Rightarrow R[x, E[x,y]] \\ \Gamma,\ P[x],\ \neg P[K[x]] \vdash R[x, E[x, \bot]] \\ \Gamma,\ P[x] \vdash R[x,y] \wedge R[x,y'] \Rightarrow y = y'\end{array}}{\Gamma \vdash \textbf{Thm}\ P[x] \Rightarrow f(x) = E[x, f(K[x])]}$$

x and y must not occur freely in the given context Γ

It is sometimes convenient to work with the following variants of the above rule. These variants are simply obtained by algebraic transformations of the second premise of rule 15 above.

Rule 17 (introduction of fixpoint equations)

$$\frac{\begin{array}{r}\Gamma \vdash \textbf{Spc}\ f(x) = y\ \textbf{Pre}\ P[x]\ \textbf{Post}\ R[x,y] \\ \Gamma,\ P[x] \vdash K^{-1}[K[x]] = x \\ \Gamma,\ P[x] \vdash R[x,y] \Rightarrow R[K^{-1}[x], E[K^{-1}[x], y]] \\ \Gamma,\ P[x],\ \neg P[K[x]] \vdash R[x, E[x, \bot]]\end{array}}{\Gamma \vdash \Diamond\ \textbf{Axm}\ P[x] \Rightarrow f(x) = E[x, f(K[x])]}$$

x and y must not occur freely in the given context Γ

Rule 18 (introduction of fixpoint equations)

$$\frac{\begin{array}{r}\Gamma \vdash \textbf{Spc}\ f(x) = y\ \textbf{Pre}\ P[x]\ \textbf{Post}\ R[x,y] \\ \Gamma,\ P[x] \vdash E^{-1}[x, E[x,y]] = y \\ \Gamma, P[x] \vdash R[K[x], E^{-1}[x,y]] \Rightarrow R[x,y] \\ \Gamma,\ P[x],\ \neg P[K[x]] \vdash R[x, E[x, \bot]]\end{array}}{\Gamma \vdash \Diamond\ \textbf{Axm}\ P[x] \Rightarrow f(x) = E[x, f(K[x])]}$$

x and y must not occur freely in the given context Γ

We can also combine the above rules with the rule 7 for **If**-introduction. In the first case we obtain the combined rule

Rule 19 (introduction of conditional fixpoint equations)

$$\frac{\begin{array}{l}\Gamma \vdash \textbf{Spec } f(x) = y \textbf{ Pre } P[x] \textbf{ Post } R[x,y] \\ \Gamma,\ P[x],\ \neg C[x] \vdash R[K[x], y] \Rightarrow R[x, E[x,y]] \\ \Gamma,\ P[x],\ C[x] \vdash R[x, A[x]] \\ \Gamma,\ P[x] \vdash \textbf{Defined}(C[x])\end{array}}{\begin{array}{l}\Gamma \vdash \Diamond\ \textbf{Axm } P[x] \Rightarrow f(x) = \textbf{ If } C[x] \textbf{ Then } A[x] \\ \qquad\qquad\qquad\qquad\qquad\qquad \textbf{Else } E[x, f(K[x])]\ \textbf{Fi}\end{array}}$$

x and y must not occur freely in the given context Γ

Analogous variants can be given for the other rules. And it is also evident, how these rules can be generalized to more than two cases.

Rules for function definitions. Our equational calculus allows us to derive equations such as the theorem $f(x) = E[x, f]$ in the above rule. And very often these equations are recursive; that is, the expression E contains applications of f. However, not every such equation constitutes a feasible recursive definition of f – as can be seen from the trivial equation $f(x) = f(x)$. The problem is that an equational calculus can at best show that a function is a solution (a "fixpoint") of a given equation, but it does not suffice to show that a function is the *least* solution.

Therefore we have to employ a domain theory on the basis of so-called *complete partial orders*, as it is described e.g. by Scott [33], Gordon [13], Schmidt [32], and also Manna [17]. We cannot go into the details of such theories but will only mention that they are based on a *definedness ordering*: $f \sqsubseteq g \quad \Leftrightarrow \quad \forall x : f(x) = g(x) \ \vee \ \mathit{undefined}(f(x))$ Using this ordering we can express the fact that our recursive function definitions not only determine some fixpoint but rather designate the unique *least* fixpoint.

Rule 20 (introduction of function definitions)

$$\frac{\begin{array}{l}\Gamma \vdash f(x) = E[x, f] \\ \Gamma, g(x) = E[x, g] \vdash f \sqsubseteq g\end{array}}{\Gamma \vdash \textbf{Def } f(x) = E[x, f]}$$

g and x must not occur freely in the given context Γ

The disadvantage of this rule is that it relies on the definedness ordering $\sqsubseteq$, which is not easy to work with. One situation, where this problem can be overcome, occurs when there is a well-founded ordering on the argument sorts. In such an ordering there are no infinite strictly decreasing chains of the kind $x_0 \succ x_1 \succ x_2 \succ \ldots$. Hence, we can guarantee termination.

We illustrate this principle by a phenotypical rule, where we use the following convention: A notation like $E[x, f(K[x])]$ expresses the fact that E is an expression containing

exactly one application of the function f, the argument of which is computed by the expression $K[x]$.

Rule 21 (terminating function definitions)

$$\frac{\begin{array}{ll}\Gamma & \vdash \textbf{Wellfounded}(\prec) \\ \Gamma,\ \neg C[x] & \vdash K[x] \prec x \\ \Gamma & \vdash f(x) = \textbf{If } C[x] \textbf{ Then } A[x] \textbf{ Else } E[x, f(K[x])] \textbf{ Fi}\end{array}}{\Gamma \quad \vdash \textbf{Def } f(x) = \textbf{If } C[x] \textbf{ Then } A[x] \textbf{ Else } E[x, f(K[x])] \textbf{ Fi}}$$

x must not occur freely in the given context Γ

If there is more than one application of f, then the well-foundedness has to be required for each of them.

One standard way for providing a well-founded ordering is to invent a mapping τ from the parameter sort s into the natural numbers $\mathcal{N}$ such that $\tau(K[x]) < \tau(x)$ holds for the standard $<$-relation in $\mathcal{N}$.

Structural Induction. No reasonable development method can do without induction. As has already been pointed out in Section 2.2 our generation constraints provide a means for establishing specific induction rules. The classical example are, of course, the natural numbers.

Rule 22 (structural induction for *Nat*)

$$\frac{\begin{array}{ll} & \Gamma \vdash nat \textbf{ Generated by } 0\ suc \\ & \Gamma \vdash \mathcal{F}[0] \\ \Gamma,\ \mathcal{F}[i] & \vdash \mathcal{F}[suc(i)]\end{array}}{\Gamma \vdash (\forall n : nat)\ \mathcal{F}[n]}$$

i must not occur freely in the given context Γ

The same principle applies also to our example *Bag*. Here we obtain actually two induction rules, because we have given two generation axioms.

Rule 23 (structural induction for *Bag*)

$$\frac{\begin{array}{ll} & \Gamma \vdash bag \textbf{ Generated by } \emptyset\ \{.\}\ \uplus \\ & \Gamma \vdash \mathcal{F}[\emptyset] \\ & \Gamma \vdash \mathcal{F}[\{x\}] \\ \Gamma,\ \mathcal{F}[A],\ \mathcal{F}[B] & \vdash \mathcal{F}[A \uplus B]\end{array}}{\Gamma \vdash (\forall B : bag)\ \mathcal{F}[B]}$$

x, A, B must not occur freely in the given context Γ

Rule 24 (structural induction for *Bag*)

$$\frac{\Gamma \vdash bag \textbf{ Generated by } \emptyset \ . \oplus . \qquad \Gamma \vdash \mathcal{F}[\emptyset] \qquad \Gamma,\ \mathcal{F}[B] \vdash \mathcal{F}[B \oplus x]}{\Gamma \vdash (\forall B : bag)\ \mathcal{F}[B]}$$

x and B must not occur freely in the given context Γ

In this way our specifications yield a wealth of rules that can be utilized in program derivations, since every generation constraint automatically induces a corresponding induction rule. The textbooks by Manna and Waldinger [18] provide an extensive overview over the structural induction rules that are typically used in programming.

4.7 Another Simple Example

Let us illustrate the impacts of this calculus on our methodology for program derivations by using an example that is equally simple as the introductory example *blog* in Section 1: We want to derive a circuit for integer division, which is, of course, based on the binary representation of numbers. The following derivation yields the algorithm underlying the envisaged circuit.

Development *Division*

1. *Establishing the domain theory.* Our given problem lives in the domain of the natural numbers. We presuppose that the corresponding laws of arithmetic are contained in a suitable specification.

   $$\textbf{Context}(Division) \vdash Arithmetic \tag{1}$$

   This context, that is, the laws of arithmetic entail in particular the following property, on which we will rely frequently in the sequel[6].

   $$\textbf{Thm}\ \ (0 \leq a - i \cdot b < b) \wedge (0 \leq a - j \cdot b < b) \Rightarrow (i = j) \tag{2}$$

   Note: We will also employ other, more basic properties of arithmetic without mentioning them explicitly.
2. *Specification of the problem.* Our task can be fomulated in terms of a single function.

   $$\textbf{Fun}\ \ div : nat \times nat \rightarrow nat \times nat$$
   $$\textbf{Spc}\ \ div(a, b) = q, r \tag{3}$$
   $$\textbf{Pre}\ \ a < 2^n \wedge 0 < b \tag{4}$$
   $$\textbf{Post}\ \ a = q \cdot b + r \tag{5}$$
   $$0 \leq r < b \tag{6}$$

[6] In a real development we would not state this theorem before we actually need it. But for keeping the presentation more readable, we list it already now.

Note that the precondition $a < 2^n$ (with some fixed n) already reflects the fact that our machines operate with bounded numbers. But this feature will only be used in later stages of our development.

The postcondition can be extended by the following equations that we will need in a moment:

$$\vdash r = a - q \cdot b \quad [\text{ by } 5] \tag{7}$$

$$\vdash 0 \leq a - q \cdot b < b \; [\text{ by } 6, 7] \tag{8}$$

3. *Termination case.* When the dividend is smaller than the divisor, no calculation is necessary.

Case 1: $a < b$ (9)

$$\vdash q = 0 \; [\text{ by } 5, 9] \tag{10}$$

$$\vdash r = a \; [\text{ by } 7, 10] \tag{11}$$

4. *Recurrences.* Since we operate within a binary number system, the idea comes to mind to aim at a logarithmic algorithm by doubling the divisor in each step.

Case 2: $b \leq a$ (12)

Let $q', r' = div(a, 2 \cdot b)$ (13)

$$\vdash a = q' \cdot 2 \cdot b + r' \quad [\text{ by } 13, 5] \tag{14}$$

$$\vdash r' = a - 2 \cdot q' \cdot b \quad [\text{ by } 14] \tag{15}$$

$$\vdash r = (2 \cdot q' - q) \cdot b + r' \; [\text{ by } 15, 5] \tag{16}$$

After collecting these basic facts, we now have to make a straightforward case distinction for r', which is complete due to the postcondition of *div*:

$$\vdash 0 \leq r' < b \vee b \leq r' < 2 \cdot b \; [\text{ by } 6, 13] \tag{17}$$

On this basis we can now make the respective case distinction:

Case 2.1: $0 \leq r' < b$ (18)

$$\vdash 0 \leq a - 2 \cdot q' \cdot b < b \quad [\text{ by } 18, 15] \tag{19}$$

$$\vdash q = 2 \cdot q' \quad [\text{ by } 2, 8, 19] \tag{20}$$

$$\vdash r = r' \quad [\text{ by } 16, 20] \tag{21}$$

Case 2.2: $b \leq r' < 2b$ (22)

$$\vdash b \leq a - 2 \cdot q' \cdot b < 2b \quad [\text{ by } 22, 15] \tag{23}$$

$$\vdash 0 \leq a - (2 \cdot q' + 1) \cdot b < b \; [\text{ by } 23] \tag{24}$$

$$\vdash q = 2 \cdot q' + 1 \quad [\text{ by } 2, 8, 24] \tag{25}$$

$$\vdash r = r' - b \quad [\text{ by } 16, 25] \tag{26}$$

5. *Fixpoint equation.* Since the above case distinctions are complete and disjoint, we can introduce a conditional fixpoint equation according to (a slightly generalized variant of) the rule 19. Note that we do not need a possibility operator here, since the postcondition determines the function uniquely. Hence, the fixpoint equation is a theorem.

$$\begin{array}{l} \vdash \textbf{Thm}\ \ div(a,b) = \\ \quad \textbf{If}\ a < b\ \textbf{Then}\ \ 0, a \\ \quad\quad \textbf{Else}\ \ \textbf{Let}\ q', r' = div(a, 2 \cdot b) \\ \quad\quad\quad\quad \textbf{In} \\ \quad\quad\quad\quad \textbf{If}\ r' < b\ \textbf{Then}\ 2 \cdot q', r' \\ \quad\quad\quad\quad\quad\quad\quad \textbf{Else}\ 2 \cdot q' + 1, r' - b\ \textbf{Fi}\ \ \textbf{Fi} \end{array} \tag{27}$$

6. *Termination.* In order to convert the above fixpoint equation 27 into a least-fixpoint definition, rule 20 requires that we find a termination ordering. Therefore we introduce the termination function

   $\textbf{Let}\ \ \tau(a,b)\ = 2 \cdot a - b$

   Then we can deduce for the recursive call $div(a, 2 \cdot b)$ within its context
   $0 < b \leq a$
   $\vdash \tau(a, 2 \cdot b) = 2 \cdot a - 2 \cdot b < 2 \cdot a - b = \tau(a,b)$
   On the other hand, we can deduce:
   $\tau(a,b) < 0$
   $\vdash \tau(a,b) = 2a - b < 0$
   $\vdash 2a < b$
   $\vdash a < b$
   And this is the condition of the termination case. Hence, theorem 27 can be converted into a definition.

$$\begin{array}{l} \vdash \textbf{Def}\ \ div(a,b) = \\ \quad \textbf{If}\ a < b\ \ \textbf{Then}\ \ 0,\ \ a \\ \quad\quad \textbf{Else}\ \ \textbf{Let}\ q', r' = div(a, 2 \cdot b) \\ \quad\quad\quad\quad \textbf{In} \\ \quad\quad\quad\quad \textbf{If}\ r' < b\ \textbf{Then}\ 2 \cdot q',\ \ r' \\ \quad\quad\quad\quad\quad\quad\quad \textbf{Else}\ 2 \cdot q' + 1,\ \ r' - b\ \ \textbf{Fi}\ \ \textbf{Fi} \end{array} \tag{28}$$

7. *Tail-recursive solution.* Our function still has a severe deficiency: It exhibits a so-called linear recursion, which requires a parameter stack for the implementation. However, since the argument function $2 \cdot b$ of the recursive call has the inversion property $(2 \cdot b) \div 2 = b$, a standard transformation rule for "recursion removal" can be applied.
   The idea behind this transformation is relatively simple: While going down into the recursion the function *div* first counts its parameter upward from b to some suitable value $2^i \cdot b$; on the way back it performs the actual calculations. But we can avoid the expensive stacking of the arguments by performing the backward counting with the help of the inverse function $b \div 2$.
   In spite of its conceptual simplicity the technical details of this transformation are a little intricate. This is a typical situation, where a subdevelopment is relatively mechanical but time-consuming. Moreover, it occurs similarly in many development tasks. Therefore, we codify the effect of this subdevelopment in a general rule such that the whole development now boils down to a single rule application. (This rule can be found e.g. in the textbooks of Bauer and Wössner [3] or Partsch [19]; but for illustration purposes we will also derive it explicitly in Section 4.8 below.) By applying this rule, our function becomes

$$
\begin{array}{l}
\textbf{Def}\ div(a,b) = divide(a,b)(b \cdot 2^i)(0,a) \\
\qquad\qquad \textbf{Where}\ i\ \textbf{Suchthat}\ b \cdot 2^i > a \\
\textbf{Def}\ divide(a,b)(b',q',r') = \\
\quad \textbf{If}\ b' = b\ \textbf{Then}\ q', r' \\
\qquad\qquad \textbf{Else}\ \ \textbf{Let}\ dvd = divide(a,b) \\
\qquad\qquad\qquad\qquad\ b'' = b' \div 2 \\
\qquad\qquad\qquad\ \textbf{In} \\
\qquad\qquad\qquad\ \textbf{If}\ a < b''\ \textbf{Then}\ dvd(b'')(0,a) \\
\qquad\qquad\qquad\qquad\qquad\ \textbf{Else}\ \ \textbf{If}\ r' < b''\ \textbf{Then}\ dvd(b'')(2 \cdot q', r') \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \textbf{Else}\ \ dvd(b'')(2 \cdot q' + 1, r' - b'') \\
\quad \textbf{Fi} \qquad\qquad \textbf{Fi} \qquad\qquad\qquad \textbf{Fi}
\end{array}
$$

The i in the body of $div(a,b)$ can be deduced from the precondition 4 of *div*: We simply take $i = n + 1$.

4.8 Derived Transformation Rules

An important feature of any calculus is that it not only allows us to apply its rules in concrete deductions but that it also permits the deduction of new "derived" rules, which may then in turn be used in other deductions. In other words, derived rules act as a kind of shortcut for deductions that occur similarly in many developments. We will exemplify this by a well-known rule for "recursion removal".

1. *Establishing the context.* Suppose that we are given the following schema for a recursive function:

   $$\textbf{Fun}\ f : s \to r$$
   $$\textbf{Def}\ f(x) = \textbf{If}\ C[x]\ \textbf{Then}\ A[x]\ \textbf{Else}\ B[x, f(K[x])]\ \textbf{Fi} \tag{1}$$

   Moreover, let us suppose that the argument expression $K[x]$ of the recursive call has an inverse; that is

   $$K[K^{-1}[x]] = x \tag{2}$$

2. *Specification of the new function.* Now we may introduce the following function:

   $$\textbf{Fun}\ \ F : s \to s \times r \to r$$
   $$\textbf{Spc}\ \ F(x)(x',y') = y$$
   $$\textbf{Pre}\ \ x' = K^i[x] \quad \text{for some } i \geq 0 \tag{3}$$
   $$y' = f(x') \tag{4}$$
   $$\textbf{Post}\ \ y = f(x) \tag{5}$$

   Note that (3) and (4) establish invariant relationships between the parameters x, x', and y'.

3. *Relating the two functions to each other.* Due to the postcondition (5) any call $F(x)(\ldots)$ yields as its result $f(x)$, provided that the preconditions of F are met. But since we want to avoid calls of f, the following choice is reasonable:

   $$\textbf{Thm}\ \ f(x) = F(x)(K^i[x], A[K^i[x]]) \tag{6}$$
   $$\textbf{Where}\ i\ \textbf{Suchthat}\ C[K^i[x]]$$

From the definition (1) it is immediatly seen that the preconditions (3) and (4) are met.

4. *Termination case for F.* We can terminate F when the parameter x' has reached x.

$$\textit{Case 1: } x' = x \tag{7}$$
$$\vdash \quad y = f(x) = f(x') = y' \text{ [by 5, 7, 4]} \tag{8}$$

5. *Recurrence.* When x' has not yet reached x, we can establish a straightforward recurrence relation for F.

$$\begin{aligned}
&\textit{Case 2: } x' \neq x \\
&\vdash F(x)(x', y') \\
&\quad = f(x) && \text{[by 5]} \\
&\quad = F(x)(K^{-1}[x'], f(K^{-1}[x'])) && \text{[by 5, 4]} \\
&\quad = \textbf{Let } x'' = K^{-1}[x'] \textbf{ In} \\
&\qquad F(x)(x'', \textbf{If } C[x''] \textbf{ Then } A[x''] \\
&\qquad\qquad \textbf{Else } B[x'', f(K[x''])] \textbf{ Fi }) && \text{[by 1]} \\
&\quad = \textbf{Let } x'' = K^{-1}[x'] \textbf{ In} \\
&\qquad \textbf{If } B[x''] \textbf{ Then } F(x)(x'', A[x'']) \\
&\qquad\qquad \textbf{Else } F(x)(x'', B[x'', y']) \textbf{ Fi } && \text{[by 2, 4]}
\end{aligned}$$

6. *Derived rule.* We can collect this subderivation into the rule given below. *Note,* however, that this rule is quite complex due to its generality. If we would choose $i = min\{j \mid C[K^j[x]]\}$, then the function F would simplify to the more efficient form

$$\begin{aligned}
&\textbf{Def } F(x)(x', y') = \\
&\quad \textbf{If } x' = x \textbf{ Then } y' \\
&\qquad \textbf{Else } \textbf{Let } x'' = K^{-1}[x] \textbf{ In } F(x)(x'', B[x'', y']) \textbf{ Fi}
\end{aligned}$$

However, as the example in the previous section demonstrates, this efficient version is not always asked for. Therefore we present the general variant here.

Rule 25 (recursion removal using function inversion)

$\vdash$ **Fun** $f : s \to r$

$\vdash$ **Def** $f(x) =$ **If** $C[x]$ **Then** $A[x]$ **Else** $B[x, f(K[x])]$ **Fi**

$\vdash$ $K[K^{-1}[x]] = x$

$\vdash \diamond$ **Fun** $F : s \to s \times r \to r$

Def $F(x)(x', y') =$

If $x' = x$ **Then** y'

Else **Let** $x'' = K^{-1}[x']$ **In**

If $C[x'']$ **Then** $F(x)(x'', A[x''])$

Else $F(x)(x'', B[x'', y'])$ **Fi Fi**

Def $f(x) =$ $F(x)(K^i[x], A[K^i[x]])$

Where i **Suchthat** $C[K^i[x]]$

where F is a new identifier

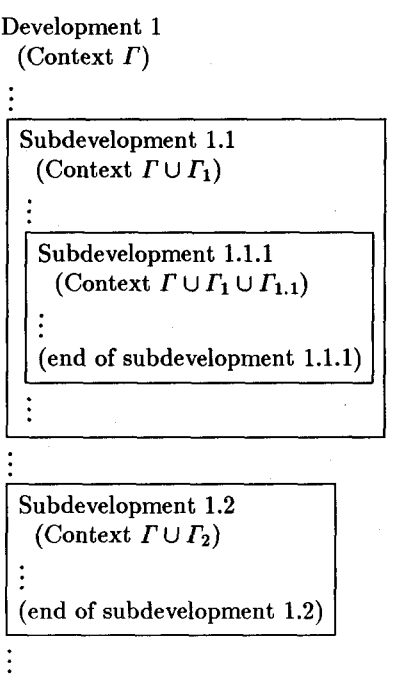
5 The Organization of Developments

A calculus alone does not yet provide a programming methodology. The rules of such a calculus are nothing but tools for ensuring that our reasoning is correct. On top of such a formalism we always need guidelines for organizing our developments. In addition, it would by nice, if there would also be an automated system to assist us in our work. In the sequel we will very briefly comment on these issues.

5.1 Developments and Subdevelopments

No reasonably large development task can be performed in one go. We always concentrate on one aspect at a time. In its simplest instance this paradigm could already be seen in our little examples *blog* and *div*. There we performed various case distinctions, each of which led to a little subdevelopment. From these examples we can infer basic principles of such an organization:

- ▷ Every (sub)development takes place within a certain *context*. Therefore we start each development by defining the appropriate context. And, of course, we need means for extending the context whenever the need arises.
- ▷ Subdevelopments are usually *nested* within each other. Therefore we inherit the context from the encompassing development level and extend it appropriately. Therefore we obtain a hierarchical tree-like structure of the following kind:



▷ However, experience shows that we usually work on more than one development simultaneously. (We all know the situation that we keep several piles of paper around, each belonging to one component of the system to be developed.) This way we can immediately extend or modify one part, when the need for doing so arises when working on some other part. Therefore we must be able to suspend (sub)developments and to resume them again.

▷ There are also situations, where we do not want to suspend our current development, even though a certain property that we need is not provided by the context. Then we must be able to *claim* this property. That is, we assume for the time being that the desired property indeed holds and just continue the actual development. In our formalism this means that the property is added to the current context as an unproven assumption. The rules for combining subdevelopments then ensure that the missing proof still has to be given.

The principles sketched above allow us to carry out developments in a well-structured and safe manner. However, they will usually end in an unordered collection of program fragments (signature fragments, axioms, theorems, function definitions, etc.). Therefore we usually have to subsequently clean up this unstructured assembly of facts.

This leads us into techniques for "programming in the large": Now we have to apply rules that organize the program fragments obtained so far into larger units that represent a proper modularization of the program. In Section 2.3 we have already listed many of the pertinent constructs that are needed for such a modularization. The rules, by which this activity can be performed in an orderly and correct fashion, are very much like the ones

given earlier in this paper. (A more detailed discussion is given in [26]. Moreover, these rules are the topic of intensive study in the research project KORSO that is currently under way at several German universities and research institutions.)

It goes without saying that the activities of detailed deriviation-in-the-small and global reorganization-in-the-large are interleaved. (It obviously does not make sense to first generate a huge mass of unstructured facts and afterwards try to get some structure into them.) But experience shows that this organization task needs as much flexibility as the detailed derivation tasks. A purely stepwise refinement – as it is often advocated in the literature on software engineering – is not adequate; it happens frequently that we have to restructure several modules by merging or splitting them, and by recombining them in new fashions.

5.2 Transformation Systems

The examples given in this paper clearly indicate, how helpful the assisitance by an automated system could be. Such a system could carry out the application of the individual rules, thus saving us from performing the tedious and boring activity of rewriting our programs in dozens or maybe even hundreds of versions. Moreover, such a system would be much safer than we are, because it makes fewer errors by thoughtlessness:

- ▷ When rewriting programs, we are likely to make copying errors, which is very unlikely for machines.
- ▷ If a rule requires many premises, we tend to overlook some of them or to just "believe" that they are fulfilled; a machine is completely stubborn here.
- ▷ In lengthy developments we may easily lose the orientation; a good system will keep an accurate record of the completed and pending activities.
- ▷ Finally, a system may ease the reuse of developments, because it allows us to redo earlier developments, when we encounter similar problems.

This sounds all very plausible. Yet, it has to be admitted that it is the details that are most intricate. A number of transformation systems have been created over the past years for experimental purposes. Examples are – among others – the CIP-system described in [1] or the KIDS-system described in [36]. The latter is probably the most advanced system of its kind. But even with this system a lot of experience is still required in order to actually carry out ambitious developments. Nevertheless, the progress that has been made is very promising.

The principles listed above represent some kind of "requirements analysis" for a transformation systems. As a matter of fact, the transformation system CIP-S has been designed on the basis of the formal calculus given in [24]. This CIP-calculus resembles the one given here, but it is much more technical, oriented towards the needs of an automated system. By contrast, our calculus in this paper is oriented at methodological considerations.

5.3 Strategies

The most challenging aspect of any methodology for program derivation is the formulation of *strategies* or at least *tactics*. Our examples have already indicated some instances of such principles, for example:

- ▷ We always start by formulating the *domain theory*, that is, the operations and properties that are relevant for the application area under consideration.
  Of course, we will usually not foresee all necessary aspects of the domain theory at the very beginning; therefore our development method allows us to extend the domain theory whenever the need for doing so arises.
- ▷ When dealing with a concrete function specification, we usually formulate subgoals by making appropriate case distinctions. Each of these subgoals then leads to a corresponding subdevelopment.
- ▷ Some of these subdevelopments aim at the formulation of suitable recurrence relations. This is frequently guided by the structure of the underlying data types. Here the generation axioms play a central role. (Theoretically speaking, this amounts to some kind of "programming with homomorphisms".)
- ▷ Finally we try to optimize the resulting algorithms by applying more technical transformations such as recursion removal and the like. Particularly helpful are rules known under buzzwords like "strength reduction" or "finite differencing", "fusion", "structure sharing", and the like.

The above techniques might be classified as tactics, by contrast to more encompassing and ambitious rules that might qualify as *strategies*. Typical instances are here:

- ▷ *Divide and conquer.* This is probably the most widespread technique in computer science. The formalization of this principle has been intensively studied by D. Smith [34]; this work also shows, how the method can be integrated in a system such as KIDS. [26] demonstrates the integration of this principle into a formal derivation calculus.
- ▷ *Global search* is another paradigm that has been integrated into the KIDS system; see [35].
- ▷ *Implementation* of data types is another challenging task. Attempts to do this by means of abstraction and representation morphisms have been made in [8] and [26].

This list could be extended. However, these papers also illustrate a problem with these strategies: The more general they are the more useless they become for the programmer. At a certain point one can no longer really benefit from the guidelines provided by the strategy, because they just formalize noncommittal truisms. So one has to achieve a subtle balacing between useful generality and abstractness on the one side and overdrawn universality on the other side.

5.4 An Example: "Majority Vote"

To illustrate the aforementioned principles we use another little example that recently has gained some popularity. We hope that the subsequent derivation will convey some of the potential elegance and clarity that can be achieved by adopting an algebraic deduction style.

Development *Majority Vote*

1. *Informal paraphrasing of the problem.* We are given a collection of coloured objects. We shall determine, whether more than 50% of the objects are equally coloured. Unfortunately, there exists an unlimited variety of possible colours.

2. *The domain theory.* As our first task we must establish the global context, that is, the theory of the underlying object domain. In our case, we are only concerned with collections of colours, since the nature of the objects themselves is irrelevant.

   Context(*Majority Vote*) = *Colour* ∪ *Bag*[*colour*]

   To ease our subsequent specifications it will be helpful to provide a function that yields the share of a colour in a bag.

   Fun $share : colour \times bag[colour] \to real$
   Axm $share(c, B) = \frac{c \text{ in } B}{card(B)}$

3. *Problem specification.* Within the given context we can now formalize the original problem.

   Fun $majority : bag[colour] \to colour$
   Spc $majority(B) = c$
   Pre $true$
   Post $share(c, B) > 50\%$
   Exception $c = nil$

   Remark: The notation **Exception** $c = nil$ is a shorthand for expressing the fact that the default value *nil* shall be taken as result, if there does not exist a value c that fulfills the rest of the postcondition.

4. *An important insight (eureka!).* No formalism in the world can replace the need to think. So we will have to ponder over our problem. (There has to be a better way than the obvious $\mathcal{O}(n^2)$-algorithm, where we simply count the colours one after the other.) The following observations put us on the right track:
   - ▷ For any given "candidate" colour a simple linear scanning will determine, whether this candidate has the majority or not.
   - ▷ So it would be nice, if we could designate one colour as the *only* candidate (which then could or could not have the majority).
   - ▷ Now let us suppose that in some given collection A no colour has the majority. This immediately has the following two consequences:
     - If we add a colour c to A, then c becomes the only candidate.
     - If we add two different colours $c \neq d$ to A, then there still is no majority.

   Now we have to turn this informal reasoning into a formal program derivation.

5. *Extending the Domain theory.* We need some auxiliary predicates to formulate our ideas.

   Fun $cand:\quad colour \times bag[colour] \to bool$
   Fun $anarchic : bag[colour] \to bool$
   Fun $uniform : bag[colour] \to bool$

   Axm $cand(c, B) \Leftrightarrow (\forall x \in B, x \neq c)\ share(x, B) \leq 50\%$
   Axm $anarchic(B) \Leftrightarrow (\forall x \in B)\ share(x, B) \leq 50\%$
   Axm $uniform(B) \Leftrightarrow (\forall x, y \in B)\ x = y$

   The properties that have been stated informally above, now can be formalized as follows:

Thm $anarchic(B) \wedge c \neq d \Rightarrow anarchic(B \oplus c \oplus d)$
Thm $anarchic(B) \Rightarrow cand(c, B \oplus c)$

Since these theorems immediately follow from the properties of *Bag* and elementary arithmetic, we skip their proofs here. (In a transformation system they would be recorded as "unfulfilled proof obligations".)

6. *Introducing subgoals.* Our main idea was to first find a candidate and then check, whether it indeed has the majority. Therefore, we split our original function into two functions.

Fun $findCandidate : bag[colour] \rightarrow colour$
Fun $dominant : colour \times bag[colour] \rightarrow bool$

Axm $findCandidate(B) = c \Rightarrow cand(c)$
Axm $dominant(c, B) \Leftrightarrow share(c, B) > 50\%$

With these specifications it is trivially demonstrated that the following property holds:

Thm $majority(B) =$ **Let** $c = findCandidate(B)$ **In**
If $dominant(c)$ **Then** c **Else** nil **Fi**

This theorem yields a definition for the function *majority* as soon as we have suitable implementations for the functions *dominant* and *candidate*.
The function *dominant* is no problem, because it is only based on elementary functions from *Bag*. Hence, the rule 21 converts the above axiom into the declaration.

Def $dominant(c, B) = share(c, B) > 50\%$.

7. *Implementing the function findCandidate.* After all these preparatory steps we can now concentrate on the real challenge of our development, viz. the finding of the candidate. To this end we introduce another auxiliary function that splits a given bag into two bags, an ararchic one and a uniform one.

Fun $split : bag[colour] \rightarrow bag[colour] \times bag[colour]$
Spc $split(B) = (A, U)$
Post $A \uplus U = B$
$anarchic(A)$
$uniform(U)$

In order to find an implementation for this specification we consider the possible cases for bags.

(a) *Termination case:* For empty bags the choice
Axm $split(\emptyset) = (\emptyset, \emptyset)$
immediately fulfills the postcondition of split.

(b) *Establishing recurrence relations.* For nonempty bags, which are of the form $B \oplus x$, we can perform the following deductions:
Assumption: **Let** $split(B) = (A', U')$.

$Case1 : U' = \emptyset$
$\quad Assumption:\ split(B \oplus x) = (A', \{x\})$
$\quad \vdash A' \uplus \{x\} = B \oplus x$
$\quad \vdash Anarchic(A')$
$\quad \vdash uniform(\{x\})$

$Case2 : x \in U'$
$\quad Assumption:\ split(B \oplus x) = (A', U' \oplus x)$
$\quad \vdash A' \uplus U' \oplus x = B \oplus x$
$\quad \vdash anarchic(A')$
$\quad \vdash uniform(U' \oplus x)$

$Case3 : x \notin U' \wedge y \in U'$
$\quad Assumption:\ split(B \oplus x) = (A' \oplus x \oplus y,\ U' \ominus y)$
$\quad \vdash A' \oplus x \oplus y \uplus U' \ominus y = B \oplus x$
$\quad \vdash anarchic(A' \oplus x \oplus y)$
$\quad \vdash uniform(U' \ominus x)$

This is an instance of (a variant of) the rule 19, because the case distinctions are complete and disjoint, and each case fulfills the postcondition. Together with the rules 21 and 25 this leads to the final definition

Def $split(B) =$
$\quad$ **If** $B = \emptyset$ **Then** $\emptyset, \emptyset$
$\quad\quad$ **Else** **Let** $(A', U') = split(B \ominus x)$ **Where** $x \in B$
$\quad\quad\quad$ **In**
$\quad\quad\quad$ **If** $U' = \emptyset$ **Then** $A', \{x\}$
$\quad\quad\quad$ **If** $x \in U'$ **Then** $A',\ U' \oplus x$
$\quad\quad\quad$ **If** $x \notin U' \wedge U' \neq \emptyset$ **Then** $A' \oplus x \oplus y,\ U' \ominus y$
$\quad\quad\quad\quad\quad\quad$ **Where** $y \in U'$
$\quad\quad\quad$ **Fi**
$\quad$ **Fi**

Now it remains to perform some optimizations for the data representation. For example, the *uniform* bags can be efficiently represented by one data element and the number of its occurrences. But this kind of cleaning up is quite straightforward and mechanical. Therefore we refrain from doing this here explicitly. What is important is the fact that we have succeeded to systematically develop an $\mathcal{O}(n)$ algorithm.

6 Conclusion

We have tried to demonstrate that programs can be formally developped on the basis of a very rigorous and simple calculus. In this framework the two seemingly irriconsilable paradigms of verification-oriented and transformation-oriented programming are naturally integrated.

The foundation of the approach is a strictly algebraic view, in which programs and specifications are unified. This algebraic treatment allows us in particular to deal with the development of data structures and algorithms simultaneously.

It is, however, evident that some assistance by a semi-automatic system is necessary in order to make this very stringent way of proceeding feasible in practice. But even if no such system is available, the general paradigm is still very helpful as an organization principle. The only difference will then be that many proofs are not carried out to the last detail but rather are only sketched – very much in the style of classical mathematical proofs.

Acknowledgement

The concepts presented in this paper were influenced by many discussions with Manfred Broy. Wolfram Schulte and the referees provided valuable comments and suggestions. Manuela Weitkamp-Smith helped in typing the manuscript. But my particular thanks go to Carola Gerke; without her assistance this paper would not have come into existence.

References

1. Bauer, F.L. et al.: The Munich Project Cip. Vol II: The Program Transformation System CIP-S. Lecture Notes in Computer Science 292. Berlin: Springer 1987.
2. Bauer, F.L., Möller, B., Partsch, H., Pepper, P.: Formal Program Construction by Transformations Computer-Aided, Intuition-Guided Programming. IEEE Trans. on Softw. Eng. 15:2 (1989), 165-180.
3. Bauer, F.L., Wössner, H.: Algorithmic Language And Program Development. Berlin: Springer 1982.
4. Bird, R.S.: An Introduction to the Theory of Lists. In: Broy, M. (ed.): Logic of Programming and Calculi of Discrete Design, NATO series F, vol. 36, Berlin: Springer 1986.
5. Bird, R.S.: Lectures on Constructive Functional Programming. In: Broy, M. (ed.): Constructive Methods in Computing Science. NATO series F, vol. 52, Berlin: Springer 1988.
6. Bird, R.S., Wadler, Ph.: Introduction to Functional Programming. Prentice Hall, 1988.
7. Broy, M.: Algebraic Methods for Program Construction: The Project CIP. In: Pepper, P. (ed.): Proc. of the Workshop on Program Transformation and Programming Environments. Berlin: Springer 1984, 199222.
8. Broy, M.: Deductive Program Development: Evaluation in Reverse Polish Notation as an Example. In: Broy, M., Wirsing, M. (eds.): Methods of Programming. Lecture Notes in Computer Science 544, Berlin: Springer 1991.
9. Broy, M., Wirsing, M.: Ultra-loose Algebraic Specification. Bulletin of the EATCS 35 (June 1988), 117-128.
10. Broy, M. et al.: The Requirement and Design Specification Language SPECTRUM – An Informal Introduction. Techn. Univ. München, Institut für Informatik, Techn. Rep. TUM-I9140, Oct. 1991.
11. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification. Vol I/II. Berlin: Springer 1985/1990.
12. Gentzen, G. Untersuchungen über das logische Schließens. Math. Zeitschrift 39 (1935), 176-210, 405-431.
13. Gordon, M.J.C.: The Denotational Description of Programming Languages. Berlin: Springer 1979.
14. Gries, D.: The Science of Programming. Berlin: Springer 1981.
15. Hughes, G.E., Cresswell, M.J.: An Introduction to Modal Logic. London: Methuen 1986.

16. Kahn, G.: Natural Semantics. In: Brandenburg et al. (eds.): Proc. STACS 87, Lecture Notes in Computer Science 247, Berlin: Springer 1987, 22-39.
17. Manna, Z.: Mathematical Theory of Computation. New York: McGraw-Hill 1974.
18. Manna, Z., Waldinger, R.: The Logical Basis for Computer Programming, Vol.1+2. Reading: Addison-Wesley, 1985,1990.
19. Partsch, H.: Specification and Transformation of Programs. Berlin: Springer 1990.
20. Partsch, H., Pepper, P.: Program transformations expressed by algebraic type manipulations. Technique et Science Informatiques 5:3 (1986), 197-212.
21. Pepper, P.: A Study on Transformational Semantics. In: Bauer, F.L., Broy, M. (eds.): Program Construction. Lect. Notes in Comp. Sc. 69, Berlin: Springer 1979, 322-405.
22. Pepper, P.: Algebraic Techniques for Program Specification. In: Pepper, P. (ed.): Proc. of the Workshop on Program Transformation and Programming Environments. Berlin: Springer 1984, 231-244.
23. Pepper, P.: Application of Modal Logics to the Reasoning About Applicative Programs. In: Meertens, L.G.L.T. (ed.): Program Specification and Transformation. Proc. IFIP TC2 Working Conf., Amsterdam: North Holland 1987, 429-449.
24. Pepper, P.: A simple calculus for program transformation (inclusive of induction). Science of Computer Programming 9 (1987), 221-262.
25. Pepper, P. (ed.): The Programming Language Opal-1. Technical Report, Fachbereich Informatik, Technische Universitt Berlin, 1991.
26. Pepper, P.: Transforming Algebraic Specifications – Lessons Learnt From An Example. In: Möller, B. (ed.): Constructing Programs From Specifications. Proc. IFIP TC2 Working Conference, Pacific Grove, Ca, May 1991. Amsterdam: North-Holland 1991, 399-426.
27. Pepper, P.: Literate Program Derivation: A Case Study. In: Broy, M., Wirsing, M. (eds.): Methods of Programming. Lecture Notes in Computer Science 544, Berlin: Springer 1991.
28. Plotkin, G.D.: A Structural Approach to Operational Semantics. DAIMI FN-19, Comp. Sc. Dept., Aarhus University, Aarhus, Denmark, Sept. 1981.
29. Sannella, D.T., Tarlecki, A.: On Observational Equivalence and Algebraic Specifications. J.Comp.System Sci. 34 (1987) 150-178.
30. Sannella, D.T., Tarlecki, A.: Toward Formal Development of Programs From Algebraic Specifications: Implementations Revisited. In: Ehrig, H. et al. (eds.): TAPSOFT '87, Lecture Notes in Computer Science 249, Berlin: Springer 1987, 96-100.
31. Sannella, D.T., Wirsing, M.: A Kernel Language for Algebraic Specification and Implementation. In: Coll. on Foundations of Comp. Th., Linkping 1983, Lecture Notes in Computer Science 158, Berlin: Springer 1983, 413-427.
32. Schmidt, D.A.: Denotational Semantics. Dubuque: Brown Publishers 1988.
33. Scott, D.: Outline of a Mathematical Theory of Computation. Proc. 4th Annual Princeton Conf. on Information Sciences and Systems, 169-176, 1970.
34. Smith, D.R.: The Design of Divide-and-Conquer Algorithms. Sci. Comp. Progr. 5 (1985) 37-58.
35. Smith, D.R.: Structure and Design of Global Search Algorithms. Techn. Rep. KES.U.87.12, Kestrel Institute, Nov. 1987. (to appear in Acta Informatica).
36. Smith, D.R.: KIDS – A Semi-automatic Program Development System. IEEE Trans. on Softw. Eng. 16:9 (1990) 1024-1043.
37. Wirsing, M.: Algebraic Specification. In: van Leeuven, J. (ed.): Handbook for Theoretical Computer Science. Amsterdam: North-Holland, 1990.