Views of
Formal Program Development

Eerke Boiten

Views of Formal Program Development

een wetenschappelijke proeve op het gebied van de Wiskunde en Informatica, in het bijzonder de informatica

Proefschrift

ter verkrijging van de graad van doctor aan de Katholieke Universiteit Nijmegen, volgens het besluit van het college van decanen in het openbaar te verdedigen op donderdag 6 februari 1992 des namiddags te 3.30 uur

door

Eerke Albert Boiten

geboren op 7 mei 1966
te Warns, thans gemeente Nijefurd

Promotor: prof. dr. H.A. Partsch

ISBN 90-9004747-6

Preface

Since 1988, the author of this thesis has been working on program specification and transformation in the STOP (Specifications and Transformations Of Programs) project, financed by NWO (the Netherlands Organization for Scientific Research) under grant NF 63/62-518. Studies on several subjects in this area resulted in a number of publications, collected in this thesis:

1. E.A. Boiten,
   Improving recursive functions by inverting the order of evaluation.
   Previous versions of (part of) this article have appeared as Technical Report no. 89-10, University of Nijmegen 1989, and in the proceedings of Computing Science in the Netherlands 1989, Eds. P.M.G. Apers, D. Bosman and J. van Leeuwen. It has been published in *Science of Computer Programming* **18**, pp. 139–179.

2. E.A. Boiten,
   Factorization of the factorial – an algorithm derived by playing with transformations.
   Previous versions of this article have appeared as Technical Report no. 90-18, University of Nijmegen 1990, and in Periodica Polytechnica, Technical University of Budapest, 1991.

3. H.A. Partsch and E.A. Boiten,
   A note on similarity of specifications and reusability of transformational developments.
   This article has appeared in the Proceedings of the IFIP TC2/WG2.1 Working Conference on Constructing Programs from Specifications, Ed. B. Möller, North-Holland 1991.

4. E.A. Boiten,
   Intersections of bags and sets of extended substructures – a class of problems.
   This article has appeared in the Proceedings of the IFIP TC2/WG2.1 Working Conference on Constructing Programs from Specifications, Ed. B. Möller, North-Holland 1991.

5. E.A. Boiten,
   Solving a combinatorial problem by transformation of abstract data types.
   This article has appeared in the Proceedings of Computing Science in the Netherlands 1991, Ed. J. van Leeuwen.

These articles are contained in chapters 2 to 6. Chapter 1 gives an introduction to the area of formal program development by transformations, an overview of the method and the relevance of the case studies in the subsequent chapters.

I could not have done this research without the invaluable guidance, assistance, and support from many people. I want to thank them all, in particular:

Helmut Partsch, my thesis supervisor, for pointing me to interesting subjects of research, for reading and thoroughly criticizing numerous drafts of all chapters, for guidance and motivation, and for a pleasurable cooperation over all, in particular writing a paper together;

Norbert Völker and Daniël Tuijnman for many inspiring conversations and their comments on my papers;

Alberto Pettorossi of the University of Roma II, for a motivating and pleasant two weeks in Rome, for interesting discussions, and his contribution to this thesis, as a member of the manuscript committee and otherwise;

Bernhard Möller of the University of Augsburg, for his rigorous and constructive criticism, both as a member of the manuscript committee and on other occasions;

Kees Koster, for much advice and criticism;

Frank Stomp, for his help in writing my first paper;

Dick van Leijenhorst, for providing valuable comments on combinatorial and complexity aspects;

Joanna Völker, for improving my English;

Mark van den Brand, Niek van Diepen, Max Geerling, Richard Bird of Oxford University, and various anonymous conference referees, for constructive reviews;

Ineke Kuster, Greta Löw, Jane Davies and many other colleagues, for a pleasurable working environment;

Lecturers at the STOP summer schools at Ameland, and other occasions, for inspiring lectures;

My friends and family, for their interest and support.

Of course, special thanks and love to Gwen who helped me do this by her continuous love, support, and patience.

Chapter 1

Introduction

1.1 Society, computers, reliable software

Computers play an increasingly important role in our society. Communication technology is advancing rapidly, more and more people own personal computers, and, most importantly, everywhere in organizations computers are at the centre of radical change. This development, however, is not beneficial in all of its aspects.

On the one hand, the fear of computers "taking over" has diminished greatly. People have become aware that computer intelligence will not surpass human intelligence for a long time yet. The results of thousands of man-years invested in "artificial intelligence" research are not as spectacular as the proponents of "strong" artificial intelligence have been predicting. Also, despite the advancements in communication technology, "Big Brother" can still watch only a little of what we do.

On the other hand, a number of different drawbacks of computerization have become more apparent. Introduction of new computers (*hardware*) and programs (*software*) usually takes more time and more money than originally planned. Even then, the errors those programs make are more absurd and less creative than those previously made by humans. Many industries and administrative organizations critically depend on the continuous and faultless operation of their computers. If the computer breaks down, the entire organization may come to a standstill. This occurs to such a great extent that often computers will even be blamed for failings of the organization itself – "Sorry, the computer cannot take that", "Why? Because the computer says so", etcetera.

However, ultimately humans are responsible for the "failings of computers", certainly in cases where physical factors like earthquakes and power failures can *not* be blamed. Thus, two of the most important problems to be addressed by the science of informatics[1] are

- the construction of reliable hardware, and
- the development of reliable software.

[1] Or maybe *computing science*, but that discussion is left to others.

This thesis aspires to be a contribution to the latter area. More precisely, we will be concerned with the *formal derivation* of *correct* and *efficient programs* from *formal specifications*.

As hinted at in the title, this thesis does not concentrate on presenting yet another slightly different version of what is well-known. A lot of work has been done in the area of formal program development, and the main part of this thesis consists of a number of case studies to enhance and supplement work by others.

The rest of this chapter gives an overview of the area of formal program development from a personal point of view. It will be made clear what is the position of the case studies in subsequent chapters in the formal program development process.

1.2 Program development

There is a long road to go from a customer's first description of a problem to a computer that actually runs the program solving the problem. The thorny path from imprecise ideas to formal specifications, the subject of *requirements engineering*, is not considered here. Likewise, the speedway from efficient (functional) programs to strings of bits, the subject of *compiler construction*, is omitted. What remains of the software development process is a part which forms the transition from the guessing games of requirements engineering to the fully automated compilers – i.e., an area where on the one hand, human intuition plays an important role, and on the other hand, higher level formalisms seek to transfer as much as possible of the human effort into compilers or automatic transformation systems.

A later section (1.4) describes the development process in more detail. An important point to be stressed now is that in our view, software development is a continuous process – initially, we have a specification which describes the problem to be solved, and development steps result in more operational, more specific, or more efficient versions. The idea that at each stage a solution to the problem is available gives a feeling of security. This is not just because this means we can cut off the development at any given time (since we already have a solution) – most likely our customer strongly prefers an operational and in most cases also an efficient solution. Indeed, by not requiring solutions to be operational we intentionally blur the distinction between programs and specifications. Consider programs as a special class of specifications, viz. the executable ones. A little juggling with terms gives us the opportunity to call something a *program* to suggest that it could be *executed*, and a *specification* to suggest that it *describes* our problem. If it is called a specification, most likely we do not care whether it can be executed – it is much more important that it gives an adequate and, hopefully, understandable description of the problem.

1.3 Historical background

From the early days of computers and programming language, the usual way of program development has been, and in many cases still is: write a program in a specific programming language based on requirements in informal natural language, edit it until the compiler accepts it, and run it and change it until it appears to satisfy the requirements.

The advent of higher level languages possessing an explicit semantics, like LISP [McC60] and Algol 60 [Nau60] led to several important improvements. First, these languages allow programs to be written using higher level control and data structures, and abstraction through meaningful names, which enhances their readability and thus their correctness. Also, they allow reasoning about programs, e.g. proving correctness with respect to some formal specification (written in e.g. predicate calculus). Thus, *a posteriori* a program could be proved correct [Bur68, Flo67]. Pioneer work was done by Cooper [Coo66], who demonstrated the equivalence of certain computations, thus providing a justification for several iterative implementations of certain types of recursive programs.

In the late 1970's, *functional* languages proved to be more amenable to formal manipulations. By their often mentioned *referential transparency*, more reasoning could be done on a local level without having to consider the side effects and aliasing problems of imperative languages like Algol and Pascal. The merits of functional languages have been extensively discussed in the literature, and thus the reader is assumed to be familiar with the arguments.

Burstall and Darlington [BD77] developed the *unfold-fold* method for program development. First, new functions are defined in terms of existing ones. Then the function calls of the old functions are *unfolded*, i.e. replaced by their bodies, appropriately instantiated. These expressions are manipulated, using all kinds of equivalences of the language and properties of the problem domain until instances of the original definitions are obtained. Under certain conditions, these can be *folded* again, i.e. replaced by calls of the new function. Steps like *fold* and *unfold* that transform programs into equivalent programs are called (correctness-preserving) *transformations*.

The CIP (Computer aided Intuition guided Programming) project at the Technical University of Munich, which started in the mid 1970's, extended the unfold-fold method with a number of aspects.

A pivotal role was played by the language CIP-L [BBB$^+$85], a language better suited to program transformation because it has a formal semantics designed with transformations in mind, and because it is a *wide spectrum language*, i.e. it allows descriptions at all levels of the formal development process. On the one hand, CIP-L includes non-executable specifications, including parts of mathematics (sets) and logic. On the other hand, the inclusion of imperative constructs and a *goto* command make it possible to proceed with the development down to a level close to machine language. Also, abstract data types were included in the language to allow a higher degree of modularity and to also consider the formal development of data structure implementations.

In the CIP project, methods were developed to proceed from non-operational specifications to operational ones. The unfold-fold method was extended with more complicated transformation rules that often abbreviate several steps in an unfold-fold development [BW82, Par90].

Since the late 70's, many activities in transformational and calculational program development have centered around IFIP Working Group 2.1 ("Algorithmic Languages and Calculi"), originally concerned with Algol 60 and Algol 68. Numerous relevant publications by members of this group, and the conferences sponsored by W.G. 2.1 on "Program Specification and Transformation" [Mee87] and on "Constructing Programs from Specifications" [Möl91] bear witness to this.

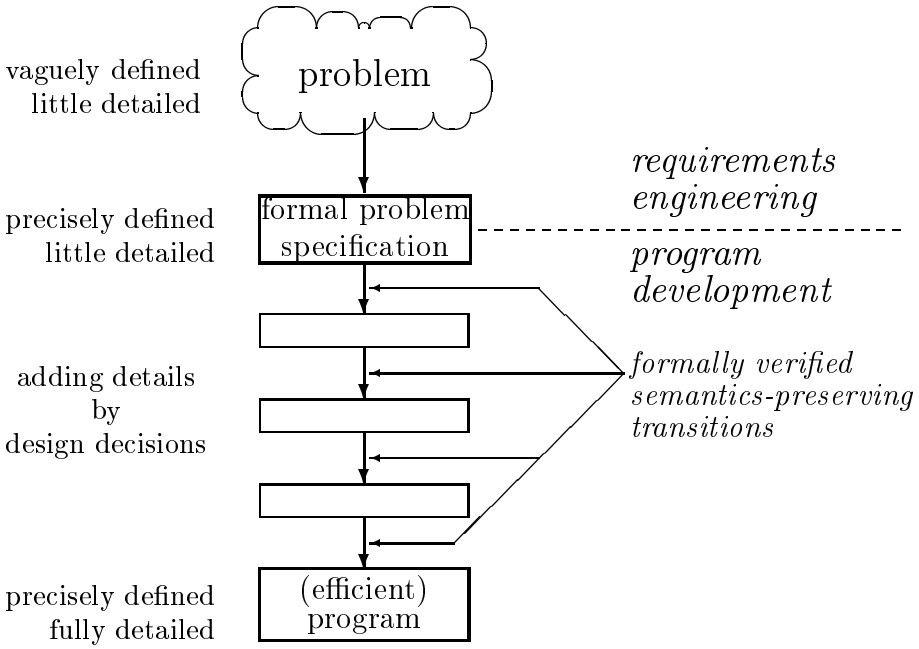
An important research subject in recent years has been the investigation of the *strategical* and tactical level of transformational programming. Feather [Fea87] gives an overview of research on transformation strategies. Of particular importance in this area is the work by Smith and others [SKW85, SL90, Smi90] on formal descriptions of transformation strategies and their implementation in a transformation system.

Bird and Meertens argued [Mee86] that languages like CIP-L that comprise conventional programming language constructs are not ideal for program transformation. In order to bring program development closer to mathematical calculation, they devised succinct and powerful notations for characteristic operations. By using these notations, commonly called BMF (for Bird Meertens Formalism) or Squiggol, program derivation can be done by syntactical calculation on mostly uninterpreted, one line formulas. The first area they aimed at was the theory of *lists* [Bir87] or *sequences*, and the related data types of sets, bags and trees. This work has resulted in a by now reasonably fixed and stable basic collection of laws and theorems (equivalences) that has risen to the status of informal standard. Specifications in BMF are already operational; in some cases this forces premature choices, e.g. of the way of processing data types.

In BMF, an important role is played by inductive function definitions over types that follow the inductive definition of the type itself, the so-called homomorphisms or *catamorphisms*, based on an old idea by von Henke [vH76]. Malcolm [Mal90] and others generalized this kind of definitions, using category theory, to a general theory of initial data types, catamorphisms, and corresponding so-called promotion laws. Backhouse et al [BdBM+91] generalized this approach to *relations*. Meijer, Fokkinga and Paterson [MFP91] describe very general kinds of recursion by including the concept of anamorphism, dual to catamorphism.

1.4 A software development model

The following picture [Par90] represents the basic software development model which is assumed in this thesis.



From an informally stated problem, by *requirements engineering*, a formal specification is obtained. By separating this mysterious process from the actual program development, admittedly the most difficult part of software engineering is left out. However, by using powerful specification languages (i.e. languages with, at the very minimum, non-operational constructs and representations of often used concepts), the level of abstraction of specifications can be raised and thus the scope of requirements engineering can be reduced.

A formal specification may be viewed as a *contract* between the customer and the software developer. The developer can fulfill his task by just implementing the formal specification. This implementation proceeds via semantics preserving transformations – i.e., the resulting programs are "*correct by construction*". This eliminates the feedback loops from traditional "software life cycle" models. The cycles concerned with *validation* form part of requirements engineering. This means that some testing and, in particular, *prototyping* are considered to be part of requirements engineering.

Testing and "maintenance" in the sense of removing errors in the program are also left out. Due to the method of developing programs, the final program contains no errors that were not already present in the initial specification. So if an error is detected in the final program, this means that the "contract" is still fulfilled, even though it did not correctly represent the problem. In principle, this means that the entire development needs to be redone – in practice, however, much of the original specification and some of the original development can be reused. This subject is discussed more extensively in chapter 4.

1.4.1 Specifications

As mentioned before, specifications need not be operational. By using a wide spectrum language we can specify *what* has to be done, often without mentioning *how*. This means

that the specification allows the derivation of all possible solutions, without suggesting particular ones.

For example, the position of the maximum in a nonempty array m, indexed by a type **ind**, can be specified by

1.1 **some ind** $i : \forall$ **ind** $j : m[i] \geq m[j]$.

The **some**-expression denotes an arbitrary choice between the values that satisfy the predicate. This is necessary here, since the maximum may occur more than once. Thus, the specification is *non-deterministic*, i.e. there are several possible solutions, and a program satisfying the specification may choose either of these. Likewise, the position of the "rightmost" maximum is specified by

1.2 **that ind** $i : \forall$ **ind** $j : m[i] > m[j] \vee (m[i] = m[j] \wedge i \geq j)$.

Here, the **that**-expression denotes the unique value satisfying the predicate, which is defined only if exactly one such value exists.

Operational specifications for such problems are cumbersome, and biased as to the direction of processing the array. Chapter 4 is concerned with general solutions to problems of this "linear search" nature.

On the other hand, no one would define the Fibonacci function in a non-operational way. There, the most natural specification is a recurrence relation, expressible by a recursive definition, i.e.

1.3 $fib(n) =$ **if** $n \leq 1$ **then** 1 **else** $fib(n-1) + fib(n-2)$ **fi**.

The derivation of an efficient program for this function is presented as an example in chapter 2.

An important role in specifications is also played by *abstract data types* (ADT's). An ADT describes in an abstract way a certain data structure with its characteristic operations. For an extensive description of ADT's, their syntax and semantics, in CIP-L, cf. [BBB+85, WPP+83]. As an example of data types occurring in this thesis, consider the type of *lists* (or sequences) over an arbitrary type α, defined in the following way:

$$\frac{}{\varepsilon : List(\alpha)} \qquad \frac{x : \alpha}{[x] : List(\alpha)} \qquad \frac{a, b : List(\alpha)}{a \# b : List(\alpha)}$$

These so-called introduction rules state that a *List* over α can be either the empty list ε, or a singleton list constructed from an element of type α, or the concatenation ($\#$) of two *List*'s. They also define the well-formed terms of type *List*. Most type definitions are not complete without a set of laws, usually equivalences stating that some terms are considered equal. The laws for *List* are ($\#$ is associative with unit ε):

$$\begin{aligned} a\#(b\#c) &= (a\#b)\#c \\ \varepsilon\#a &= a \\ a\#\varepsilon &= a. \end{aligned}$$

Usually we take the initial model semantics for such types [WPP+83], which means that two different terms of the type are equal only if that is implied by the laws.

In this thesis ADT's mainly serve as abstract descriptions of data structures with their operations. Apart from that, they also provide a way of separating concerns on the aspects of data and control in program developments. ADT's can be implemented by more concrete structures separately from the program development, although usually the context determines which is the most efficient implementation of a data type [Par90].

1.4.2 Developments

It has long been acknowledged that the development from specifications to programs should be a *stepwise* one. In our approach, these are *formal* steps: at each intermediate stage, the current program constitutes a solution to the initial specification, and this property is maintained by the requirement that each transition be a *semantics preserving* transformation. Previously, the term *correctness preserving* was used in this context, implying the transformation of programs into equivalent ones; a semantics preserving transformation may also result in a *descendant* program, i.e. one that returns a subset of the possible values of the previous one.

We follow Partsch [Par90] in distinguishing three subphases of transformational developments, characterized by the respective intermediate results:

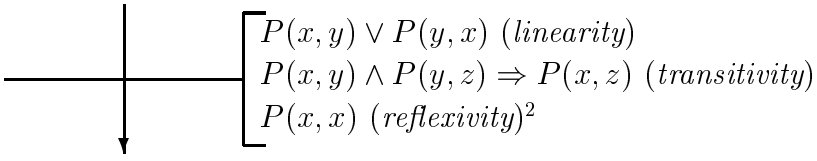
1. from descriptive specifications to operational (recursive) ones (*operationalization*);
2. from recursive programs to efficient tail-recursive programs (*optimization*);
3. from efficient tail-recursive programs to efficient imperative ones (*coding*).

In the first phase, non-operational constructs (e.g. quantifiers, set comprehensions) are replaced by operational ones, usually by enumeration of domains, or by induction over recursive types. An example of a transformation rule in this phase is the following one.

Transformation 1.4

---

some (**nat** $x : x \leq N$) : $\forall$(**nat** $y : y \leq N$) : $P(x, y)$



$f(N)$

where

[2]Mentioned for clarity only; it is already implied by linearity.

$f(\textbf{nat } n)\textbf{nat } = \textbf{ if } n = 0 \textbf{ then } 0 \textbf{ else } g(f(n-1), n) \textbf{ fi},$
$g(\textbf{nat } x, \textbf{nat } y)\textbf{nat } = \textbf{ if } P(x, y) \textbf{ then } x \textbf{ else } y \textbf{ fi}$

This represents the validity of transforming a program which is an instance of the first scheme into an instance of the second scheme, provided the applicability conditions (next to the arrow) are fulfilled. Names (like N and P) that are not bound in the input or output scheme are variables that can be instantiated to obtain a special case of the transformation rule. Free variables that only occur in the applicability conditions are assumed to be universally quantified.

The above transformation rule can also be viewed as a summary of a small transformational development (here, types are omitted for brevity):

$\textbf{some } (x : x \leq N) : \forall (y : y \leq N) : P(x, y)$

$=$ { embedding }

$f(N)$ **where**
$f(n) = \textbf{some } (x : x \leq n) : \forall (y : y \leq n) : P(x, y)$

$=$ { case introduction: $n = 0$? }

$f(N)$ **where**
$f(n) = \textbf{if } n = 0 \textbf{ then some } (x : x \leq n) : \forall (y : y \leq n) : P(x, y)$
$\quad\quad \textbf{else some } (x : x \leq n) : \forall (y : y \leq n) : P(x, y) \textbf{ fi}$

$=$ { simplification in **then**-branch, using reflectivity of P }

$f(N)$ **where**
$f(n) = \textbf{if } n = 0 \textbf{ then } 0$
$\quad\quad \textbf{else some } (x : x \leq n) : \forall (y : y \leq n) : P(x, y) \textbf{ fi}$

$=$ { abstraction; case introduction in **else**-branch, allowed since $f(n)$
is defined for all n, due to applicability conditions }

$f(N)$ **where**
$f(n) = \textbf{if } n = 0 \textbf{ then } 0$
$\quad\quad \textbf{else } r = f(n-1);$
$\quad\quad\quad \textbf{if } P(r, n) \textbf{ then some } (x : x \leq n) : \forall (y : y \leq n) : P(x, y)$
$\quad\quad\quad\quad \textbf{else some } (x : x \leq n) : \forall (y : y \leq n) : P(x, y) \textbf{ fi fi}$

$\subseteq$ { from $r = f(n-1)$ it follows that $\forall (y : y \leq n-1) : P(r, y)$;
extend quantification and choose r in **else then**-branch;
linearity, reflexivity and transitivity in **else else**-branch }

$f(N)$ **where**
$f(n) = \textbf{if } n = 0 \textbf{ then } 0$
$\quad\quad \textbf{else } r = f(n-1);$
$\quad\quad\quad \textbf{if } P(r, n) \textbf{ then } r \textbf{ else } n \textbf{ fi fi}$

$=$ { abstraction, eliminating common subexpression, unfold r }

$f(N)$ **where**
$f(n) =$ **if** $n = 0$ **then** 0 **else** $g(f(n-1), n)$ **fi**
$g(x, y) =$ **if** $P(x, y)$ **then** x **else** y **fi**.

There is one step in the above development which is not an equivalence, but a refinement ($\subseteq$). At this point, the "rightmost" value is chosen. This is why transformation 1.4 only has a single arrow: only the transition from the upper program scheme to the lower one is semantics preserving.

If we take **ind** $\equiv$ (**nat** $x : x \leq N$) for some $N \geq 1$, then the application of transformation rule 1.4 to specification 1.1, which is allowed since $P(i, j) = m[i] \geq m[j]$ is linear, transitive, and reflexive, yields:

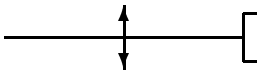
1.5 $f(N)$
where $f($**nat** $n)$**nat** $=$ **if** $n = 0$ **then** 0 **else** $g(f(n-1), n)$ **fi**,
$g($**nat** $x,$ **nat** $y)$**nat** $=$ **if** $m[x] \geq m[y]$ **then** x **else** y **fi**.

In general, the rough shape of the algorithm arises during the first phase. Chapter 4 gives some idea of the variety of recursive algorithms that can be derived from one descriptive specification.

Because of the referential transparency of functional languages, the second phase is at the correct level of abstraction for introducing important optimizations and clever ideas. Some of the better known transformations on this level are finite differencing [PK82], tupling [Pet84b], and accumulation [Bir84, Boi91b]. A particular version of the latter transformation is captured in the following rule:

Transformation 1.6

$f($**m** $x)$**m** $=$ **if** $T(x)$ **then** $H(x)$ **else** $E(x, f(K(x))$ **fi**



$E(x, E(y, z)) = E(E(x, y), z)$ *(associativity)*

$f($**m** $x)$**m** $=$ **if** $T(x)$ **then** $H(x)$ **else** $f'(x, K(x))$ **fi where**
$f'($**m** $y,$ **m** $x)$**m** $=$ **if** $T(x)$ **then** $E(y, H(x))$ **else** $f'(E(y, x), K(x))$ **fi**,

---

which also can be viewed as the result of a small transformational development. Its application to program 1.5 (note that g is associative) yields:

1.7 $f(N)$
where $f($**nat** $n)$**nat** $=$ **if** $n = 0$ **then** 0 **else** $f'(n, n-1)$ **fi**,
$f'($**nat** $r,$ **nat** $n)$**nat** $=$ **if** $n = 0$ **then** $g(r, 0)$ **else** $f'(g(r, n), n-1)$ **fi**,
$g($**nat** $x,$ **nat** $y)$**nat** $=$ **if** $m[x] \geq m[y]$ **then** x **else** y **fi**,

where the call of f can be unfolded and replaced by (using the premise $N \geq 1$)

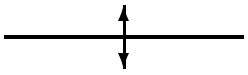
1.8 $f'(N, N-1)$.

Another important transformation on the optimization level, viz. *inverting the order of evaluation*, is discussed in chapter 2.

The third phase is of minor importance. The transition from tail-recursive functions to while-loops can be seen as a simple change of notation, as may become clear from the following transformation rule:

Transformation 1.9

$f(X, Y)$ **where**
$f(\mathbf{m}\ x, \mathbf{n}\ y)\mathbf{p} =$
if $T(x, y)$ **then** $H(x, y)$
else $f(K(x, y), L(x, y))$ **fi**



var m $vx := X$; **var n** $vy := Y$;
while $\neg T(vx, vy)$
do $(vx, vy) := (K(vx, vy), L(vx, vy))$ **od**;
$H(vx, vy)$

The application of this rule to program 1.7/1.8 yields

1.10 **var nat** $vr := N$; **var nat** $vn := N - 1$;
while $vn \neq 0$
do $(vr, vn) := (g(vr, vn), vn - 1)$ **od**;
$g(vr, 0)$.

Relatively little more can be done at this point. The following steps could just as well be executed by a moderately clever compiler. We now focus on the statement inside the loop.

$(vr, vn) := (g(vr, vn), vn - 1)$
$=$ { sequentialization: assignment to vn is independent of vr }
$vr := g(vr, vn); vn := vn - 1$
$=$ { unfold g }
$vr :=$ **if** $m[vr] \geq m[vn]$ **then** vr **else** vn **fi**;
$vn := vn - 1$
$=$ { distributivity of assignment over conditional }
if $m[vr] \geq m[vn]$ **then** $vr := vr$ **else** $vr := vn$ **fi**;
$vn := vn - 1$
$=$ { elimination of identity assignment }
if $m[vr] \geq m[vn]$ **then skip else** $vr := vn$ **fi**;
$vn := vn - 1$

Also unfolding g in the result expression leads to our final program:

1.11 **var nat** $vr := N$; **var nat** $vn := N - 1$;
while $vn \neq 0$
do if $m[vr] \geq m[vn]$ **then skip else** $vr := vn$ **fi**;
$vn := vn - 1$
od;
if $m[vr] \geq m[0]$ **then** vr **else** 0 **fi**.

See chapter 3 for another example of a derivation that is carried through to the imperative level. There, it is also the case that (except maybe for sequentialization) all optimizations possible on the imperative level had already been implemented in optimizing compilers 20 years ago.

Bauer and Wössner [BW82] and Partsch [Par90] even give transitions from while-loops to labels and jumps, to show that transformational development can continue down to a very primitive language level.

1.4.3 Extensions to the model

When one also considers transformations of data structures, the software development process gets a bit more complicated. Partsch [Par90] gives a survey of the various possibilities for data type implementation: independently or jointly with the program development, or based on libraries of standard implementations.

Of course, the program development model as sketched is a very simple one. It suggests that programs are changed in small steps only. The example development starting from specification 1.1 shows that often these small steps can be condensed into more general transformation rules.

A recurrent subject in software development is that of *reuse*. Indeed, no development methodology can be practical if it does not allow using previously acquired knowledge in new developments. In order to do so, general knowledge has to be captured in abstract concepts. On the level of specifications, these concepts are the algebraic data types. On the level of transformations, these are the schematic transformation rules. On the level of transformational developments, these are the reusable developments (further discussed in chapter 4). This results in an adaptation of the software development model: where possible, reduction to known concepts should take place before further transformational development.

A complementary approach is the investigation of *theories* of certain (combinations of) abstract types with their typical operations and corresponding equivalences (transformation rules), for example the theory of lists investigated in [Bir87]. Another example is the class of problems discussed in chapter 5.

One may argue that reusable program developments are "just" theorems in such a theory. Even then, however, reuse of program developments is relevant since it provides a way of looking for useful new theorems.

1.4.4 On notation

Each particular notation has its advantages and disadvantages. This thesis does not take a dogmatic view in this respect – usually, notation is chosen based on considerations of *expressiveness*, *clarity*, and *succinctness*, in that order. For the example derivation above, e.g., a notation close to CIP-L seemed most appropriate, since this allows the expression of descriptive specifications. Chapter 2 also employs a CIP-L like notation, since it discusses forms of recursion that can only recently be expressed in BMF [MFP91]. BMF-like notation, however, showed to be more suitable for describing inductions over data types in chapters 5 and 6.

1.5 Computer aided formal software development

A recurring argument in the area of transformational programming is to what extent program transformations *could* and *should* eventually be applied by automatic transformation systems. This question will be discussed by considering a number of related subjects in informatics. A more elaborate study may be found in [BvdBvD$^+$90].

As mentioned before, *compilers* are at the lower end of program developments – usually, an efficient program in a higher level language suffices as an end product. Several optimizing transformations discussed in literature had already been known from the area of optimizing compilers (e.g. *finite differencing* [PK82] which was known as *strength reduction* [Ear76]). Obviously, in cases where it is clear how such transformations can be applied, it makes no sense to apply them manually when the compiler can also do the job. The extent to which such transformations should be applied at all is still open to debate, since their application may obscure possibilities for other compiler level transformations.

Thus, in this sense fully automatic transformation systems are "just" optimizing compilers [Fea87] – although the artificial separation between these areas may stand in the way of cross-fertilization.

Automation of program development can also be seen as an interesting test case for *artificial intelligence* research. This is because, in general, program development problems cannot be solved by pure and simple-minded calculation, but *heuristics* are necessary for limiting the search space to a manageable size.

An important achievement in this respect is the KIDS system, developed at Kestrel Institute by Smith [Smi90]. In KIDS, a number of relatively well understood classes of problems and program development strategies have been automated, in such a way that their application requires relatively little user action.

In general, however, artificial intelligence techniques will not contribute very much to solving the program development problem, at least not while still so little is known about useful program development heuristics (cf. Feather's remark that "*little is known about how to deal with all these issues at once in anything other than an ad-hoc manner*" [Fea87]).

The third related area, that of *programming environments*, syntax directed editors, etc. undeniably provides a useful contribution. Many of the errors made in program derivations are caused by errors in copying and substitution. Much of the annoyance with easily readable

languages like CIP-L comes from the excessive copying necessary in applying transformations. A sophisticated language based editor can be a very useful tool for doing all this clerical work.

Summing it up, currently program transformation systems are most useful as clerical assistants. As our knowledge of program development grows, more tasks are viewed as clerical and can be automated. For the derivation of interesting new algorithms, however, *intelligence* and *intuition* will remain necessary – attributes which computers have not yet shown they possess.

1.6 Overview of this thesis

This thesis contains a number of case studies aiming at the exploration of new territories in the area of program specification and transformation. The early days of transformational programming, when everyone could derive a new version of the factorial or Fibonacci function, and then call it a generally applicable strategy, have gone. The central question nowadays is not *whether* e.g. the pattern matching algorithm by Knuth, Morris and Pratt, or that by Boyer and Moore, can be derived – the emphasis is on the *structure* and *effort* of such derivations. Overall, current research is concentrated on higher level knowledge: data types with their characteristic algorithms and properties (*theories of data*); strategies so well understood that they can be automated (*theories of programs*); derivations and their general shapes (*theories of derivations*). Besides that, another important area is the extension of the transformational/calculational approach to other areas, like "relational programming", and program derivation for all kinds of parallel architectures.

Chapter 2 gives a comprehensive survey of one particular transformation strategy, viz. inverting the order of evaluation. This strategy entails the derivation of equivalent functions that use in their recursive evaluations the same arguments in an inverted order. For linear recursive functions, this may improve efficiency because intermediate results of recursive function evaluations are delivered in a different order. Also, usually it results in *tail-recursive* functions that have an immediate imperative interpretation.

In the general sense in which this strategy is described in chapter 2, it is the most important optimization of tree-like recursive functions. Many tree-like recursive functions are defined in such a way that they require multiple evaluations of certain recursive calls. Evaluating recursive calls in an inverted order ensures that no call is evaluated twice, at a usually small overhead cost.

Many of the transformations in chapter 2 have been described before in the literature. However, an overview relating all these techniques has not been presented before. Another important contribution of this thesis is the abstract description of *tabulation* using new language constructs **remember** and **recall**. This allows separate consideration of function *arguments* and function *results*.

Apart from aptly illustrating the transformations on linear recursive functions from the preceding chapter, chapter 3 also demonstrates the state of the art in recursion simplification transformations. Starting from an algebraic property of the factorial function, guided by simple heuristics, a previously unknown algorithm is derived. The techniques of *finite*

differencing and *accumulation*, together with inverting the order of evaluation, suffice to derive a sophisticated algorithm that is only intelligible by way of its derivation.

Chapter 3 also presents a definition of the factorial function using two processors and a one-way communication channel. The communication channel is modelled by a sequence, and the values on the channel are defined by an assertion over that sequence. From this definition, using analogous steps, a variant of the new algorithm is derived that runs on this particular architecture.

Chapter 4 considers the problem of *reuse* of *transformational developments*. It is well understood how parts of specifications (e.g. abstract data types, with libraries of implementations) and individual transformation steps (i.e. transformation rules) can be reused. It has also been claimed that mechanical reuse of complete developments is possible. The results of considering a number of analogous derivations in this chapter indicate that this claim is somewhat preposterous. Only by describing the transformation steps in a very abstract way (here, using just natural language) and by considering very general specifications, can the developments be reused. The central concept is *similarity*, and several definitions of this informal notion are given, each leading to a particular kind of reuse of derivations. The running example is a derivation of *linear search*. Variants of this derivation lead to several interesting search algorithms, culminating in derivations by reuse of the pattern matching algorithms by Boyer and Moore [BM77] and Knuth, Morris and Pratt [KMP77]. Thus, this study also follows up to previous work by the Nijmegen STOP group on derivations of pattern matching algorithms [PS90, PV91].

Chapter 5 generalizes the *specification* of pattern matching. It describes a class of problems that can be viewed as a generalization of pattern matching problems. The essence of pattern matching is considered to be the intersection of a particular set with a bag (multiset) of extended substructures of a structured object. The set contains the patterns, the extended substructures are possible occurrences, extended with labels that mark their positions in the original object. This leads to the first ideas on an (interesting) theory on (extended) substructures. It is shown how the abstract description of this class of problems lends itself to calculation in a BMF style. Also, clearly exhibiting the basic structure of such problems facilitates connecting them with various solution strategies.

Chapter 6 gives an application of techniques from program development in a different area, viz. combinatorics. By describing a given combinatorial problem in terms of abstract data types with equivalences, and transforming those data types, a reduction to a known problem is obtained. A previous study of this problem employed the generally accepted specification mechanism of context free grammars. Our results indicate that abstract data types are much more suitable for such purposes, since they allow the introduction of arbitrary explicit equivalences on data types, whereas context free grammars only have the implicit associativity of concatenation.

Chapter 2

Improving recursive functions by inverting the order of evaluation

2.1 Introduction

In the last decade, *transformational programming* has proved to be an appropriate methodology for developing correct programs (see, e.g. [Fea87, Par90]). The essence of this methodology is the derivation of (efficient) programs from formal specifications by applying *semantics preserving transformations*, i.e. applying transformation rules that result in programs that are semantically equivalent or more defined and determinate (*descendant* or *refinement*). An overview of the transformational method can be found in [Par90].

An important research topic in transformational programming is the identification of relevant transformation *strategies*. Feather argues in his survey paper [Fea87] that little is known about transformational programming on the *strategic level*, and proceeds to describe a large number of so-called transformation *tactics*, like fusion, filter promotion, precomputation. We follow Pettorossi (and others) by calling these tactics *strategies*. Thus, our definition of a transformation strategy is: a larger conceptual step in a transformational development which can be described at a more abstract level.

Several useful strategies have been identified and extensively described, like accumulation [Bir84] and finite differencing [PK82]. Many so-called algorithm theories have been described by Smith [SL90]. This chapter describes one such strategy, viz. *inverting the order of evaluation*, which has been the subject of a respectable body of research. No general framework, however, has been presented yet to put together the large number of individual transformation rules that have been developed.

Informally speaking, inverting the order of evaluation is a transformation on recursive functions that results in functions that use the same arguments but evaluate them in an inverted order. Several specific transformation rules for inverting the order of evaluation have been described. Cooper [Coo66] inverted the order of evaluation of the factorial function. Cohen presented a number of transformation rules for classes of tree-like recursive functions in [Coh83]. The same kind of analysis can also be used for finding linear recursive definitions of these functions. Pettorossi used the tupling strategy for linearizing many of Cohen's and

other examples in [Pet84b], as did Harrison (in an FP-setting) in [Har88]. Finally, also the well-known implementation of recursion by stacks is strongly related to inverting the order of evaluation [BW82, PP86].

This chapter attempts to provide more insight into this strategy by giving a structured overview of all these techniques and showing their relationship with other techniques. Furthermore, the range of applicability of many of the transformation rules is widened by presenting them in a more general form.

2.2 Inverting the order of evaluation

Consider a linear recursive function, i.e. a function f such that $f(x)$ is defined in terms of x and possibly $f(K(x))$ for some expression K. For a certain initial value x_0, the evaluation of $f(x_0)$ causes a number of recursive calls $f(x_1), f(x_2), \ldots$ with $x_{i+1} = K(x_i)$. Also, if f is well-defined, and thus terminates, this number of recursive calls is finite, say m. Our goal is then to find a function f' which computes the same value as f, that is also linear recursive with $f'(x)$ defined in terms of x and possibly $f'(K'(x))$. Furthermore, the sequence of calls of f' for computing $f(x_0)$ is required to be $f'(x_m), f'(x_{m-1}), \ldots, f'(x_0)$. This will be elaborated in section 2.4.

For linear recursive functions, the technique of inverting the order of evaluation aims at the improvement of their efficiency, mainly with respect to execution time. Inverting the order of evaluation as described above may improve efficiency in several ways. The resulting function may have a tail-recursive structure, which means that it can immediately be transformed into a loop. Also, the intermediate results of the computation may be different or computed in a different order. This may enable optimization by way of finite differencing [PK82] or similar techniques. An example of this can be found in chapter 3.

For tree-like recursive functions, i.e. functions f such that $f(x)$ is defined in terms of x and $f(K_1(x)), \ldots, f(K_n(x))$ for some finite $n \geq 2$, inverting the order of evaluation shows a more definite improvement. In that case, the evaluation structure of f is not linear, but has a tree shape. Generally, some arguments to f occur at several places in one such tree. The transformation rules described in section 2.7 eliminate such *multiple evaluations of identical function calls* by introducing a linear recursive function that traverses the tree in some bottom-up fashion.

The next section introduces the language used, and some special notations. In section 2.4 a variant of Cooper's inversion of the factorial function is described, and a very general version of this transformation is presented.

In section 2.5, stacks are introduced in order to show that inverting the order of evaluation is, in principle, always applicable to linear recursive functions. This gives a new perspective on the preceding results.

Section 2.6 prepares for the treatment of tree-like recursive functions by replacing stacks by a general control structure (**remember/recall**) that can be implemented in various ways. Section 2.7 presents a number of direct inversions of tree-like recursive functions. General "common generator" and "commutative periodic redundancy" rules, based on those from [Coh83], are given. A new rule for "bounded disjoint generations" is introduced. Finally,

we consider a new general tabulation rule, using *compatible orderings*. Section 2.8 discusses the *tupling* or linearization strategy, which may be applied to certain tree-like recursive functions yielding linear recursive ones. It is shown how it can be applied to several of the classes of functions described in the preceding section. From this, some general heuristics are derived. In section 2.9 a final comparison with other approaches is made and conclusions are presented.

2.3 Language and notation

The results are presented in a functional language, similar to CIP-L [BBB+85]. Most of its constructs used in the sequel are self-explanatory. As in CIP-L, the semantics is strict and call-by-value. In this chapter, types of functions, arguments, etc., will be omitted when they are clear from the context. It is assumed that all expressions, functions, etc., are defined and deterministic [BBB+85] on their domains. Note that this implies in particular that functions terminate.

The functions considered here are of the form:

2.1 $f(x : Q(x)) =$ **if** $T(x)$
then $H(x)$
else $E(x, f(K_1(x)), \ldots, f(K_n(x)))$ **fi**.

The predicate $Q(x)$ restricts the domain of f to those elements that satisfy Q. $T(x)$, $H(x)$, and $K_i(x)$ (for $i \in [1..n]$) are expressions not defined in terms of f. In case $n = 1$, we simply write $K(x)$ instead of $K_1(x)$. The K_i functions will be referred to as *descent functions* (cf. [Coh83]). Because arguments may be tuples, and expressions may contain conditionals, this describes a large class of recursive functions. Also, many functions that use different control structures can be brought into the above form.

Example 2.1
As an example of a function of the form **2.1**, consider the well-known Fibonacci function, defined by:

2.2 $\mathit{fib}(x : x \geq 0) =$ **if** $x \leq 1$ **then** 1 **else** $\mathit{fib}(x-1) + \mathit{fib}(x-2)$ **fi**.

End of example 2.1

The notation $f^n(x)$, $n \geq 0$, denotes the n-fold application of f to x, i.e.:

$$\begin{aligned} f^0(x) &= x, \\ f^n(x) &= f(f^{n-1}(x)) \text{ for } n \geq 1. \end{aligned}$$

The function g^{-1} denotes the inverse of g, provided that it exists. $(g^{-1})^k(x)$ is abbreviated to $g^{-k}(x)$.

An important notion throughout this chapter is the dependency relation between function calls. We say that argument x depends on argument y for function f, denoted by $x \leftarrow_f y$, if

the value $f(y)$ is evaluated in order to determine the value of $f(x)$. Although this description suffices for our purposes, we give a formal definition in order to show that these dependency relations have an operational interpretation, and can therefore be included in the program text. The operators Δ and ∇ denote sequential conjunction and disjunction, respectively.

Definition 2.3 *For a function f defined by a scheme*

$$\begin{aligned} f(x : Q(x)) = \ & \textbf{if } T(x) \\ & \textbf{then } H(x) \\ & \textbf{else } E(x, f(K_1(x)), \ldots, f(K_n(x))) \textbf{ fi} \end{aligned}$$

the dependency relation $\leftarrow_f$ is defined by

$$x \leftarrow_f y \equiv Q(x) \wedge (x = y \,\nabla\, (\neg T(x) \,\Delta\, \exists i \in [1..n] : K_i(x) \leftarrow_f y)).$$

Lemma 2.4 *For any defined and deterministic function f of scheme* **2.1**, *the relation $\leftarrow_f$ constitutes a partial ordering on the set $\{x|Q(x)\}$.*

Proof. In order to prove that $\leftarrow_f$ is a partial ordering, we have to prove reflexivity, transitivity and antisymmetry of $\leftarrow_f$. From the verbal description, it is clear that these properties hold. A complete formal proof is given in [Boi89]. □

Often, computationally more efficient expressions for the dependency relation $\leftarrow_f$ can be derived. As an example, the following holds according to definition 2.3:

$$x \leftarrow_{fib} y \equiv x \geq 0 \wedge (x = y \,\nabla\, (x > 1 \,\Delta\, (x - 1 \leftarrow_{fib} y \,\nabla\, x - 2 \leftarrow_{fib} y))),$$

where *fib* is as defined above. It can even be simplified to the following non-recursive expression:

$$x \leftarrow_{fib} y \equiv 0 \leq y \leq x \wedge (x = 1 \Rightarrow y = 1).$$

2.4 Linear recursive functions

This section considers a particular subset of the functions of scheme **2.1**, viz. those where n, the number of recursive calls in the body, equals 1. We call these *linear recursive* functions (thus avoiding the need to refer to "comb-shaped reduction graphs" or "executing in linear time with respect to the magnitude of its argument" [Har88]).

Introducing some more terminology, a function of scheme **2.1** is said to be *tail-recursive* (or *iterative*) when $E(x, y) \equiv y$. The notion of tail recursion is important because the recursive function

2.5 $f(x) =$ **if** $T(x)$ **then** $H(x)$ **else** $f(K(x))$ **fi**

can be considered as "just a different notation" for the loop

2.6 **while not** $T(x)$ **do** $x := K(x)$ **od**;
return $H(x)$,

thus providing the link with the imperative language level.

Following a longstanding tradition, we use as an example the (obviously linear recursive) factorial function, defined here by:

2.7 $fact(x : x \geq 0) =$ **if** $x = 0$ **then** 1 **else** $x \times fact(x-1)$ **fi**.

The factorial function was the starting point for Cooper's pioneering work on changing the recursion structure of linear recursive functions [Coo66]. He presented several iterative versions of the factorial, exploiting algebraic properties of the operators involved (commutativity and associativity of multiplication, invertibility of addition). From these, general transformation rules were derived. Many similar rules have since been presented, most notably by Bauer and Wössner [BW82] (whose treatment of linear recursion strongly influenced ours), and by Arsac and Kodratoff [AK82].[1]

We will start by presenting a transformation rule that inverts the order of evaluation of the factorial function. By gradually relaxing the applicability conditions, a more general rule is obtained that is applicable to a large class of functions.

As was mentioned in the introduction, it is our intention to find a function that uses *the same* arguments in an inverted order. This excludes Cooper's rules, since those of his rules that may be said to invert the order of evaluation also add arguments that accumulate the function result. In general, this is useful, because this usually results in tail recursion. Here, however, we wish to maintain the distinction between the accumulation and inversion techniques, in order to study them separately.

In order to informally motivate the transformation rule to be presented, consider an inverted computation of the factorial function *fact* for argument x. First, it is clear that it should start with argument 0, since this is the last argument in any evaluation of *fact*. If x is also 0, we are ready and should return 1. Generally, let us take as an invariant that all factors up until the current argument have already been incorporated into the result. So, if x is not 0, we have to incorporate the next factor – that should be the next argument, since the current one had already been incorporated. Thus we have

2.8
$$\begin{aligned} fact(x : x \geq 0) &= fact'(0) \\ &\quad \textbf{where} \\ fact'(y : y \geq 0) &= \textbf{if } y = x \\ &\quad \textbf{then } 1 \\ &\quad \textbf{else } (y+1) \times fact'(y+1) \textbf{ fi}. \end{aligned}$$

In order to obtain a general transformation rule from this, the assumptions used in the reasoning above have to be made explicit. These are the following:

[1] Note, however, that many of the transformation rules in the literature that use the factorial function as their prime example are *not* concerned with inverting the order of evaluation, but with *accumulation* (i.e., adding a parameter that accumulates the function result, exploiting associativity of E in scheme **2.1**) [Bir84, Boi91b].

- every computation of $fact(x)$ ends with the computation of $fact(0)$, which can be expressed using the function call dependency relation $\leftarrow_{fact}$ as

$$T(0) \wedge \forall\, x : x \leftarrow_{fact} 0;$$

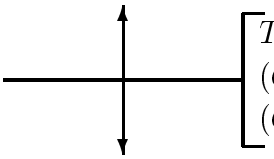
- it is immediately clear which is the "next" argument in the inverted evaluation, i.e. K is invertible;

- it does not matter in which order the "factors" are "incorporated", i.e. the binary operation E is (left-)commutative.

Consequently, we have the following rule. Note that the conditions have been restricted to arguments in the domain of the function.

Transformation 2.9

$f(x : Q(x)) =$ **if** $T(x)$
then $H(x)$
else $E(x, f(K(x)))$ **fi**



$$\begin{array}{l} T(c) \wedge \forall\, x : Q(x) \Rightarrow x \leftarrow_f c \\ (Q(x) \wedge \neg T(x)) \Rightarrow K^{-1}(K(x)) = x \\ (Q(x) \wedge Q(y)) \Rightarrow E(x, E(y, z)) = E(y, E(x, z)) \end{array}$$

$f(x : Q(x)) = f'(c)$
where
$f'(y : Q(y)) =$ **if** $y = x$
then $H(c)$
else $E(K^{-1}(y), f'(K^{-1}(y)))$ **fi**

Proof of Transformation 2.9
Rename f in the input scheme to f_1, in the output scheme to f_2. Let

$$m_x = \textbf{min}\ m' : c = K^{m'}(x).$$

The existence of m_x is guaranteed due to definedness of f. From the condition on c, it follows that m_x is the number of recursive calls to f_1 in the evaluation of $f_1(x)$. The subscript in m_x will be omitted when x is clear from the context.

Lemma 2.10 *If* $\neg T(x)$, *then*

$$f_2(x) = E(K^{m-1}(x), E(K^{m-2}(x), \ldots, E(x, H(c)) \ldots)).$$

Proof of Lemma 2.10 By $m-1$ times unfolding f', using invertibility of K. □

We now prove the transformation rule by induction on m_x.

Basis: $m_x = 0 \Rightarrow c = x$, thus $f_1(x) = H(x)$ and $f_2(x) = H(c) = H(x)$. □

Induction step: Let $m_x = a + 1$, then

$f_1(x)$
$\quad = \{$ unfold f_1 $\}$
$E(x, f_1(K(x)))$
$\quad = \{$ induction hypothesis $\}$
$E(x, f_2(K(x)))$
$\quad = \{$ lemma 2.10 $\}$
$E(x, E(K^a(x), E(K^{a-1}(x), \ldots, E(K(x), H(c))\ldots)))$
$\quad = \{$ left-commutativity of E, a times $\}$
$E(K^a(x), E(K^{a-1}(x), \ldots, E(K(x), E(x, H(c)))\ldots))$
$\quad = \{$ lemma 2.10 $\}$
$f_2(x)$ □

End of proof.

By relaxing the conditions on the above rule, more general transformation rules are obtained. This already follows from the proof above: apart from left-commutativity, only the existence of m_x and the validity of Lemma 2.10 are used.

A trivial generalization is the one where c is determined in the transformed program instead of in the applicability conditions. This can be seen in the proof above, where we could also have used $m_x = \mathbf{min}\ m' : T(K^{m'}(x))$, thus eliminating the need for a globally defined c. Instead, the following definition can be added to the **where**-clause:

$$c = \ \mathbf{that}\ c' : T(c') \wedge x \leftarrow_f c'.$$

The rule thus obtained is applicable to the function *facthalf*, defined by

2.11 $fact(x) = \ \mathbf{if}\ x \geq 1\ \mathbf{then}\ facthalf(x) \times facthalf(x-1)\ \mathbf{else}\ 1\ \mathbf{fi}$
where
$facthalf(x : x \geq 0) = \ \mathbf{if}\ x \leq 1\ \mathbf{then}\ 1\ \mathbf{else}\ x \times facthalf(x-2)\ \mathbf{fi}$

whereas Transformation 2.9 is not. The last argument in the evaluation of $facthalf(x)$ is 0 or 1, if x is even or odd, respectively. So, one c such that the evaluation always ends with $facthalf(c)$ does not exist – if x is known, however, the correct value for c can be chosen. This may seem an artificial example; it plays a crucial role, however, in the derivation of a new factorial algorithm in chapter 3.

Another generalization is concerned with the inverse of K. In short, it is not necessary for K to have an inverse - in concrete cases, a *generalized* inverse can always be found.

The existence of an inverse K^{-1} was required because the value z such that $K(z) = y$ needs to be determined. In general, there may be multiple values z such that $K(z) = y$. In the computation of $f(x)$ for one particular x, however, only one of those values actually occurs as an argument of f (otherwise, from determinacy of K it follows that f may not terminate for some arguments, and thus be undefined). This already yields a descriptive specification of the generalized inverse, which we also call K^{-1} (it has the x mentioned before as an extra parameter):

2.12 $K^{-1}(x, y : \exists\, j \geq 1 : y = K^j(x)) =$ **that** $z : (\neg T(z) \wedge K(z) = y \wedge x \leftarrow_f z)$.

In concrete cases, efficient operational versions of K^{-1} can be derived, in particular if $x \leftarrow_f z$ can be expressed non-recursively. Even though **that**-expressions are not operational, K^{-1} can always be computed, as was shown by Paterson and Hewitt [PH70]. They determine $K^{-1}(x, y)$ by computing all arguments $K^i(x)$ for $i = 1 \ldots j$ such that $K^j(x) = y$. Such a j always exists, because y occurs as an argument to f in the evaluation of $f(x)$. The argument that precedes the argument y in the computation of $f(x)$ is $K^{j-1}(x)$. Thus we have

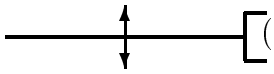
$$\begin{aligned} K^{-1}_{PatHew}(x, y : \exists\, j \geq 1 : y = K^j(x)) \quad = \quad & \textbf{if } y = K(x) \\ & \textbf{then } x \\ & \textbf{else } K^{-1}_{PatHew}(K(x), y) \textbf{ fi}. \end{aligned}$$

Note that the computation of K^{-1}_{PatHew} may be relatively expensive.

Altogether we have the following rule.

Transformation 2.13

$$\begin{aligned} f(x : Q(x)) = \; & \textbf{if } T(x) \\ & \textbf{then } H(x) \\ & \textbf{else } E(x, f(K(x))) \textbf{ fi} \end{aligned}$$



$$(Q(x) \wedge Q(y)) \Rightarrow E(x, E(y, z)) = E(y, E(x, z))$$

$$\begin{aligned} f(x : Q(x)) = \; & f'(c) \\ & \textbf{where} \\ & f'(y : Q(y)) = \textbf{if } y = x \\ & \qquad\qquad \textbf{then } H(c) \\ & \qquad\qquad \textbf{else } E(K^{-1}(x, y), f'(K^{-1}(x, y))) \textbf{ fi}, \\ & c = \textbf{ that } c' : T(c') \wedge x \leftarrow_f c', \\ & K^{-1}(x, y) = \textbf{ that } z : \neg T(z) \wedge K(z) = y \wedge x \leftarrow_f z \end{aligned}$$

An example function that can be inverted using Transformation 2.13 is the function ff (cf. chapter 3), defined by:

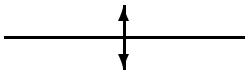
2.14 $ff(x) =$ **if** $x = 0$ **then** 1
else $facthalf(x - 1 + (x \textbf{ mod } 2)) \times 2^{x/2} \times ff(x/2)$.

Note that $K(x) = x/2$, and thus K does not have an inverse. In the derivation in chapter 3, an efficient definition for the *generalized inverse* $K^{-1}(x, y)$ is derived by embedding with the value n such that $y = x/2^n$. On the machine language level, K^{-1} can thus be implemented by gradually shifting a value into a register.

With these two generalizations, only left-commutativity of E is left as an applicability condition. An obvious generalization of Transformation 2.13 that discards left-commutativity of E does not exist. In order to present a rule without applicability conditions, we return to one of Cooper's rules, and generalize it with the steps that led from Transformation 2.9 to Transformation 2.13. The result of this reasoning is captured in the following rule, where inverting the order of evaluation is combined with result accumulation.

Transformation 2.15

$f(x : Q(x)) =$ **if** $T(x)$
then $H(x)$
else $E(x, f(K(x)))$ **fi**



$f(x : Q(x)) = f'(c, H(c))$
where
$f'(y, z : Q(y)) =$ **if** $y = x$
then z
else $f'(K^{-1}(x, y), E(K^{-1}(x, y), z))$ **fi**,
$c =$ **that** $c' : T(c') \wedge x \leftarrow_f c'$,
$K^{-1}(x, y) =$ **that** $z : \neg T(z) \wedge K(z) = y \wedge x \leftarrow_f z$

For more recursion structure changing transformations for linear recursive functions, the reader is referred to e.g. [BW82].

2.5 Stacks for linear recursion

Often, in transformational developments, it is not immediately clear how more operational improvement can be obtained. In those cases, it can be useful to consider the program from a different perspective by explicitly representing information that is only implicitly present in the program. An example of this is given by Wand [Wan80], who describes optimizations obtained by first introducing continuations, and then replacing them by efficient representations.

In this section, we make explicit the usual *implementation of recursion by stacks*. First, we present a transformation rule that introduces stacks. Then, the results of the preceding section are considered from that viewpoint. Finally, we prove the transformation rule.

2.5.1 Introducing stacks

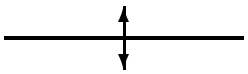
Informally, the transformation rule below works as follows. First, a stack of all arguments that occur in the evaluation of $f(x)$ is constructed. Then, f is applied to all values in that stack, resulting in a stack of results. The top of this stack corresponds to $f(x)$, since x is the first element to be put on the argument stack, and thus $f(x)$ appears last on the result stack.

We assume we have a data type $stack(\mathbf{M})$ for any type $\mathbf{M}$ with associated sort **Mstack** and the usual operations *empty*, *push*, *pop* and *top*. The empty stack is denoted by **empty**. Then we can express the following transformation rule.

Transformation 2.16

(For clarity, complete type information is given here).

$f(\mathbf{m}\ x : Q(x))\mathbf{n} = \mathbf{if}\ T(x)$
$\quad \mathbf{then}\ H(x)$
$\quad \mathbf{else}\ E(x, f(K(x)))\ \mathbf{fi}$



$f(\mathbf{m}\ x : Q(x))\mathbf{n}$
$= top(fres(fargs(x, \mathbf{empty}), \mathbf{empty}))$
$\quad \mathbf{where}$
$\quad fargs(\mathbf{m}\ x, \mathbf{mstack}\ s)\mathbf{mstack}$
$\quad = \mathbf{if}\ T(x)$
$\quad\quad \mathbf{then}\ push(x, s)$
$\quad\quad \mathbf{else}\ fargs(K(x), push(x, s))\ \mathbf{fi},$
$\quad fres(\mathbf{mstack}\ args, \mathbf{nstack}\ res)\mathbf{nstack}$
$\quad = \mathbf{if}\ empty(args)$
$\quad\quad \mathbf{then}\ res$
$\quad\quad \mathbf{else}\ fres(pop(args), push(newres, res))$
$\quad\quad \mathbf{where}\ \mathbf{n}\ newres = \mathbf{if}\ T(top(args))$
$\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{then}\ H(top(args))$
$\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{else}\ E(top(args), top(res))$
$\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{fi}$
$\quad\quad \mathbf{fi}$

The reader who is convinced of the correctness of this rule can safely skip the proof of this rule in section 2.5.3.

2.5.2 The stacks viewpoint of inversion

There are a number of points to be made about the above transformation.

First, two simplifications are possible, but omitted for presentation purposes. A trivial one is that it is not necessary for the intermediate results to be put on a stack – the result stack only grows, and only the top element is ever accessed. Thus a single variable suffices. The transformation is given *with* a stack, however, to stress the similarities between the treatment of arguments and results, and to facilitate the generalization to tree-like recursive functions in section 2.7. Another optimization is that the test $T(top(args))$ will return **true** exactly once, viz. for the first argument. Thus, by unfolding *fres* and appropriate simplifications this test can be eliminated.

In a sense, Transformation 2.16 makes explicit the traditional implementation of recursion via stacks. In "real" implementations, however, usually the two stacks are combined.

More importantly, there is a strong relation with the techniques described in section 2.4. Bauer and Wössner [BW82] motivate the introduction of stacks as a tabulation of Paterson and Hewitt's K^{-1} function. Indeed, in the special case that K is an invertible function, the argument stack in Transformation 2.16 can be eliminated, since the value directly below the top can be obtained by applying K^{-1} to the current top. In Transformations 2.9 and 2.13, the condition on E implies that the function computes the same result for an inverted stack. The value c in those transformations is the top of the argument stack. In Transformation 2.9, it is fixed for all arguments, whereas in Transformation 2.13 it can be directly determined from the argument to the function f.

There is yet another view of stacks in this context. Bauer and Wössner [BW82] present a slightly different inversion rule using stacks. That rule can be transformationally derived by introducing an argument stack as an extra parameter. This implies that an inverse for the new K exists, using $pop(push(x, s)) = x$. By applying inversion to that function, a general inversion rule using stacks is obtained.

Technically, it is also possible to prove Transformation 2.16 along similar lines, using a series of embeddings and Transformation 2.15. The proof presented in the next section, although less mechanical, provides more insight in the transformation rule, however.

2.5.3 Proof of Transformation 2.16

The proof proceeds as follows. Since we want to prove a lemma (viz. the definition of f in terms of *fres* and *fargs* in Lemma 2.19) that cannot be easily proved from these functions as defined in Transformation 2.16, new definitions for *fres* and *fargs* are given. From these new definitions, the lemmas are proved. Then, the new versions of *fres* and *fargs* are shown to be equivalent to those in Transformation 2.16 by simple unfold-fold derivations. Finally, the output scheme of Transformation 2.16 is derived using the lemmas.

Note that stacks are defined by the following **mode**-declaration [Par90]:

$$\textbf{mode mstack} \;=\; \textbf{empty} \mid push(\textbf{m}\ top, \textbf{mstack}\ pop).$$

This definition implies the usual axioms for stacks, viz.

$$top(push(x, s)) \;=\; x,$$

$$pop(push(x, s)) \quad = \quad s.$$

A number of functions on stacks are defined. For all **m** x, y, **mstack** s, t, (**m**)**n** f:

$$\begin{aligned}
empty(\mathbf{empty}) &= \mathbf{true} \\
empty(push(x, s)) &= \mathbf{false} \\
push(x, s){+\!\!+}t &= push(x, s{+\!\!+}t) \\
\mathbf{empty}{+\!\!+}t &= t \\
map(f, \mathbf{empty}) &= \mathbf{empty} \\
map(f, push(x, s)) &= push(f(x), map(f, s)) \\
rev(\mathbf{empty}) &= \mathbf{empty} \\
rev(push(x, s)) &= rev(x){+\!\!+}push(x, \mathbf{empty}) \\
length(\mathbf{empty}) &= 0 \\
length(push(x, s)) &= length(s) + 1
\end{aligned}$$

The definitions of the auxiliary functions *fres* and *fargs* in Transformation 2.16 can be viewed as "optimized" versions. In order to facilitate the proofs of the theorems below, the following definitions of these functions are assumed. These versions will be shown to be equivalent to those in Transformation 2.16.

2.17 *fres*(**mstack** *args*, **nstack** *res*)**nstack**
$= rev(map(f, args)){+\!\!+}res$
fargs(**m** x, **mstack** s)**mstack**
$=$ **if** $T(x)$ **then** $push(x, \mathbf{empty})$
else $fargs(K(x), \mathbf{empty}){+\!\!+}push(x, \mathbf{empty})$ **fi**$+\!\!+s$

Lemma 2.18 *Let* $s = fargs(x, \mathbf{empty})$. *Then* $\forall\, n : 1 \leq n \leq length(s)$*:*

$$top(pop^{n-1}(s)) = K(top(pop^{n}(s))).$$

Proof of Lemma 2.18. This can be proved by induction on $length(s)$. □

Lemma 2.19 *For all* x *in the domain of* f*, we have*

$$f(x) = top(fres(fargs(x, \mathbf{empty}), \mathbf{empty})).$$

Proof of Lemma 2.19. This follows from the definitions of *fres* and *fargs*, in particular the fact that $top(rev(fargs(x, \mathbf{empty}))) = x$. □

Now a recursive definition for *fres* is calculated:

fres(**mstack** *args*, **nstack** *res*)**nstack**
$= rev(map(f, args)){+\!\!+}res$
$=$ { case introduction: $empty(args)$ }

if $empty(args)$
then $rev(map(f, args)){+\!\!+}res$
else $rev(map(f, args)){+\!\!+}res$ **fi**

$=$ { instantiate $args$, map, rev, $+\!\!+$ in branches }

if $empty(args)$ **then** res
else $(rev(map(f, pop(args))){+\!\!+}push(f(top(args)), \mathbf{empty})){+\!\!+}res$ **fi**

$=$ { associativity $+\!\!+$, definition $+\!\!+$ }

if $empty(args)$ **then** res
else $rev(map(f, pop(args))){+\!\!+}push(f(top(args)), res)$**fi**

$=$ { fold $fres$, unfold f }

if $empty(args)$ **then** res
else $fres(pop(args), push(newres, res))$
where $x = top(args)$,
$newres =$ **if** $T(x)$ **then** $H(x)$ **else** $E(x, f(K(x)))$ **fi**
fi,

and also for $fargs$:

$fargs(\mathbf{m}\ x, \mathbf{mstack}\ s)\mathbf{mstack}$
$=$**if** $T(x)$ **then** $push(x, \mathbf{empty})$
else $fargs(K(x), \mathbf{empty}){+\!\!+}push(x, \mathbf{empty})$**fi**$+\!\!+s$

$=$ { distribution of conditional over $+\!\!+$ }

if $T(x)$ **then** $push(x, \mathbf{empty}){+\!\!+}s$
else $(fargs(K(x), \mathbf{empty}){+\!\!+}push(x, \mathbf{empty})){+\!\!+}s$ **fi**

$=$ { associativity $+\!\!+$, definition $+\!\!+$ }

if $T(x)$ **then** $push(x, s)$
else $fargs(K(x), \mathbf{empty}){+\!\!+}push(x, s)$ **fi**

$=$ { unfold $fargs$, distributivity, properties $+\!\!+$, fold $fargs$ }

if $T(x)$ **then** $push(x, s)$
else $fargs(K(x), push(x, s))$ **fi**.

In the final step, $fres$ is optimized by replacing the value of the recursive call to f by the top of the result stack so far. Technically, this is done as follows. A function $fres'$ is defined to be equal to $fres$, and is augmented with the assertion:

$$\neg empty(res) \Rightarrow top(res) = f(K(top(args))).$$

The initial call to $fres$ (with $res = \mathbf{empty}$) trivially respects this property. Using Lemma 2.18, it can be proved that the recursive call in $fres$ maintains the assertion. This proves that $fres'$ is equivalent to $fres$ (in context), and thus the assertion may be added to $fres$. This allows optimization of $fres$ by replacing $f(K(x))$ by $top(res)$. Unfolding the definition of x in $fres$ then yields the output scheme of Transformation 2.16, and thus the validity of the rule is proved. □

2.6 More general storage of results

The previous section showed how inversion techniques could be explained in terms of the implementation of linear recursion by stacks. The invertibility of stacks proved to be an important prerequisite for this.

Although tree-like recursion can also be implemented by stacks, the fact that a temporal ordering must be imposed on the evaluation of recursive calls makes this conceptually less interesting. Also, devising protocols to determine where the results of recursive calls can be found on the stack is not a trivial task. (This so-called *offset problem* is even more apparent in the case of attribute evaluation during parsing, where the parse tree is not actually built but all information is stored on the stack [JM80, odAMT90]).

What is also important is that the natural data structure for tree-like recursion, i.e. the tree, is not invertible - inverting a tree results in a *cactus stack*, i.e. a data structure with multiple entry points, as used e.g. in the implementation of backtracking.

Another important point is that, in argument *stacks*, every argument occurs only once (due to definedness of the function), whereas in argument *trees* values may occur more than once. On an abstract level, much of the gain of inverting the order of evaluation of tree-like recursive functions comes from compressing argument trees into argument DAG's (directed acyclic graphs) [Coh83].

For these reasons, stacks will not be used in the subsequent sections. Function calls will be evaluated during a linear traversal of the argument DAG, since we are mainly concerned with sequential evaluation. The results of recursive calls will be stored using a form of *dynamic binding*, to be described below. Informally, the **remember** and **recall** constructs defined below do as they say: computed results are remembered, and if something has been computed and remembered, it can be recalled.

Thus, we introduce two kinds of auxiliary expressions, referred to in this section as "REM" and "REC":

$$\begin{aligned} \textsc{rem}(f, E_0, E_1, E_2) &: \textbf{remember } f(E_0) \textbf{ is } E_1 \textbf{ in } E_2 \textbf{ ni}, \\ \textsc{rec}(f, E) &: \textbf{recall } f(E). \end{aligned}$$

Here, E, E_0, E_1, and E_2 denote arbitrary expressions of the appropriate types; f denotes an arbitrary identifier.

A formal semantics for these constructs could be given in the style of the CIP language definition [BBB+85], or similarly to the semantics of memoization given by Pettorossi [Pet84a]. A later paper [Boi92] will deal with these issues; for the presentation of our results the following description should suffice.

Since REM and REC can be viewed as expressions with *side-effects*, the easiest way of understanding these constructs is an *imperative* one. Assume, for function f, the existence of a *global* array $store_f$.

The expression $\textsc{rem}(f, E_0, E_1, E_2)$ denotes the value of E_2, evaluated in a context where $\textsc{rec}(f, E)$ denotes the value of E_1 whenever E and E_0 are equivalent. So the sequential interpretation of $\textsc{rem}(f, E_0, E_1, E_2)$ is

$$store_f[E_0] := E_1;\ E_2,$$

and the interpretation of REC(f, E) is

$$store_f[E].$$

Note that this is an *interpretation* of REM and REC, not a *semantics*. E.g. the semantics of REC-expressions with possible scope conflicts, like

remember $f(1)$ **is** 1 **in** **remember** $f(1)$ **is** 2 **in recall** $f(1)$ **ni ni**,

is not determined (the above description favours the innermost REM). Also, REC(f, E) is left undefined for contexts where $f(E)$ has not been remembered. This is because such (incorrect) expressions will not be introduced by transformations, and every choice for resolving this conflict would limit the possibilities of implementation.

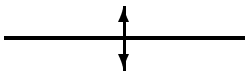
Generally speaking, REM and REC are very powerful language constructs that, when used injudiciously, can lead to incomprehensible "spaghetti" programs with very hard to find bugs. By only introducing these constructs via transformations, such problems are avoided. This shows another advantage of transformational programming: constructs so powerful that errors in using them are very likely can be introduced transformationally in such a way that their usage is guaranteed to be correct.

Note that our emphasis is not on extending the language. The expressions REM and REC above can be viewed as abbreviations for a large class of constructs that respect their semantics, such as global or local tables, associative memory, environments, memo functions [Mic67, Kho90], or even, in some cases, variables.

As an example, a general inversion rule for linear recursive functions that uses REM and REC is given below.

Transformation 2.20

$f(x : Q(x)) =$ **if** $T(x)$
then $H(x)$
else $E(x, f(K(x)))$ **fi**



$f(x : Q(x)) =$ **remember** $f(c)$ **is** $H(c)$
in $f'(c)$ **ni**

where

$f'(y : Q(y) \wedge x \leftarrow_f y)$
$=$ **if** $y = x$
then recall $f(x)$
else let $w = K^{-1}(x, y)$ **in**
remember $f(w)$
is $E(w,$ **recall** $f(y))$

in $f'(w)$ **ni ni**
fi,
$c =$ **that** $m : x \leftarrow_f m \wedge T(m)$,
$K^{-1}(x, y) =$ **if** $y = K(x)$ **then** x **else** $K^{-1}(K(x), y)$ **fi**

Mostly, remembered values will be used only some finite number of times. Therefore, a form of garbage collection would be appropriate. We do not introduce a language construct ("**forget**" ?) for that purpose. We will, however, whenever possible point out which values become obsolete and thus could be discarded. For the formal derivation of space-efficient implementations of REM/REC, the introduction of a predicate **known** $f(E)$, that holds whenever REC(f, E) is defined, seems useful. This will not be elaborated upon here, however, since this chapter is mostly concerned with the reduction of computation *time*.

In the above transformation, **remember/recall** can be straightforwardly implemented by argument stacks, or even by a single variable, in which case Transformation 2.15 results as a special case.

2.7 Tree-like recursive functions

In the case of tree-like recursion, again functions of scheme **2.1** are considered, with $n \geq 2$. A large class of functions can be brought into this form, using e.g. distributivity and other properties of conditionals. This may cause some inefficiency (e.g. duplication of tests), but many, if not all, of these inefficiencies can be removed again after applying inversion, using similar properties. An example of this is given in section 2.7.3.

Substantial work on improving tree-like recursive functions has been done by Cohen [Coh79, Coh83] and Harrison [Har88]. Cohen describes several classes of tree-like recursive functions for which redundant recursive calls can be eliminated by inverting the order of evaluation. In most cases, his transformations are only applicable if certain so-called *frontier conditions* are fulfilled, that state that computation of the function may in some cases safely proceed beyond the function domain. In our treatment of his classes of functions, such conditions do not need to be included because in our function scheme the domain is explicitly included via the definedness predicate Q.

The first case Cohen mentions is *explicit redundancy*, i.e. where some of the descent functions K_i are equal. This kind of redundancy can be eliminated using abstraction, and will thus not be further considered.

Cohen's second class is that of Fibonacci-like functions, called the class of *common generator redundancy* for reasons that will become clear in the treatment of this class in section 2.7.1.

A generalization of the common generator class is Cohen's third class of *commutative periodic redundancy*, described in section 2.7.2.

Cohen's fourth, even more general class, "*commutative redundancy*", will not be considered here for several reasons. First, Cohen's techniques are only obvious for functions with

2 recursive calls. Moreover, their gain in efficiency can just as easily be obtained by using more general techniques to be described in section 2.7.4.

Section 2.7.3 discusses a class of functions containing Dijkstra's obfuscating function [Dij76b].

Section 2.7.4 describes a new general *tabulation* rule, using an extension of the dependency relation $\leftarrow_f$ to a linear ordering to invert the order of evaluation.

For many of these classes of functions, similar optimizations can be obtained by first transforming the functions into linear recursive ones. The description of this *linearization* or *tupling* strategy in section 2.8 refers to many of these classes.

2.7.1 Common generators

Another well-known example function in transformational programming is the Fibonacci function, that was defined by

2.2 $fib(x : x \geq 0) =$ **if** $x \leq 1$ **then** 1 **else** $fib(x-1) + fib(x-2)$ **fi**.

It clearly exhibits redundant recursive calls – in fact, for $n > m > 1$, in the evaluation of $fib(n)$, $fib(m)$ is evaluated $fib(n-m)$ times. Most of the possible optimizations of the Fibonacci function depend on the trivial fact that $(x-1)-1 = x-2$, i.e. $K_1(x) = x-1$, $K_2(x) = x-2 = K_1^2(x)$. This property can be more abstractly described by saying that the descent functions have a *common generator*.

Definition 2.21 *A function g is a* common generator *for a function f of scheme* **2.1** *iff every descent function K_i equals a power of g, i.e.,*

$$\forall\, i : 1 \leq i \leq n : \exists\, m_i : K_i = g^{m_i}$$

Definition 2.22 *A function g is a* maximal *common generator for f, if g is a common generator for f with $K_i = g^{m_i} (1 \leq i \leq n)$, and the greatest common divisor (gcd) of $m_1, \ldots, m_n$ equals* 1.

Obviously, a maximal common generator exists whenever a common generator exists.[2]

Cohen first identified the class of functions with "common generator redundancy", and gave a transformation rule for them. Harrison [Har88] and Khoshnevisan [Kho90] developed an extensive theory of so-called *degenerate multilinear forms*, that essentially delivers the same results as Cohen's.

We present a rule more general than Cohen's. We do not impose his *frontier conditions*, and deal with explicit partiality of the function (by having the definedness predicate Q in the scheme).

The motivation for the rule is as follows. Consider an arbitrary argument to f, say y, in the computation of $f(x)$. This argument must be of the form $y = K_{i_1}(\ldots(K_{i_p}(x))\ldots)$ by

[2]Let g be a common generator with $K_i = g^{m_i}$ and $gcd(m_1, \ldots, m_n) = p$. Then $h = g^p$ constitutes a maximal common generator, with $K_i = h^{m_i/p}$.

definition of f. If f has a common generator g, y also equals $g^q(x)$, where $q = \sum_{i=1}^{n} \lambda_i m_i$ with $\lambda_i \geq 0$, $\sum_{i=1}^{n} \lambda_i = p$.

Since all values that occur as arguments are of this form, it is clear that $f(x)$ can be computed by computing $f(g(x))$, $f(g^2(x))$, etcetera, in an inverse order (using the generalized inverse of g if necessary). There are, however, two problems to be taken care of, viz. partiality of f, and the computation of f for superfluous arguments.

When $Q(g^b(x))$ does not hold for some b, then $f(g^b(x))$ is not defined and should thus not be computed.

Also, according to this scheme, in some cases $f(g^b(x))$ is computed for some b such that $x \leftarrow_f g^b(x)$ does not hold. Consider, e.g., a function f with $n = 2$, $K_1 = g^2$, $K_2 = g^3$; then $x \leftarrow_f g(x)$ does not hold. Since only a very limited number of such values exist (this is proved in [Coh79]), it is generally more efficient to also compute f for those values than to explicitly check whether it is really necessary, i.e. whether $b = \sum_{i=1}^{n} \lambda_i m_i$ with $\lambda_i \geq 0$. This idea is formalized in the definition of the *extended dependency relation* $\leftarrow_{f,g}$ defined below, where $x \leftarrow_{f,g} y$ holds whenever $y = g^k(x)$ for some k and it is not immediately clear that it is not necessary to compute $f(y)$ in the inverted computation of $f(x)$.

Definition 2.23 *If g is a common generator of a function f of scheme* **2.1**, *with $K_i = g^{m_i}(1 \leq i \leq n)$ and $max = max_{i \in [1..n]}(m_i)$, then*

$$x \leftarrow_{f,g} y = Q(x) \wedge Q(y) \wedge (x = y \nabla (\neg T(x) \Delta \exists\, k \in [1..max] : g^k(x) \leftarrow_{f,g} y)).$$

For the Fibonacci function *fib*, with $g(x) = x - 1$, $x \leftarrow_{fib} y \equiv x \leftarrow_{fib,g} y$ holds.

An abbreviating construct that is used in the transformation rule is the following:

$$\textbf{forall } x : P(x) \textbf{ remember } f(E_1(x)) \textbf{ is } E_2(x) \textbf{ in } E_3 \textbf{ ni}.$$

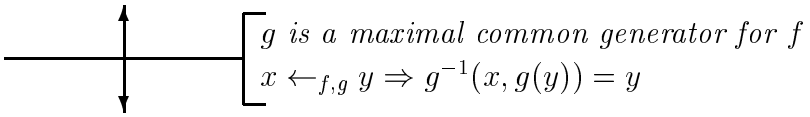
Here, P is a predicate that holds for a finite number of values x only. E_1, E_2, and E_3 are expressions of the appropriate types, of which only E_1 and E_2 may contain the free variable x; f is an arbitrary identifier. The semantics of this construct is given below.

$$\begin{array}{ll} & \textbf{forall } x : P(x) \textbf{ remember } f(E_1(x)) \textbf{ is } E_2(x) \textbf{ in } E_3 \textbf{ ni} \\ \equiv & \textbf{if } \exists\, x : P(x) \\ & \textbf{then remember } f(E_1(y)) \textbf{ is } E_2(y) \\ & \quad \textbf{in forall } x : P(x) \wedge x \neq y \textbf{ remember } f(E_1(x)) \textbf{ is } E_2(x) \textbf{ in } E_3 \textbf{ ni ni} \\ & \quad \textbf{where } y = \textbf{ some } z : P(z) \\ & \textbf{else } E_3 \\ & \textbf{fi} \end{array}$$

Now the transformation rule for common generator redundancy can be given. It results in a function that computes $f(x)$ by first remembering all necessary values for which T holds, and then proceeding towards x from one of those, using g^{-1}.

Transformation 2.24

$f(x : Q(x)) =$ **if** $T(x)$
then $H(x)$
else $E(x, f(K_1(x)), \ldots, f(K_n(x)))$ **fi**



$f(x : Q(x))$
$=$ **forall** $z : Tx(z)$
remember $f(z)$ **is** $H(z)$
in $f'(c)$ **ni**
where
$f'(y : x \leftarrow_{f,g} y)$
$=$ **if** $y = x$ **then recall** $f(y)$
else let $w = next(y)$ **in**
remember $f(w)$
is $E(w,$ **recall** $f(K_1(w)), \ldots,$ **recall** $f(K_n(w)))$
in $f'(w)$ **ni ni**
fi,
$next(y) =$ **if** $Q(z)$ **then** z **else** $next(z)$ **fi where** $z = g^{-1}(x, y)$,
$Tx(z) = T(z) \wedge x \leftarrow_{f,g} z$,
$c =$ **some** $z : Tx(z) \wedge \forall z' : (\neg T(z') \wedge x \leftarrow_{f,g} z' \Rightarrow z' \leftarrow_{f,g} z)$

In the above transformation, **remember**/**recall** can be implemented by $max_i(m_i)$ variables, that contain the values remembered last. This result, which is not very deep, will become more obvious when we consider linearization of common generator functions in section 2.8. Khoshnevisan [Kho90] presents an elaborate theory to derive this result.

Example 2.2
The application of Transformation 2.24 to the Fibonacci function yields

2.25 $fib(x : x \geq 0)$
$=$ **forall** $z : Tx(z)$
remember $fib(z)$ **is** $H(z)$
in $fib'(c)$ **ni**
where
$fib'(y : x \leftarrow_{fib,g} y)$
$=$ **if** $y = x$ **then recall** $fib(y)$
else let $w = next(y)$ **in**
remember $fib(w)$

is recall $fib(w-1) + $ **recall** $fib(w-2)$
in $fib'(w)$ **ni ni**
fi,
$next(y) =$ **if** $z \geq 0$ **then** z **else** $next(z)$ **fi where** $z = y+1$,
$Tx(z) = z \leq 1 \wedge x \leftarrow_{fib,g} z$,
$c =$ **some** $z : Tx(z) \wedge \forall z' : (\neg T(z') \wedge x \leftarrow_{fib,g} z' \Rightarrow z' \leftarrow_{fib,g} z)$.

This can be enormously simplified. Using

$$\begin{aligned} \leftarrow_{fib,g} &= \leftarrow_{fib} \\ next(y) &= y+1 \\ Tx(z) &= z = 1 \vee (z = 0 \wedge x \neq 0) \\ c &= \textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } 1 \textbf{ fi} \end{aligned}$$

and computing $fib(0)$ also for $x = 1$, we get:

2.26 $fib(x)$
$=$ **forall** $z : z \in \{0,1\}$
remember $fib(z)$ **is** 1
in $fib'(1)$ **ni**
where
$fib'(y : x \leftarrow_{fib} y)$
$=$ **if** $y \geq x$ **then recall** $fib(y)$
else remember $fib(y+1)$
is recall $fib(y) +$ **recall** $fib(y-1)$
in $fib'(y+1)$ **ni**
fi.

Note that a much simpler version of the transformation rule would also return this result. The full power of the rule is shown in [Boi89] with (contrived) example functions that violate Cohen's frontier conditions.
End of example 2.2

2.7.2 Commutative periodic redundancy

Whereas in the common generator redundancy class the descent functions have a greatest common divisor, for the class of functions with commutative periodic redundancy a *least common multiple* of the descent functions exists. I.e., there is a function K_* such that K_* is a power of each of the descent functions. Also, the descent functions are required to *commute*, i.e. $K_i(K_j(x)) = K_j(K_i(x))$ for relevant x and $1 \leq i, j \leq n$.

Definition 2.27 *A function K_* is a* common power *for a function f of scheme* **2.1** *iff*

$$\forall\, i \in [1..n] : \exists\, c_i : K_i^{c_i} = K_*.$$

Definition 2.28 *A function K_* is a* minimal *common power for a function f of scheme* **2.1** *if it is a common power for f, and no common power K_{**} for f exists such that $K_* = K_{**}^p$ for some $p \geq 2$.*

Obviously, if f has a maximal common generator g, with $K_i = g^{m_i}$, then f also has a minimal common power, viz. $g^{lcm(m_1,\ldots,m_n)}$ where *lcm* denotes the least common multiple function. Thus, this class is a generalization of the previous one. The generalization is strict, as shown by the following example.

Example 2.3
Consider the contrived example function *fibn* (which computes for any x, *fib*$(|x|)$), defined by:

2.29 *fibn*$(x) =$ **if** $|x| \leq 1$
then 1
else *fibn*$(sign(x) - x) +$ *fibn*$(2 \times sign(x) - x)$
fi
where $sign(x) =$ **if** $x = 0$ **then** 0 **elsf** $x > 0$ **then** 1 **else** -1 **fi**.

This function exhibits commutative periodic redundancy, since

- the descent functions commute:

$$\begin{aligned} sign(2 \times sign(x) - x) - (2 \times sign(x) - x) &= \\ 2 \times sign(sign(x) - x) - (sign(x) - x) &= x - 3 \times sign(x) \text{ for } |x| \geq 3, \end{aligned}$$

- there is a minimal common power of the descent functions, viz. $K_1^4 = K_2^2 = x - 4 \times sign(x)$.

The descent functions do *not* have a common generator, however.
End of example 2.3

The reasoning here proceeds similar to that for common generator redundancy. Arbitrary arguments for f can be reduced to a particular normal form, and the different normal form arguments suggest an order of computation for the inverted function.

Consider an arbitrary argument of f that occurs in the computation of $f(x)$ for some value x. By definition of f, this value equals (omitting brackets and the argument x, i.e. reasoning at the function level)

$$K_{a_1} \ldots K_{a_m} \text{ with } 1 \leq a_j \leq n, 1 \leq j \leq m.$$

The descent functions commute, and thus all applications of any one descent function can be combined, and the entire sequence $K_{a_1} \ldots K_{a_m}$ can be ordered, resulting in:

$$K_1^{p_1} \ldots K_n^{p_n}, \text{with } p_i = \#\{j \in [1..m] \mid a_j = i\}, 1 \leq i \leq n.$$

According to the division theorem, this equals (using c_i from Definition 2.27)

$$K_1^{q_1 \times c_1 + r_1} \ldots K_n^{q_n \times c_n + r_n} \text{ with } 0 \leq r_i < c_i, 1 \leq i \leq n.$$

Then the factors $K_*(= K_i^{c_i})$ can be "multiplied out", again using commutativity, resulting in

$$K_*^p K_1^{r_1} \ldots K_n^{r_n}, \text{ with } p = \sum_{i=1}^{n} q_i.$$

For convenience, let us denote the matrix resulting from elementwise application of a function h to a matrix X by $h(X)$. It can now be seen that the computation of $f(x)$ can be done by computing a series of matrices $f(S)$, $f(K_*(S)), \ldots, f(K_*^p(S))$, where S is a $c_1 \times \ldots \times c_n$ matrix, containing at every index $[m_1, \ldots, m_n]$ with $0 \leq m_i < c_i$, $1 \leq i \leq n$, the value $K_1^{m_1}(\ldots K_n^{m_n}(x)\ldots)$. The number p is the minimal number such that $K_*^p(S)$ contains no arguments that require recursive calls, i.e. for all y in $K_*^p(S)$, either $\neg Q(y)$ or $T(y)$ holds. Recall that f is only defined if Q holds, and thus $K_*^p(S)$ may contain many arguments for which f is not even defined.

Thus, it is not trivial to invert the order of evaluation of this class of functions. Cohen's analysis [Coh83] is restricted to functions with 2 recursive calls, and cannot be easily extended to n recursive calls. Moreover, in order to ensure definedness, very restrictive *frontier conditions* are imposed on the functions. The function *fibn* above, for example, does not fulfill these conditions.

In order to describe that part of a matrix for which f may be computed, we define a function *Matrix* that returns for any argument x to f the set of all values in the matrix S as above such that f is defined for those values.

$$\begin{aligned} \mathit{Matrix}(x) \quad = \quad & \{K_1^{r_1}(\ldots(K_n^{r_n}(x)\ldots) \mid \forall\, i \in [1..n] : r_i < c_i \wedge \\ & \forall r_i' \in [0\,..\,r_i - 1] : \neg T(K_1^{r_1}(\ldots(K_i^{r_i'}(\ldots(K_n^{r_n}(x))\ldots))\ldots))\} \end{aligned}$$

Now we can informally describe how to invert the order of evaluation of functions with commutative periodic redundancy:

2.30
$f(x) = f'(p)$
where
$f'(m)$
=**if** $m < 0$ **then recall** $f(x)$
else forall $y : y \in \mathit{Matrix}(K_*^m(x))$
remember $f(y)$ **is** ...
in $f'(m-1)$ **ni fi**,

where p is as defined above. The only thing left uncertain is from what to compute $f(y)$ (cf. the ... above). If it is done efficiently, the definition of f as in scheme **2.1** can be used with **recall**-expressions instead of recursive calls. I.e., the matrix should be traversed in an order that ensures that all recursive calls are evaluated before they are needed. Any traversal of the matrix from high index to low index fulfills this requirement.

For a further treatment of this class of functions, the reader is referred to section 2.8, where this class will be handled using linearization and an inverted computation of the function *fibn* will be derived.

2.7.3 Bounded disjoint generations

The analysis of the previous two sections was, more or less implicitly, based on the shape of the argument DAG's. In this section, we consider functions for which an upper bound for the number of nodes at any level in any argument DAG exists. Furthermore, in a given argument DAG each value is constrained to occur on at most one level. This is formalized in the notion of *bounded disjoint generations* (the word "generation" refers to the usual analogy between trees and family relationships – a generation consists of all nodes at a certain depth).

Definition 2.31 *For a function f of scheme* **2.1**, *the m-th generation of x in f, written as $(\Box_{i=1\ldots n}K_i)^m(x)$, is defined by:*

$$\begin{aligned}(\Box_{i=1\ldots n}K_i)^0(x) &= \{x\},\\ (\Box_{i=1\ldots n}K_i)^m(x) &= \{K_j(y) \mid \neg T(y)\,\Delta\, j \in [1\ldots n]\,\Delta\, y \in (\Box_{i=1\ldots n}K_i)^{m-1}(x)\}\\ &\quad \textit{for } m \geq 1.\end{aligned}$$

When n and K_i, $1 \leq i \leq n$, are clear from the context, $(\Box_{i=1..n}K_i)^m(x)$ is abbreviated to $\Box K^m(x)$.

The relation between the generations and the dependency relation $\leftarrow_f$ is as follows:

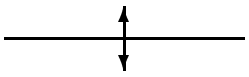
$$x \leftarrow_f y \equiv \exists\, k : y \in \Box K^k(x).$$

We can compute $f(x)$ by first computing f for all elements of the "last" generation, and then going back generation by generation, computing new f-values from those of the previously considered generation (and possibly discarding those afterwards) until the 0-th generation, i.e. x, is reached.

This is represented in the transformation rule below.

Transformation 2.32

$f(x : Q(x)) =$ **if** $T(x)$
then $H(x)$
else $E(x, f(K_1(x)), \ldots, f(K_n(x)))$ **fi**



$f(x : Q(x))$
$=$ **forall** $y : x \leftarrow_f y \wedge T(y)$
remember $f(y)$ **is** $H(y)$
in if $T(x)$ **then recall** $f(x)$
else $f'(t)$ **fi**
ni
where
$f'(m : 0 \leq m \leq t)$

$= \textbf{if } m = 0$
$\quad \textbf{then recall } f(x)$
$\quad \textbf{else forall } y : (y \in \Box K^{m-1}(x)) \wedge \neg T(y)$
$\quad\quad \textbf{remember } f(y) \textbf{ is } E(y, \textbf{recall } f(K_1(y)), \ldots, \textbf{recall } f(K_n(y)))$
$\quad\quad \textbf{in } f'(m-1) \textbf{ ni}$
$\quad \textbf{fi}$
$t = \textbf{ that } k : \Box K^k(x) \neq \emptyset \wedge \Box K^{k+1}(x) = \emptyset$

A couple of remarks about the transformation above are in order.

First, the transformation is applicable to all tree-like recursive functions; by taking n to be 1, a close variant of Transformation 2.20 is obtained, as is the case with most of the transformations in this chapter.

Most profit is obtained from this transformation, however, when the generations are *bounded* and *disjoint*. When the generations are *not* disjoint, one value may occur in several generations, in which case f is computed more than once for that value. Furthermore, when the generations are not bounded by a constant k, the **remember/recall** scheme in the transformation cannot be implemented by (k) variables.

Finally, the transformation rule explicitly mentions the generations, and by transition from the m-th to the $(m-1)$-th generation, a computation similar to that for K^{-1}_{PatHew} in section 2.4 is suggested. However, in concrete cases non-recursive expressions for $\Box K^m$ may exist, or functions that compute the previous generation (similarly to K^{-1} for linear recursion).

Example 2.4
As an example function that has bounded disjoint generations, consider the function *fusc*, commonly defined by:

2.33 $fusc(x : x \geq 1)$
$\quad = \textbf{if } x = 1 \textbf{ then } 1$
$\quad\quad \textbf{elsf } odd(x) \textbf{ then } fusc((x-1)/2) + fusc((x+1)/2)$
$\quad\quad \textbf{else } fusc(x/2) \textbf{ fi}.$

This function originates from mathematics [dR47], and was introduced into computing science by E.W. Dijkstra [Dij76b, Dij76c] as "a challenge for dr. R.M. Burstall". Since then, it has been one of the benchmarks of transformation techniques. It is an interesting example in this field, because it has a complex recursion structure (two branches with different numbers of recursive calls), and an efficient imperative solution for it exists.

In the form **2.33**, *fusc* does not fit scheme **2.1**, because of the two branches with recursive calls. By using distributivity and operators "ceil" and "floor", defined by:

$$\begin{aligned} \lceil \textbf{real } x \rceil &= \textbf{that int } y : 0 \leq y - x < 1 \\ \lfloor \textbf{real } x \rfloor &= \textbf{that int } y : 0 \leq x - y < 1, \end{aligned}$$

a definition that fits scheme **2.1** can be given:

2.34 $fusc(x : x \geq 1)$
$= \textbf{if } x = 1 \textbf{ then } 1$
$\quad \textbf{else if } odd(x) \textbf{ then } a$
$\quad\quad \textbf{else } a/2 \textbf{ fi}$
$\quad\quad \textbf{where } a = fusc(\lfloor x/2 \rfloor) + fusc(\lceil x/2 \rceil)$
$\quad \textbf{fi}.$

This is justified because $even(x) \Rightarrow \lfloor x/2 \rfloor = \lceil x/2 \rceil = x/2$.

Some particular properties hold for expressions containing ceil, floor and division by 2 (or, in fact, any other natural number):

$$\begin{array}{lcl} \lfloor \lfloor x \rfloor /2 \rfloor & = & \lfloor x/2 \rfloor \\ \lceil \lceil x \rceil /2 \rceil & = & \lceil x/2 \rceil \\ \lceil \lfloor x \rfloor /2 \rceil & \in & \{\lceil x/2 \rceil, \lfloor x/2 \rfloor\} \\ \lfloor \lceil x \rceil /2 \rfloor & \in & \{\lceil x/2 \rceil, \lfloor x/2 \rfloor\} \end{array}$$

Any argument value $y \in \Box K^k(x)$ is of the form:

$$\lambda_1(\lambda_2(\ldots(\lambda_k(x/2)/2)\ldots)/2) \text{ with } \lambda_i \in \{\lceil\ \rceil, \lfloor\ \rfloor\}.$$

Using the above properties, it follows that:

$$y = \lambda(x/2^k) \text{ with } \lambda \in \{\lceil\ \rceil, \lfloor\ \rfloor\}.$$

Furthermore, because *all* arguments y of the above form actually occur in the evaluation of $fusc(x)$,

2.35 $x > 2^{k-1} \Rightarrow \Box K^k(x) = \{\lfloor x/2^k \rfloor, \lceil x/2^k \rceil\}$.

This implies that $\Box K^k(x)$ contains exactly one element iff $x = a \times 2^k$ for some natural number a, otherwise it contains two elements. Thus we have shown that the generations are bounded for *fusc*. Not all generations are disjoint, however:

$$\begin{array}{lcl} \Box K^1(5) & = & \{2, 3\}, \\ \Box K^2(5) & = & \{1, 2\}. \end{array}$$

The only value that may occur in multiple generations is the value 2. Thus, we can overcome this problem by rewriting *fusc* (using a few trivial unfold- and case introduction steps) into:

2.36 $fusc(x : x \geq 1)$
$= \textbf{if } x \leq 3 \textbf{ then if } x = 3 \textbf{ then } 2 \textbf{ else } 1 \textbf{ fi}$
$\quad \textbf{else if } odd(x) \textbf{ then } a$
$\quad\quad \textbf{else } a/2 \textbf{ fi}$
$\quad\quad \textbf{where } a = fusc(\lfloor x/2 \rfloor) + fusc(\lceil x/2 \rceil)$
$\quad \textbf{fi}.$

In this version, $\Box K^1(5) = \{2,3\}$ but $\Box K^2(5) = \emptyset$.

Now we apply Transformation 2.32. According to **2.35**, the generations $\Box K^k(x)$ can be instantiated with $\{\lfloor x/2^k \rfloor, \lceil x/2^k \rceil\}$. This implies that for t (the number of the "last" generation), the value $\lceil {}^2\log x \rceil - 1$ can be substituted. Finally, we undo the first change of recursion structure in *fusc*. This results in:

2.37 *fusc*$(x : x \geq 1)$
= **forall** $y : (y \in \{1,2,3\}) \wedge x \leftarrow_f y$
remember *fusc*(y)
is if $y = 3$ **then** 2 **else** 1 **fi**
in if $x \leq 3$
then recall *fusc*(x)
else *fusc*$'(\lceil {}^2\log x \rceil - 1)$ **fi**
ni
where
fusc$'(m : 0 \leq m < \lceil {}^2\log x \rceil)$
= **if** $m = 0$
then recall *fusc*(x)
else forall $y : y \in \{\lceil x/2^{m-1} \rceil, \lfloor x/2^{m-1} \rfloor\} - \{2,3\}$
remember *fusc*(y)
is if *odd*(y)
then recall *fusc*$((y-1)/2)$ + **recall** *fusc*$((y+1)/2)$
else recall *fusc*$(y/2)$ **fi**
in *fusc*$'(m-1)$ **ni**
fi.

As suggested before, here **remember/recall** can be implemented using 2 variables. This results in the same program which would result from applying Transformation 2.15 to the well-known tail-recursive program for *fusc* [Dij76b, BW82].
End of example 2.4

Apart from illustrating Transformation 2.32, the above example also supports our claim that inefficiencies introduced in order to make functions fit scheme **2.1** can be fully eliminated after application of the transformation.

Another well-known example of a function with bounded disjoint generations is the Towers of Hanoi function discussed e.g. in [PP76].

2.7.4 Tabulation using compatible orderings

As mentioned before, our goal is to find, for tree-like recursive functions, a linear bottom-up traversal of the argument tree (or DAG). Of course, the most general way to do this is by explicitly giving the order in which the values in a DAG should be considered.

A true bottom up traversal implies that results of recursive calls are available when they are needed. This means that always $K_i(x)$ should be visited before x, i.e., the order of traversal should respect (i.e., extend) the partial ordering $\leftarrow_f$ (cf. Lemma 2.4).

A transformation based on such an ordering is given in [Par90], where it is called *tabulation*. The transformation we will consider here is more general. In section 2.4 it was shown how more powerful transformations could be given if, in the inverted computation, arguments were added that denoted the "final destination" (viz. the extra argument x in $K^{-1}(x,y)$). Here, the same tactic is employed; orderings are extended to ternary relations by adding an extra argument. The extra argument will be the second element of the triples of the relation, since it will usually be found as a superscript to the infix operator $\leq$.

Definition 2.38 *A ternary relation R is a* parameterized partial ordering in the second component (PO-2) *if each projection $\leq^y = \{(a,b) \mid (a,y,b) \in R\}$ to the first and third components constitutes a partial ordering.*

PO-2's will be denoted by the symbol $\leq$.

For a linear traversal of a DAG based on an ordering, it is necessary that a *successor function* can be defined for that ordering. A successor function is a function that returns the smallest element that is strictly greater than its argument. For a PO-2 $\leq$, the induced successor function $succ_\leq$, also decorated with an extra argument, is defined as follows:

$$succ_\leq(a, x : \exists y : x \neq y \wedge x \leq^a y) \equiv \textbf{that } y : \begin{array}{l} x \neq y \wedge x \leq^a y \\ \wedge \forall z : (x \leq^a z \Rightarrow y \leq^a z). \end{array}$$

Such a successor function is well-defined and can be profitably used for transformation if the PO-2 $\leq$ is of a particular nature, characterized by the predicate *Plinord*, defined by:

$$\begin{array}{rcl} Plinord(\leq) \equiv \forall\, a,x,y,z & : & (x \leq^a z \wedge y \leq^a z \Rightarrow x \leq^a y \vee y \leq^a x) \\ & \wedge & (x \leq^a y \wedge x \leq^a z \Rightarrow y \leq^a z \vee z \leq^a y) \\ & \wedge & \exists n : \#\{z' \mid x \leq^a z' \wedge z' \leq^a y\} \leq n, \end{array}$$

(informally, if two arguments both can be compared to a third one, they also can be compared to each other, and all intervals are finite), which implies that for each a a partition P of the domain of $\leq^a$ exists, such that $\leq^a$ is a total ordering on each member of P.

Given a PO-2 $\leq$ with its induced successor function, tabulation of $f(x)$ may proceed as follows. First compute f for the minimum of $\leq^x$ on $\{y \mid x \leftarrow_f y\}$. This minimum always exists, since the set is finite. Then, compute f for the successor of the current value, until x is reached.

For this to work, the ordering must respect $\leftarrow_f$. It is *not* necessary to compute f *only* for values y for which $x \leftarrow_f y$ holds. However, if f is computed for an unnecessary value x, the f-values for the recursive calls $f(K_i(x))$ must be available as well. This leads to the following definition of compatibility.

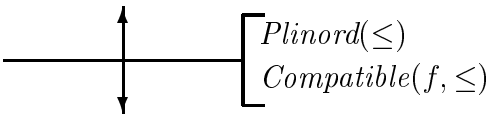
Definition 2.39 *The* PO-2 $\leq$ *is compatible with a function f of scheme* **2.1** *iff Compatible$(f, \leq)$ holds, where*

$$Compatible(f, \leq) = \forall\, x,y,z : (y \leq^x x \wedge y \leftarrow_f z) \Rightarrow z \leq^x y.$$

Then the transformation rule can be given.

Transformation 2.40

$f(x : Q(x)) =$ **if** $T(x)$
then $H(x)$
else $E(x, f(K_1(x)), \ldots, f(K_n(x)))$**fi**



$f(x : Q(x))$
$=$ **remember** $f(m)$ **is** $H(m)$
in $f'(m)$ **ni**
where
$f'(y : Q(x) \wedge y \leq^x x)$
$=$ **if** $y = x$
then recall $f(x)$
else let $z = succ_{\leq}(x, y)$ **in**
remember $f(z)$
is **if** $T(z)$
then $H(z)$
else $E(z,$ **recall** $f(K_1(z)), \ldots,$ **recall** $f(K_n(z)))$
fi
in $f'(z)$ **ni ni**
fi,
$m =$ **that** $m' : \forall p : (p \leq^x m') \Rightarrow p = m'$

It may not always seem obvious how to find a compatible ordering; however, for instance a linear ordering that has been used to prove termination of a function must be compatible.

Example 2.5 : Newton's binomial
The Newton binomial function, defined by

2.41 $bin(i, j : 0 \leq j \leq i) =$ **if** $j = 0 \vee i = j$
then 1
else $bin(i-1, j) + bin(i-1, j-1)$
fi,

is also a famous example of a function with a redundant evaluation. Using non-parameterized orderings, tabulation of the binomial can be done in at least two different ways, as shown in figure 2.1.

The "bars" tabulation uses as its ordering:

$$(i, j) \leq^{(p,q)} (k, l) \equiv (i = k \wedge j \leq l) \vee (i < k).$$

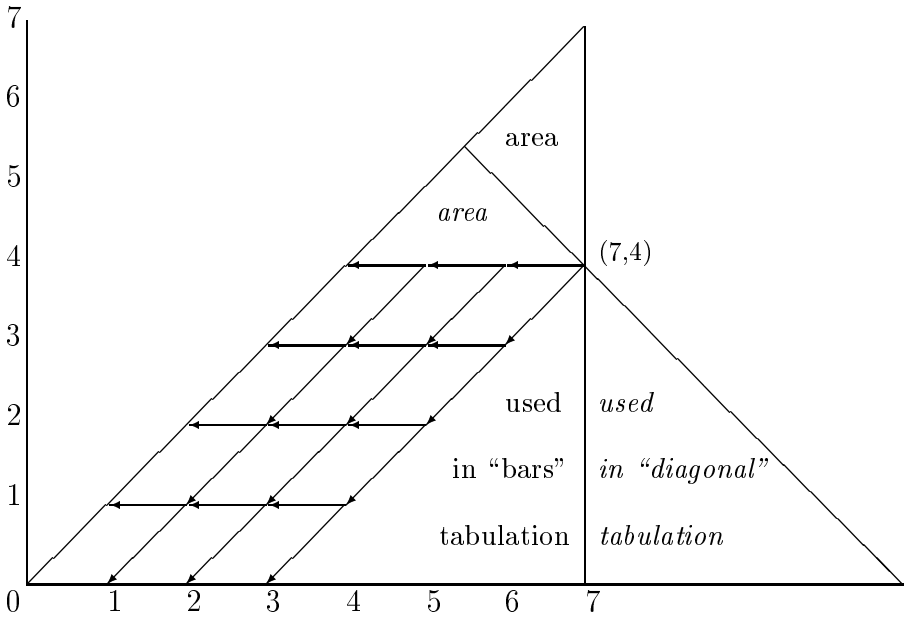


Figure 2.1: Dependencies and possibilities for tabulation for $bin(7,4)$

The inverted version, using this ordering, proceeds by calculating $bin(0,0)$, $bin(1,0)$, $bin(1,1)$, $bin(2,0)$, $bin(2,1)$, $bin(2,2)$, $\ldots$, $bin(p,1)$, $\ldots$, $bin(p,q)$. It calculates the values $bin(i,j)$ for each different value of i in turn. Thus, $p(p+1)/2+q+1$ function values are computed. For $(p,q)=(7,4)$, this amounts to 33.

The diagonal tabulation uses as its ordering:

$$(i,j) \leq^{(p,q)} (k,l) \equiv (i+j = k+l \wedge j \leq l) \vee (i+j < k+l).$$

It calculates the values $bin(i,j)$ for each different value of $i+j$ in turn, i.e. $bin(0,0)$, $bin(1,0)$, $bin(2,0)$, $bin(1,1)$, $bin(3,0)$, $bin(2,1)$, $\ldots$, $bin(p+q,0)$, $\ldots$, $bin(p+1,q-1)$, $bin(p,q)$. In this case the number of values calculated for the calculation of $bin(p,q)$ is $\lfloor (p+q+1)^2/4+q+1 \rfloor$. This amounts to 41 for $bin(7,4)$. If the recursive definition is used, 69 values (not all different, of course) have to be computed, so both tabulations *are* improvements.

One characteristic these two tabulations have in common is that they are based on orderings $\leq^{(p,q)}$ where the (p,q) are irrelevant. That is the reason why they are still inefficient: the number of values actually needed when $(p,q) \neq (0,0)$ is $(q+1)(p-q+1)-1$, viz. the number of points in the parallelogram in figure 2.1, with the exception of $(0,0)$. Only 19 values are necessary for $bin(7,4)$. We give an alternative ordering, which restricts the bars tabulation to the parallelogram of necessary values:

$$(i,j) \leq^{(p,q)} (k,l) \quad \equiv \quad ((i = k \wedge j \leq l) \vee (i < k)) \wedge \mathit{Need}(i,j) \wedge \mathit{Need}(k,l)$$

where

$$\mathit{Need}(a,b) \quad = \quad (0 \leq a - b \leq p - q) \wedge (0 \leq b \leq q).$$

The associated successor function s is given by:

$$\begin{aligned} s(p,q,i,j) \quad = \quad & \textbf{if } (i = j \vee j = q) \\ & \textbf{then } (i+1, \mathit{max}(1, i+1-p-q)) \\ & \textbf{else } (i, j+1) \textbf{ fi}. \end{aligned}$$

The reader is invited to try to visualize the route of the function through the parallelogram. This will clarify the above choices of ordering and successor functions; a lengthy proof could be given for the property $s = \mathit{succ}_{\leq}$, but it would only blur the issue.

We choose $(0,0)$ as a minimum for the above ordering (for any (p,q)).

Now we can transform *bin*, with instantiation of the minimum, into:

2.42
$\mathit{bin}(i, j : 0 \leq j \leq i)$
= **remember** $\mathit{bin}(0,0)$ **is** 1
in $\mathit{bin}'(0,0)$ **ni**
where
$\mathit{bin}'(a, b : 0 \leq b \leq a \wedge (a,b) \leq^{(i,j)} (i,j))$
= **if** $(a,b) = (i,j)$
then recall $\mathit{bin}(i,j)$
else let $(p,q) = s(i,j,a,b)$ **in**
remember $\mathit{bin}(p,q)$
is **if** $p = q \vee q = 0$
then 1
else recall $\mathit{bin}(p-1,q) +$ **recall** $\mathit{bin}(p-1,q-1)$
fi
in $\mathit{bin}'(p,q)$ **ni ni**
fi.

End of example 2.5

2.8 Linearization of tree-like recursive functions

The preceding sections showed how inversion of the order of evaluation could be achieved directly for several classes of tree-like recursive functions. There are two important reasons for considering the transition from tree-like recursion to linear recursion as well. First, *problem reduction* is an issue – the inversion techniques for linear recursive functions are simpler than those for tree-like recursion, and thus it may be profitable to invert the order of evaluation by first linearizing and then applying one of the rules from section 2.4. Furthermore, studies by Pettorossi [Pet84b] and Harrison [Har88] have shown that among the main example classes for which linearization is possible are those described in sections 2.7.1-2.7.3.

The *tupling* strategy was first described by Burstall and Darlington [BD77] and Pettorossi [Pet77]. In transformational developments in the unfold-fold style, progress is mostly made by definition of new functions (*eureka steps*, *embeddings*, *generalizations*), usually in terms of known ones. By unfolding the definition, rearranging expressions, and folding, an independent definition of the new function should be obtained. The crucial step in such a development is the choice of a new function to be defined. An important paradigm for defining new functions is *tupling*: define the value of a new function to be a tuple of applications of known functions.

Apart from recursion simplification, tupling is also used to combine applications of several functions that have very similar recursion structures (e.g. functions that visit the same data structure). Pettorossi [Pet84b] also describes tupling for systems of mutually recursive functions. By encoding the names of functions as an additional argument, systems of mutually recursive functions can be reduced to single recursive functions. Therefore, we do not consider systems of mutually recursive functions.

The tuple consisting of $a_1, \ldots, a_m$ is denoted by $[a_1, \ldots, a_m]$. The projection to the j-th component of a tuple a is denoted by $a.j$.

If tupling is used for linearization of a tree-like recursive function f, the embeddings are of the form

2.43 $F(x) = [f(x), f(h_1(x)), \ldots, f(h_m(x))]$.

When a definition for F independent of f, has been obtained, f can be defined in terms of F, viz.

2.44 $f(x) = F(x).1$.

For definedness of F, the following condition on the h functions should hold:

$$\forall\, i \in [1..m] : Q(x) \Rightarrow Q(h_i(x)).$$

This condition ensures that all components of the tuples are well defined. When this condition is not fulfilled for some values of x, case introductions may be added to the definition of f in terms of F (**2.44**) to ensure definedness of the tuple in all cases.

For the classes of functions defined in previous sections, embeddings that allow the derivation of linear recursive functions can be given directly. First, they are given for common generator redundancy. Then commutative periodic redundancy will be treated with an example. For bounded disjoint generations no general embedding can be given; we only note that for the function *fusc* an embedding

$$F(x) = [fusc(x), fusc(x+1)]$$

leads to the standard iterative program [Dij76b].

Common generator redundancy

Suppose f has a maximal common generator g, such that $K_i = g^{m_i} (1 \leq i \leq n)$ and $max = max_{i \in [1..n]}(m_i)$. Furthermore, suppose g has an inverse. Then the following embedding leads to a linear recursive version of f:

$$F(x) = [f(x), f(g^{-1}(x)), \ldots, f(g^{-(max-1)}(x))].$$

For each component of the tuple a variable can be used, and thus it is obvious that indeed max variables suffice for inverting functions with a common generator.

Depending on the number of times each of the recursive calls is unfolded, any function g^k with $k \geq 1$ can be chosen as the descent function for f. E.g., when $k = 4$ is chosen for the Fibonacci function *fib*, the derivation of a linear recursive version is completely mechanical. The embedding is:

2.45 $F(x) = [fib(x), fib(x+1)]$.

This expression has to be unfolded until it can be expressed in terms of $F(g^k(x))$, i.e. $[fib(x-4), fib(x-3)]$. Since $fib(x-4)$ is only defined for $x \geq 4$, case introductions have to be made for $x \in [0..3]$. The result of this derivation is:

2.46 $F(x) =$ **if** $x = 0$ **then** $[1, 1]$
elsf $x = 1$ **then** $[1, 2]$
elsf $x = 2$ **then** $[2, 3]$
elsf $x = 3$ **then** $[3, 5]$
else $[3a + 2b, 3a + 5b]$ **where** $[a, b] = F(x - 4)$
fi.

Commutative periodic redundancy

Suppose f has a least common power K_*, such that $K_* = K_i^{c_i} (1 \leq i \leq n)$. The analysis of this class in section 2.7.2 suggests an embedding where the h functions enumerate the $c_1 \times \ldots \times c_n$ matrix with at the index $[m_1, \ldots, m_n]$ the value $K_1^{m_1}(\ldots(K_n^{m_n}(x))\ldots)$. Cohen [Coh83] observed that for $n = 2$ it suffices to take only the column and the row with index 0 from the matrix, i.e. $c_1 + c_2$ elements. The further optimization suggested by Pettorossi [Pet84b] is based on the fact that a row and a column of a matrix have an element in common (i.e., only $c_1 + c_2 - 1$ values are needed). This optimization can be understood by considering minimal subsets H of such a matrix S that allow the expression of f for some value in S in terms of the value of f for values in H. Pettorossi's analysis of descent graphs gives a good explanation why this optimization works.

Extending this to the general n-dimensional case, we take all elements on the "edge" of the n-dimensional matrix, i.e. the embedding

$$F(x) = [h_1(x), \ldots, h_{\#H}(x)]$$

where the expressions $h_i(x)$ form an arbitrary enumeration of the set

$$H = \{f(K_1^{i_1}(\ldots(K_n^{i_n}(x))\ldots)) \mid (\forall j \in [1..n] : i_j < c_j) \wedge (\exists j \in [1..n] : i_j = 0)\}.$$

The first conjunct in the definition of H describes the set $Matrix(x)$, possibly extended to include values for which $f(x)$ is undefined; the second conjunct, by requiring one of the i_j to be 0, restricts this to the "edges" of the matrix.

Then K_* may be chosen as a descent function for g. Note, however, that since we do not require invertibility of the descent functions, the definition of f in terms of F may require some case introductions to ensure definedness of $h_i(x)$ for all i and x.

Example 2.6
The function *fibn* defined by:

2.47
$$\begin{aligned} fibn(x) = \ & \textbf{if } |x| \leq 1 \\ & \textbf{then } 1 \\ & \textbf{else } fibn(sign(x) - x) + fibn(2 \times sign(x) - x) \\ & \textbf{fi} \\ \textbf{where } sign(x) = \ & \textbf{if } x = 0 \textbf{ then } 0 \textbf{ elsf } x > 0 \textbf{ then } 1 \textbf{ else} - 1 \textbf{ fi}, \end{aligned}$$

was shown to exhibit commutative periodic redundancy in Example 2.3. This analysis, with $K_* = x - 4 \times sign(x)$, $c_1 = 4$, $c_2 = 2$, completely determines the following unfold-fold derivation leading to a linear recursive version of *fibn*.

The set H defined above is in this case

$$\{f(K_1^{i_1}(K_2^{i_2}(x))) \mid 0 \leq i_1 < 4 \wedge 0 \leq i_2 < 2 \wedge (i_1 = 0 \vee i_2 = 0)\}.$$

Thus, one of the possible embeddings is:

$$\begin{aligned} F(x : |x| \geq 3) &= [fibn(x), fibn(K_2(x)), fibn(K_1(x)), fibn(K_1^2(x)), fibn(K_1^3(x))] \\ &= [fibn(x), fibn(2s-x), fibn(s-x), fibn(x-2s), fibn(3s-x)] \\ &\quad \textbf{where } s = sign(x). \end{aligned}$$

Since $K_*(x) = x - 4 \times sign(x)$ is to be the descent function for f, calls to *fibn* in the definition of F must be unfolded until all elements of the tuple are expressed in terms of $F(K_*(x))$, i.e.

$$[fibn(x-4s), fibn(6s-x), fibn(5s-s), fibn(x-6s), fibn(7s-x)].$$

First, a number of case introductions (and instantiations) are necessary, since for folding F, $K_*(x) \geq 3$ should hold, i.e. $|x| \geq 7$. This yields:

$$\begin{aligned} F(x : |x| \geq 3) = \ & \textbf{if } |x| = 3 \textbf{ then } [3,1,2,1,1] \\ & \textbf{elsf } |x| = 4 \textbf{ then } [5,2,3,2,1] \\ & \textbf{elsf } |x| = 5 \textbf{ then } [8,3,5,3,2] \\ & \textbf{elsf } |x| = 6 \textbf{ then } [13,5,8,5,3] \\ & \textbf{else } [fibn(x), fibn(2s-x), fibn(s-x), fibn(x-2s), fibn(3s-x)] \\ & \quad \textbf{where } s = sign(x) \\ & \textbf{fi}. \end{aligned}$$

Unfolding the **else**-part until all elements are expressed in terms of $F(K_*(x))$ and subsequent folding yield:

$$\begin{aligned} & [2a+4b+2c+3d+4e, a+b+c+d+e, a+3b+c+2d+3e, a+2b+d+2e, a+b+e] \\ & \textbf{where } [a,b,c,d,e] = F(x - 4 * sign(x)). \end{aligned}$$

For $|x| < 3$, $F(x)$ is not defined, and thus some case introductions are necessary. Altogether, we then have:

2.48
$$
\begin{array}{ll}
\mathit{fibn}(x) = & \textbf{if } |x| \leq 1 \textbf{ then } 1 \\
& \textbf{elsf } |x| = 2 \textbf{ then } 2 \\
& \textbf{else } F(x).1 \textbf{ fi} \\
\textbf{where} & \\
F(x : |x| \geq 3) & \\
= & \textbf{if } |x| = 3 \textbf{ then } [3,1,2,1,1] \\
& \textbf{elsf } |x| = 4 \textbf{ then } [5,2,3,2,1] \\
& \textbf{elsf } |x| = 5 \textbf{ then } [8,3,5,3,2] \\
& \textbf{elsf } |x| = 6 \textbf{ then } [13,5,8,5,3] \\
& \textbf{else } [2a{+}4b{+}2c{+}3d{+}4e, a{+}b{+}c{+}d{+}e \\
& \quad , a{+}3b{+}c{+}2d{+}3e, a{+}2b{+}d{+}2e, a{+}b{+}e] \\
& \quad \textbf{where } [a,b,c,d,e] = F(x - 4 * \mathit{sign}(x)) \\
& \textbf{fi}.
\end{array}
$$

This example shows that the analysis for a class of functions may yield so much information that, for functions in that class, elaborate unfold-fold developments with complicated eureka's and numerous unfoldings can be more or less mechanically constructed.
End of example 2.6

Tupling in general

Cohen [Coh83] suggests constructing "a tree representing a typical value of $f(x)$" for analysis of redundant computations. Pettorossi [Pet84b] finds the motive for tupling in an analysis of so-called descent DAG's (called argument DAG's before in this chapter). Tupling, in that case, is applicable when a *progressive sequence* of equal sized *cuts* can be found. A cut is defined as a set of nodes which, when eliminated from a graph, turns it into a disconnected graph. Progressive sequences of cuts are sets of nodes, such that each "next" set contains some more nodes deeper in the tree and some less nodes nearer the root. The tuples to be defined should consist of all values in such a cut.

Indeed, drawing sample argument graphs may facilitate the understanding of what arguments occur in evaluations, and what their relationship is. This results in useful ideas for tupling embeddings, or other recursion simplifying transformations.

Another simple heuristic is also apparent from the general embeddings for common generator redundancy and commutative periodic redundancy. In general, finding the right embeddings requires ingenuity. In many cases, however, the right tupling embeddings can be found by *comparing* the descent functions. The embedding should be based on the *difference* between the descent functions. Furthermore, the descent function for the new function should be based on a *common part* of the descent functions.

For example, for common generator redundancy, the common generator forms both the difference and the similarity between the descent functions. Thus, the embedding and the descent function of the new function are based on powers of the common generator.

The embedding $[fusc(x), fusc(x+1)]$ can also be (somewhat informally) explained by this heuristic: the difference between $x/2$, $\lceil x/2 \rceil$, and $\lfloor x/2 \rfloor$ is indeed (at most) 1.

2.9 Discussion

2.9.1 Relevance of the results

The main purpose of this study was to give a synthetic view of techniques for inverting the order of evaluation. Thus, many techniques were presented that have been described in earlier papers. The main contributions of this chapter with respect to the known techniques, are the inclusion of descriptive (as opposed to operational) notation in the transformations, thus allowing subproblems to be considered separately, and the explicit treatment of partiality (using the predicate Q).

The most important new aspects are: the relation between inversion techniques for linear recursion, implementation via stacks, and techniques for tree-like recursion; the introduction of **remember/recall**; the systematic description of tupling for several classes of functions; and the general and efficient tabulation rule in section 2.7.4.

Many of the ideas on linear recursion can also be found in the textbooks by Bauer and Wössner [BW82] and Partsch [Par90], and in numerous papers on recursion removal (e.g. [AK82, HL78, PP86]). The explicit introduction of argument stacks was also mentioned by Bauer and Wössner. It is not so surprising that *result* stacks have not been discussed before, since they were included here mainly to facilitate the transition to tree-like recursion.

Although many papers have been published that describe Cooper's [Coo66], or similar, rules for linear recursive functions, and also a number of papers describe inversion techniques for tree-like recursive functions, these have not been brought together before. Considering argument and result stacks for linear recursion helps in understanding why complicated arguments are necessary for storing results and compressing DAG's in the case of tree-like recursion.

One of the most important contributions of this chapter is the introduction of **remember** and **recall**. These constructs have a simple intuitive meaning, and they are at a useful level of abstraction. On one hand, they can easily be integrated into a (quasi-)functional formalism, on the other hand they carry a hint of sequentiality. Most of the actual reasoning for tree-like recursive functions is (in this and other treatments) about the function *arguments*; using **remember/recall** yields the possibility of solving the problem of maintaining and retrieving the corresponding function *results* separately, after a change of recursion structure has taken place.

Comparing this with previous studies on tree-like recursion, Cohen [Coh83] uses an imperative language with recursion, and thus it is never completely obvious which of his large arrays of function results are still maintained (in some stack frame). One of the reasons that the analysis of common generator redundancy by Harrison [Har88] and Khoshnevisan [Kho90] is so complicated may be the fact that it has been done purely on the functional level (viz. in FP), whereas in the kinds of problems we have considered arguments (and *values*, in general) *are* important. This might also explain why Khoshnevisan [Kho90] needs such a

complicated proof to derive a fairly trivial result on the function *result* side. Likewise, the complicated functional "pointer structures" used to describe tabulation in [BGJ89] suggest that a purely functional notation is not suitable for describing tabulation and related techniques. Partsch [Par90] describes tabulation by introducing tables as additional arguments to recursive functions. Because tables have to be implemented in some way (it is certainly not efficient to copy table arguments for all recursive calls), this is not too different from the **remember/recall** approach. By including explicit tables, however, particular kinds of implementations become more obvious than others. E.g., implementations of tables by variables are difficult to derive.

A construct related to **remember/recall** is the *delay/force* construct in Scheme [CR89]. The *delay* function creates a continuation that can be evaluated on demand, i.e. using the *force* function. After it has been evaluated, the continuation is replaced by its value, and thus subsequent *force* demands do not need to evaluate it again. This mechanism is used for *lazy evaluation*. The main difference between *delay* and **remember** is that **remember** *does* compute the value. Another difference is that the continuation created by *delay* is a value that must be bound in order to be used, whereas **remember** only creates a dynamic context in which **recall** is well-defined.

The transformation rule given in this chapter for *common generator* redundancy is more generally applicable than those in [Coh83] and [Har88]. As is mostly the case in this chapter the gain in generality is due to explicit inclusion of partiality via the predicate Q, use of descriptive notation (e.g. **some**-expressions) and predicates (e.g. $\leftarrow_{f,g}$), and the use of **remember/recall** which allows to discard Cohen's frontier conditions.

Pettorossi [Pet84b] suggested to treat *commutative periodic redundancy* by tupling. We considered an example function to which tupling applies whereas Cohen's transformation does not (again, due to failed frontier conditions). The transition from 2 descent functions to $n \geq 2$ descent functions, which Cohen claims could be applied to his solutions, does in our case (contrary to Cohen's) not result in uglier notations.

The class of functions with *bounded disjoint generations* has, to our knowledge, not been described before. It is more general than the class of functions with *periodic redundancy* briefly mentioned by Cohen [Coh83].

The tabulation rule in section 2.7.4 is more general than the one presented in [Par90], due to the fact that orderings are decorated with an extra argument which allows to restrict orderings to the set of necessary values. This rule encompasses all techniques described in [Bir80].

Our main contribution in the discussion of *tupling* is the idea of comparing descent functions. Pettorossi [Pet84b] does not explicitly give the general embeddings for Cohen's classes of functions; he gives examples from those classes, however.

In general, this chapter presents many techniques and shows many relationships between those. Most results have been presented in an applicative style, using predicates and some new notations. The important intuitive aspects of the techniques have been described in such a way that they can be better understood.

2.9.2 Related techniques and extensions

The idea of *memoization*, introduced by Michie [Mic67], is that, in recursive evaluations, function results are stored after they are computed, and retrieved when they would otherwise be computed again. Thus, one could describe it as tabulation without inverting the order of evaluation. A formal semantics of memoization in functional languages is given in [Pet84a]. Memoization could be described using **remember/recall**. For efficient implementation of memoization, the compatible orderings described in section 2.7.4 could be useful.

Inversion techniques have been described for linear recursive and tree-like recursive functions. Systems of mutually recursive functions provide no additional problems, since they can be reduced to single recursive functions. Most of the techniques in this chapter do not apply to *nested* recursive functions, e.g. because the dependency relation $\leftarrow_f$ is expressed in terms of f. Because nested recursive functions also need a termination proof, usually based on an ordering that respects $\leftarrow_f$, tabulation techniques similar to that in section 2.7.4 may nevertheless be applicable to them.

Chapter 3

Factorization of the factorial – an algorithm derived by playing with transformations

3.1 Introduction

Chapter 2 presented some transformation rules for *inverting the order of evaluation*. This transformation technique can be applied to recursive functions, aiming at improvement of efficiency. The functions resulting from this transformation use (possibly, among others) the same arguments in the recursive evaluation as the original functions, but in an inverted order.

The example function *facthalf* in chapter 2 (cf. scheme 2.11) possesses a particular algebraic property. Using this property, by a sequence of transformations (including inverting the order of evaluation) a, to our knowledge, new algorithm for computing factorials is derived. It is also possible to derive a version of the algorithm for execution on two processors.

This chapter is organized as follows. In the next two sections our framework is introduced: the language and some notations specific to this chapter in section 3.2, and a short description of the methodology in section 3.3.

In section 3.4, some transformation rules (*inverting the flow of computation* and *splitting linear recursion*) that are needed later are discussed.

The definition of the factorial function *fact* using the new function *facthalf* is presented in section 3.4.2. It is shown in section 3.5.1 how a particular property can be used to optimize the new algorithm for the factorial function. Section 3.5.2 shows how the algorithm in section 3.5.1 can be optimized by inverting the flow of computation. The efficiency of the resulting algorithm may be clearer to a computer scientist than it is to a mathematician, because multiplications and divisions by 2 are not considered "special" in mathematics. The resulting algorithm is, in our opinion, only intelligible by way of its derivation.

In section 3.6, the complexity of the resulting algorithm is investigated. It appears that the time complexity of the algorithm is $\mathcal{O}(\log x\ \mathcal{M}(x\ \log x, x\ \log x))$, where $\mathcal{M}(y, z)$ denotes the complexity of multiplying a number of $\mathcal{O}(y)$ and a number of $\mathcal{O}(z)$ bits. Assuming

efficient multiplication, this seems to be more complex than the usual factorial definition. Computer tests have shown, however, that our algorithm is still faster than the usual one for $2^{32} < fact(x) < 2^{3200}$.

A derivation of a variant of the algorithm in section 3.5.2 that might be implemented to run on 2 processors is presented in section 3.7.

Appendix 1 contains a proof of the transformation rule in section 3.4.2. Appendix 2 contains the Pascal version of the resulting program, and some results of comparing the new and the traditional algorithm.

3.2 Language and notation

The language used in this chapter is a variant of CIP-L [BBB+85], as used in chapter 2. Because most functions in this chapter are functions on natural numbers, the type **nat** of arguments and results is frequently omitted. $\oplus$ and $\otimes$ denote binary operators.

3.3 Methodology

A derivation in the *transformational programming* methodology is presented. The essence of this methodology is the derivation of (efficient) programs from formal specifications by applying *semantics preserving transformations*, i.e. applying a transformation rule results in a semantically equivalent program.

The strategy we use is mainly the unfold-fold strategy [BD77]. *Unfolding* is the substitution of a function call by the body of the function, with replacement of the formal parameters by the actual parameters. *Folding* is the inverse of unfolding, i.e., an instance of a function body is replaced by a function call with suitable parameters.

Most phases of the derivation start with the introduction of a new function, defined in terms of existing ones. Some motivation is usually given for the introduction of the new function, we refer to well-known strategies like *finite differencing* [PK82] and *accumulation* [Bir84]. Function calls are unfolded, often simplifications and rearrangements are done, until by folding an independent version of the new function can be obtained.

In this chapter, some special transformation rules will be used as well (cf. section 3.4). These are rules that require more complicated inductive proofs than can be provided by unfolding and folding only.

3.4 Transformation rules

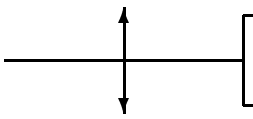
In this section, we discuss three transformation rules to be used in the rest of this chapter.

3.4.1 Inverting the flow of computation

For inverting the flow of computation, techniques from chapter 2 are used, in particular Transformation 2.13 and the following variant of Transformation 2.9:

Transformation 3.1

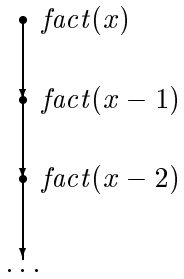
$$
\begin{array}{l}
f(x:Q(x)) = \textbf{if } T(x) \\
\qquad\qquad \textbf{then } H(x) \\
\qquad\qquad \textbf{else } x \oplus f(K(x)) \textbf{ fi}
\end{array}
$$



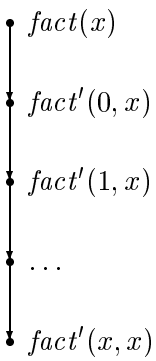
$$
\begin{array}{l}
(Q(x) \wedge \neg T(x)) \Rightarrow K^{-1}(K(x)) = x \\
(Q(x) \wedge Q(y)) \Rightarrow x \oplus (y \oplus z) = y \oplus (x \oplus z)
\end{array}
$$

$$
\begin{array}{l}
f(x:Q(x)) = f'(c,x) \\
\qquad\qquad \textbf{where} \\
\qquad\qquad f'(y,z:Q(y) \wedge Q(z)) = \\
\qquad\qquad \textbf{if } y = z \\
\qquad\qquad \textbf{then } H(c) \\
\qquad\qquad \textbf{else } K^{-1}(y) \oplus f'(K^{-1}(y),z) \textbf{ fi}
\end{array}
$$

The computational sequence of the original factorial function, viz.



is transformed by Transformation 3.1 into



More general conditions for this transformation rule can be found in chapter 2.

3.4.2 Splitting linear recursion

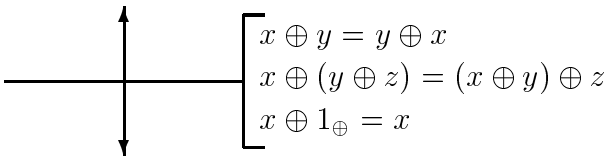
Chapter 2 presents the following alternative definition of the factorial function (cf. scheme 2.11):

3.2 $fact(x) =$ **if** $x = 0$
then 1
else $facthalf(x) \times facthalf(x-1)$ **fi**
where
$facthalf(x) =$ **if** $x \leq 1$
then 1
else $x \times facthalf(x-2)$ **fi**.

It is clear that the above function calculates the factorial of a number by calculating the products of the odd and even factors separately. Intuitively, its correctness is obvious. Formally, it is guaranteed by the correctness of the following transformation rule, which is proved in appendix 1:

Transformation 3.3

$f(x) =$ **if** $T(x)$
then $H(x)$
else $Q(x) \oplus f(K(x))$ **fi**



$f(x) =$ **if** $T(x)$
then $\bigoplus_{j=0}^{-1} Q(K^j(x)) \oplus H(x)$
elsf $T(K(x))$
then $\bigoplus_{j=0}^{0} Q(K^j(x)) \oplus H(K(x))$
$\ldots$
elsf $T(K^{n-1}(x))$
then $\bigoplus_{j=0}^{n-2} Q(K^j(x)) \oplus H(K^{n-1}(x))$
else $\bigoplus_{i=0}^{n-1} fn(K^i(x))$
fi
where
$fn(x) =$ **if** $T(x)$ **then** $H(x)$
elsf $\exists_{i=1}^{n-1} T(K^i(x))$
then $Q(x)$
else $Q(x) \oplus fn(K^n(x))$
fi,

$$\bigoplus_{i=p}^{q} g(i) = \begin{cases} 1_\oplus & \text{if } p > q \\ g(p) \oplus (\bigoplus_{i=p+1}^{q} g(i)) & \text{otherwise} \end{cases}$$

Note that the only expression of the form $\bigoplus_{i=p}^{q} g(i)$ that does not contain a term syntactically different from $1_\oplus$ occurs only in the context $\bigoplus_{i=p}^{q} g(i) \oplus H(x)$. That means that, for fixed n, all expressions of the form $\bigoplus_{i=p}^{q} g(i)$ in the transformation rule may be eliminated or simplified to expressions not containing $1_\oplus$. So, the value $1_\oplus$ may be fictitious, i.e. if no unit of $\oplus$ exists, a new element $1_\oplus$ may be adjoined to the type $\mathbf{t}$ of $\oplus$'s operands. The only property that is required of the new value $1_\oplus$ is that for all $\mathbf{t}$ $x : x \oplus 1_\oplus = x$.

Intuitively, this transformation rule splits the calculation of a term

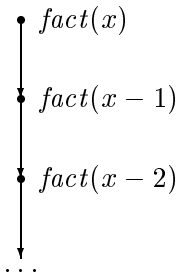
$$x_1 \oplus x_2 \oplus x_3 \oplus \ldots \oplus x_p$$

into n calculations of terms

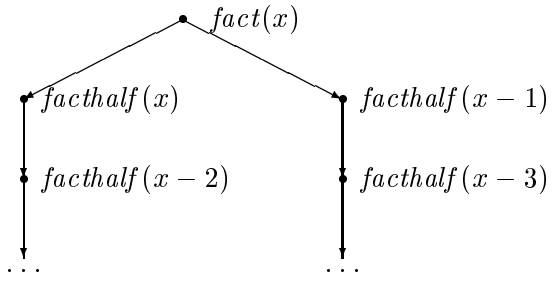
$$\begin{aligned} x_1 \oplus x_{n+1} \oplus x_{2n+1} \oplus \ldots &\quad , \\ x_2 \oplus x_{n+2} \oplus x_{2n+2} \oplus \ldots &\quad , \\ \ldots &\quad , \\ x_n \oplus x_{2n} \oplus x_{3n} \oplus \ldots &\quad . \end{aligned}$$

Thus, a computational sequence is transformed into a computational tree with n linear branches.

The computational sequence of the original factorial function, viz.



is transformed by transformation rule 3.3 into



(for $x \geq 3$). Thus, one computation is split up into two independent computations that may be executed in parallel. It is clear that transformation rule 3.3 may be used for the evaluation of certain kinds of linear recursive functions on architectures with a fixed number (≥ 2) of processors. Bush and Gurd [BG85] present a rule for transforming linear recursion to tree-like recursion (i.e., the computation tree is split at each level, instead of only once), which is also exemplified by the factorial function.

3.5 Transformational development

3.5.1 A property of *facthalf*

By induction, it can easily be proved that the following property holds for the function *facthalf* in program **3.2**.

$$even(x) \Rightarrow facthalf\,(x) = 2^{x/2} \times fact(x/2) \tag{3.4}$$

This property is also mentioned in [PB85, exercise 4 on page 98]. In the following, / denotes integer division.

3.5	$facthalf\,(x) \times facthalf\,(x-1)$
$=_{\{(3.4)\}}$	**if** $odd(x)$
	then $facthalf\,(x) \times 2^{(x-1)/2} \times fact((x-1)/2)$
	else $fact(x/2) \times 2^{x/2} \times facthalf\,(x-1)$
	fi
$=_{\{\text{use } \mathbf{mod}\text{, comm. } \times\}}$	$facthalf\,(x-1+(x \bmod 2)) \times 2^{x/2} \times fact(x/2)$

It is clear that in this resulting expression all arguments to *facthalf* are odd. When we add this to the assertion in *facthalf*, and simplify *facthalf* accordingly, we have altogether:

3.6
$fact(x)$
$= $ **if** $x = 0$ **then** 1
else $facthalf\,(x-1+(x \bmod 2)) \times 2^{x/2} \times fact(x/2)$
fi
where
$facthalf\,(x : x \geq 1 \wedge odd(x))$
$=$ **if** $x = 1$
then 1
else $x \times facthalf\,(x-2)$ **fi**.

This is our second, in our view surprising version of *fact*, which uses mainly subtraction and division by 2 instead of subtraction by 1 in its recursion.

3.5.2 Possibilities for improving *fact*/*facthalf*

As an example, consider the evaluation of $fact(31)$ according to scheme 3.6. By repeatedly unfolding *fact*, we get:

$$fact(31) = facthalf(31) \times facthalf(15) \times facthalf(7) \times facthalf(3) \times facthalf(1) \times 2^{26}.$$

Obviously,

$$facthalf(31) = 31 \times 29 \times \ldots \times 17 \times facthalf(15),$$

i.e., there is some redundancy in the computation. We are, however, not able as yet to eliminate that redundancy. This is because $facthalf(31)$ is computed "first", and only later the intermediate result $facthalf(15)$ is again useful. Therefore, we aim at inverting the flow of computation of *fact*. The result of that is that $fact(15)$ is computed first in the computation of $fact(31)$, and only afterwards it is multiplied with the value of $facthalf(31)$. Thus, $facthalf(15)$ is used before $facthalf(31)$, and can be used as a starting value for computing $facthalf(31)$. If we want to do so, we have to find a version of *facthalf* that computes $facthalf(31)$ as $(\ldots((facthalf(15) \times 17) \times 19) \times \ldots) \times 31$. This can be achieved by also inverting the flow of computation of *facthalf*.

3.5.3 Transforming *facthalf*

Transformation rule 3.1 can be applied to *facthalf*, resulting in:

3.7
$facthalf(x : x \geq 1 \wedge odd(x))$
$= fh(1, x)$
where
$fh(y, x : x \geq y \geq 1 \wedge odd(x) \wedge odd(y))$
$=$ **if** $y = x$
then 1
else $(y + 2) \times fh(y + 2, x)$ **fi**.

The correctness of this version is guaranteed by the invertibility of $K(x) = x - 2$ and the commutativity of multiplication.

We now prove a lemma that will be useful later in the derivation. A close variant of this lemma occurs in [BG85] as an example of the application of one of their transformation rules.

Lemma 3.8 *For all p,q,x such that* $1 \leq p \leq q \leq x \wedge odd(p) \wedge odd(q) \wedge odd(x)$, $fh(p, x) = fh(p, q) \times fh(q, x)$.

Proof: Intuitively it is clear that the lemma holds, because $fh(p, q)$ is simply the product of all odd numbers from $p + 2$ up to q. Formally, this can be proved by induction on $q - x$.

Basis. If $q - x = 0$, then $fh(p, x) = fh(p, x) \times 1 = fh(p, q) \times fh(q, x)$.

Induction. Suppose the lemma holds for $x - 2k \leq q \leq x$. Then
$fh(p, x-2k-2) \times fh(x-2k-2, x) =_{\{\text{unfold } fh\}}$
$fh(p, x-2k-2) \times (x-2k) \times fh(x-2k, x) =_{\{\text{fold } fh, \text{ commutativity } \times\}}$
$fh(p, x-2k) \times fh(x-2k, x) =_{\{\text{induction}\}}$
$fh(p, x)$.□

Because multiplication is associative, the accumulation strategy (cf. [Bir84]) can be applied by definition of

$$fhf(x, y, res) = res \times fh(x, y) \tag{3.9}$$

resulting in:

3.10
$facthalf(x : x \geq 1 \wedge odd(x))$
$= fhf(1, x, 1)$
where
$fhf(y, x, res : x \geq y \geq 1 \wedge odd(x) \wedge odd(y))$
$=$ **if** $y = x$
then res
else $fhf(y+2, x, res \times (y+2))$
fi

Note that the function *fhf* is now tail recursive. The straightforward correspondence between tail-recursive programs and **while**-loops [BW82, Par90] yields the following imperative program:

3.11
$fhf(y, x, res : x \geq y \geq 1 \wedge odd(x) \wedge odd(y))$
$=$ **begin** **var** $(vy, vx, vres) := (y, x, res)$;
while $vy \neq vx$
do $(vy, vx, vres) := (vy+2, vx, vres \times (vy+2))$
od;
$vres$
end

This can be simplified by eliminating all assignments to vx and replacing all other occurrences of vx by x, yielding:

3.12
$fhf(y, x, res : x \geq y \geq 1 \wedge odd(x) \wedge odd(y))$
$=$ **begin** **var** $(vy, vres) := (y, res)$;
while $vy \neq x$
do $(vy, vres) := (vy+2, vres \times (vy+2))$
od;
$vres$
end.

3.5.4 Transforming *fact*

As mentioned before, in order to profit from the new version of *facthalf*, we need to invert the flow of computation of *fact* as well. Furthermore, some optimizations (viz. accumulation and finite differencing) are possible afterwards.

First the complicated expression $x - 1 + (x \textbf{ mod } 2)$ is abstracted. Note that it denotes the greatest odd number less than or equal to x. This definition of *toodd* is used throughout this chapter.

3.13 $toodd(x : x > 0) = x - 1 + (x \textbf{ mod } 2)$

The function *fact* can now be improved by inverting the flow of computation.

If $K(x) = x/2$ were invertible, transformation rule 3.1 would be applicable. This is not the case, and section 2.4 suggests that we should find a generalized left inverse of K in order to apply transformation rule 2.13. Since $K^j(x) = x/2^j$, we can define K^{-1}, using scheme 2.12, by:

3.14 $K^{-1}(x, y : \exists k > 0 : y = x/2^k) = \textbf{that } z : z/2 = y \wedge \exists k : z = x/2^k.$

Later on, an efficient definition of K^{-1} can be given. The definition of K^{-1} will not be repeated in the derivations. Using transformation rule 2.13, we now invert the flow of computation of *fact*, resulting in:

3.15
$fact(x) = fact'(0)$
where
$fact'(y)$
$=$ **if** $y = x$
then 1
else $2^y \times facthalf(toodd(ny)) \times fact'(ny)$
where $ny = K^{-1}(x, y)$
fi.

By using the definition of *facthalf* above, the **else**-branch transforms into:

$$2^y \times fhf(1, toodd(ny), 1) \times fact'(ny).$$

The next goal is now improvement of *fact* by finite differencing. We aim at carrying along the value of *fhf* last computed. Furthermore, because the last call of *fhf* has $toodd(ny)$ as an argument, the value of $toodd(ny)$ will also be kept. First we define a new function $fact''$ with appropriate assertion (note that when $y = 0$, no *fhf* value has been computed yet, and thus the assertion should give no extra information):

3.16
$fact''(y, z, oddy :$
$\quad y \neq 0 \Rightarrow (oddy = toodd(y) \wedge z = fhf(1, oddy, 1)))$
$= fact'(y).$

By unfolding, abstraction and simplification we get:

3.17 $fact''(y, z, oddy :$
$\quad y \neq 0 \Rightarrow (oddy = toodd(y) \wedge z = fhf(1, oddy, 1)))$
$= \textbf{if } y = x$
$\quad \textbf{then } 1$
$\quad \textbf{else } 2^y \times fhf(1, toodd(ny), 1) \times fact'(ny)$
$\quad\ \textbf{where } ny = K^{-1}(x, y)$
$\quad \textbf{fi}.$

The following simplification is possible:

$$\begin{aligned} fhf(1, toodd(ny), 1) &=_{\{\text{lemma 3.8, def. } fhf\}} fhf(1, oddy, 1) \times fhf(oddy, toodd(ny), 1) \\ &=_{\{(3.9)\}} fhf(oddy, toodd(ny), fhf(1, oddy, 1)) \\ &=_{\{\text{assertion } fact''\}} fhf(oddy, toodd(ny), z). \end{aligned}$$

Using this, we can fold $fact''$ (the fhf value just computed is the correct new value for z, according to the assertion), resulting in:

3.18 $fact''(y, z, oddy :$
$\quad y \neq 0 \Rightarrow (oddy = toodd(y) \wedge z = fhf(1, oddy, 1)))$
$= \textbf{if } y = x$
$\quad \textbf{then } 1$
$\quad \textbf{else } 2^y \times nz \times fact''(ny, nz, oddny)$
$\quad\ \textbf{where } ny = K^{-1}(x, y), oddny = toodd(ny), nz = fhf(oddy, oddny, z)$
$\quad \textbf{fi}.$

For $fact$, we then have

$$fact(x) = fact''(0, 1, 1).$$

Due to commutativity of multiplication and addition, the accumulation strategy can also be applied to $fact''$. We define

$$fact'''(y, z, oddy, res, two) = fact''(y, z, oddy) \times res \times 2^{two}.$$

This allows the derivation of:

3.19 $fact(x)$
$= fact'''(0, 1, 1, 1, 0)$
$\quad \textbf{where}$
$fact'''(y, z, oddy, res, two :$
$\quad y \neq 0 \Rightarrow (oddy = toodd(y) \wedge z = fhf(1, oddy, 1)))$
$= \textbf{if } y = x$
$\quad \textbf{then } res \times 2^{two}$
$\quad \textbf{else } fact'''(ny, nz, oddny, res \times nz, two + y)$
$\quad\ \textbf{where } ny = K^{-1}(x, y), oddny = toodd(ny), nz = fhf(oddy, oddny, z)$
$\quad \textbf{fi}.$

A parameter n is added, such that $y = x/2^n$. Thus, we get the variant of Paterson and Hewitt's method presented by Bauer and Wössner [BW82]. Because $ny = x/2^{n-1}$, we have a more efficient expression for K^{-1}. The initial value should be $\lfloor {}^2\log x \rfloor + 1$ for $x > 0$, since $x/2^{\lfloor {}^2\log x \rfloor + 1} = 0$. Because ${}^2\log 0$ is undefined, we single out 0 in the definition of *fact*, which results in:

3.20 $fact(x)$
$= \textbf{if } x = 0 \textbf{ then } 1$
$\quad \textbf{else } fact'''(0, 1, 1, 1, 0) \textbf{ fi}.$

Note that K^{-1} might be implemented even more efficiently: all first arguments to $fact'''$ are of the form $x/2^k$, and so their binary representations are *prefixes* of the binary representation of x. K^{-1} transforms a prefix of length n into a prefix of length $n+1$, and this could also be achieved by gradually *shifting* x into a location.

Finite differencing by introduction of a parameter n such that $y = x/2^n$ yields:

3.21 $fact(x)$
$= \textbf{if } x = 0 \textbf{ then } 1$
$\quad \textbf{else } fact''''(0, 1, 1, 1, 0, \lfloor {}^2\log x \rfloor + 1) \textbf{ fi}$
$\quad \textbf{where}$
$fact''''(y, z, oddy, res, two, n :$
$\quad oddy = toodd(y) \wedge z = fhf(1, oddy, 1) \wedge y = x/2^n)$
$= \quad \textbf{if } y = x$
$\quad \textbf{then } res \times 2^{two}$
$\quad \textbf{else } fact''''(ny, nz, oddny, res \times nz, y + two, n - 1)$
$\quad\ \textbf{where } ny = x/2^{n-1}, oddny = toodd(ny), nz = fhf(oddy, oddny, z)$
$\quad \textbf{fi}.$

3.5.5 The imperative level

Because $fact''''$ is tail recursive, we can transform **3.21** (with unfolding of all value abstractions and the imperative counterpart of $fact''''$) into:

3.22 $fact(x)$
$= \textbf{if } x = 0 \textbf{ then } 1 \textbf{ else}$
$\quad \textbf{begin}$
$\quad\ \textbf{var } (vy, vz, voddy, vres, vtwo, vn) := (0, 1, 1, 1, 0, \lfloor {}^2\log x \rfloor + 1);$
$\quad\ \textbf{while } vy \neq x$
$\quad\ \textbf{do}$
$\quad\quad (vy, vz, voddy, vres, vtwo, vn) :=$
$\quad\quad (x/2^{vn-1}, fhf(voddy, toodd(x/2^{vn-1}), vz), toodd(x/2^{vn-1})$
$\quad\quad , vres \times fhf(voddy, toodd(x/2^{vn-1}), vz), vtwo + vy, vn - 1)$
$\quad\ \textbf{od};$
$\quad\ vres \times 2^{vtwo}$
$\quad \textbf{end fi}.$

The inner assignment statement can be sequentialized as follows:

3.23
$vtwo := vtwo + vy;$
$vn := vn - 1;$
$vy := x/2^{vn};$
$vz := \mathit{fhf}\,(voddy, toodd(vy), vz);$
$voddy := toodd(vy);$
$vres := vres \times vz$

Now we unfold *fhf* in the assignment to vz, yielding:

3.24
$vz :=$ **begin**
var $(va, vb) := (voddy, vz);$
while $va \neq toodd(vy)$
do $(va, vb) := (va + 2, vb \times (va + 2))$ **od**
vb
end;

The following optimizations are now possible:

- because va is initialized with $voddy$, and $voddy$ is not used in the inner loop, and va equals $toodd(vy)$ upon termination, $voddy$ can replace va, thereby making the assignment to $voddy$ superfluous;
- vb is initialized with vz, vz is not used in the inner loop, and after the inner loop vz is assigned vb; thus, vz can replace vb. This can also be derived via a sequence of small transformation steps.
- The assignments in the inner loop can be sequentialized in such a way that the expression $va + 2$ (now: $voddy + 2$) is computed only once.

This yields our final program, in which independent collateral assignments are not sequentialized:

3.25
$fact(x)$
$=$ **if** $x = 0$ **then** 1 **else**
begin
var $(vy, vz, voddy, vres, vtwo, vn) := (0, 1, 1, 1, 0, \lfloor {}^2\log x \rfloor + 1);$
while $vy \neq x$
do $vtwo := vtwo + vy;$
$vn := vn - 1;$
$vy := x/2^{vn};$
while $voddy \neq toodd(vy)$
do $voddy := voddy + 2;$
$vz := vz \times voddy$
od;
$vres := vres \times vz$

od;
$vres \times 2^{vtwo}$
end fi.

3.6 Complexity of the resulting algorithm

The *bit complexity* [AHU75] of the above algorithm will be calculated in this section. We will use the fact that $fact(x)$ has $\mathcal{O}(x \log x)$ bits. The expression $\mathcal{M}(f, g)$ denotes the complexity of multiplying a number of $\mathcal{O}(f)$ and a number of $\mathcal{O}(g)$. $\mathcal{M}(f, f)$ is abbreviated to $\mathcal{M}(f)$. Multiplication is assumed to be more expensive than addition, division by 2^n, or assignments and tests.

The time complexity of algorithm **3.25** is calculated as follows:

- The inner loop is executed $x/2$ times in the calculation of $fact(x)$. The complexities of the operations that occur in the inner loop are:
  - $\mathcal{O}(\log x) + 1$ (2 times),
  - $\mathcal{O}(\log x) \times \mathcal{O}(x \ \log x)$.

  Thus, the total complexity of all executions of the inner loop is $\mathcal{O}(x \ \mathcal{M}(\log x, x \ \log x))$. Note that this is also the time complexity of the usual method of computing $fact(x)$.

- The outer loop is executed $\log x$ times. The complexities of the operations that occur in the outer loop are:
  - $\mathcal{O}(\log x) + \mathcal{O}(\log x)$,
  - $\mathcal{O}(\log \log x) - 1$,
  - $\mathcal{O}(\log x)/2^{\mathcal{O}(\log \log x)}$,
  - $\mathcal{O}(\log x) - 1$,
  - $\mathcal{O}(x \ \log x) \times \mathcal{O}(x \ \log x)$,
  - some assignments and tests.

  Thus, the total complexity of all executions of the rest of the outer loop is $\mathcal{O}(\log x \ \mathcal{M}(x \ \log x))$.

- Finally, an $\mathcal{O}(x \ \log x)$ number is left-shifted $\leq 2x$ positions.

The total time complexity of the algorithm is $\mathcal{O}(\log x \ \mathcal{M}(x \ \log x))$, if fast multiplication, i.e. Schönhage-Strassen multiplication, is used. Thus, the complexity of the above algorithm is slightly (a factor $\log x / \log \log \log x$) larger than that of the usual factorial algorithm.

The test results in appendix 2 suggest that for many values of x the algorithm above is much faster than the traditional one, however.

The space complexity of the algorithm is as follows:

- *vz* and *vres* are of size $\mathcal{O}(x \ \log x)$;
- x, *vy*, *voddy*, and *vtwo* are of size $\mathcal{O}(\log x)$;
- *vn* is of size $\mathcal{O}(\log \log x)$.

The total space complexity is thus about twice as much as that of the "usual" factorial algorithm: here two locations that can hold a value of $\mathcal{O}(fact(x))$ are necessary instead of one.

3.7 Implementation on two processors

We will demonstrate how the above algorithm can be implemented on two processors. The first processor sends a sequence of appropriate *facthalf* values to the second one, which computes *fact* using those values.

In order to derive a version of the algorithm which is close to a parallel one, we need to introduce sequences. Often, in functional descriptions of parallel systems so-called *streams* are used to describe the communication (cf. [BB84]), but in this case only finite streams, i.e., sequences are needed. Because now multiple types occur in our functions, we write **nat** for natural number arguments and results, **bool** for booleans, and **seq** for sequences of natural numbers.

The type **seq**	
$<>$	the empty sequence
$\#$	prepend a natural number to a sequence
first	the first element of a sequence
rest	all but the first element

The derivation starts from the version of *fact* in program **3.15**. **nat** x is assumed to be known in the context. Below, a function *factp2* is defined which is equal to $fact'$, except that it takes as an extra argument the sequence of all necessary *facthalf* values. This is expressed by the predicate *allfh*.

3.26
$$
\begin{array}{l}
factp2(\textbf{seq } s, \textbf{nat } y : y \leq x \wedge allfh(s,y))\textbf{nat} \\
= fact'(y) \\
\textbf{where} \\
allfh(\textbf{seq } s, \textbf{nat } y)\textbf{bool} \\
= \textbf{if } y = x \textbf{ then } s = <> \\
\quad \textbf{else first } s = facthalf(toodd(ny)) \wedge allfh(ny, \textbf{rest } s) \\
\qquad \textbf{where } ny = K^{-1}(x,y) \textbf{ fi}
\end{array}
$$

In order for *factp2* to replace $fact'$, we need to derive:

- a definition of *factp2* independent of $fact'$, and
- a value z such that $factp2(z,0) = fact'(0)$; in particular, this means that the assertion $allfh(z,0)$ should hold.

First we derive a definition of *factp2*.

$$
\begin{array}{rl}
factp2(s,y) =_{\{\text{unfold } fact'\}} & \textbf{if } y = x \textbf{ then } 1 \\
& \textbf{else } 2^y \times facthalf(toodd(ny)) \times fact'(ny) \\
& \textbf{where } ny = K^{-1}(x,y) \textbf{ fi} \\
=_{\{\text{definition } allfh\}} & \textbf{if } y = x \textbf{ then } 1 \\
& \textbf{else } 2^y \times \textbf{first } s \times factp2(\textbf{rest } s, ny) \\
& \textbf{where } ny = K^{-1}(x,y) \textbf{ fi}
\end{array}
$$

The value of s for the initial call can be computed from the assertion. A value z is needed, such that $allfh(z,0)$ holds. That value of z will be denoted by *factp1*, which implicitly depends on x, but also on y. The dependence on y is necessary to derive a recursive definition of *factp1*. In the derivation below, ny is assumed to be $K^{-1}(x,y)$.

$$
\begin{array}{lll}
factp1(y) & =_{\{\text{above}\}} & \textbf{some } s : allfh(s,y) \\
& =_{\{\text{def. } allfh\}} & \textbf{some } s : \textbf{if } y = x \textbf{ then } s = <> \\
& & \qquad \textbf{else first } s = facthalf(toodd(ny)) \\
& & \qquad\quad \wedge allfh(\textbf{rest } s, ny) \\
& & \qquad \textbf{fi} \\
& =_{\{\text{distributivity}\}} & \textbf{if } y = x \\
& & \textbf{then some } s : s = <> \\
& & \textbf{else some } s : \textbf{first } s = facthalf(toodd(ny)) \\
& & \qquad\quad \wedge allfh(\textbf{rest } s, ny) \textbf{ fi} \\
& =_{\{\textbf{some}\text{-simplification}\}} & \textbf{if } y = x \\
& & \textbf{then } <> \\
& & \textbf{else some } s : \textbf{first } s = facthalf(toodd(ny)) \\
& & \qquad\quad \wedge allfh(\textbf{rest } s, ny) \textbf{ fi} \\
& =_{\{\text{seq. decomposition}\}} & \textbf{if } y = x \\
& & \textbf{then } <> \\
& & \textbf{else } facthalf(toodd(ny)) \\
& & \qquad \# \textbf{some } s' : allfh(s', ny) \textbf{ fi} \\
& =_{\{\text{fold } factp1\}} & \textbf{if } y = x \\
& & \textbf{then } <> \\
& & \textbf{else } facthalf(toodd(ny)) \# factp1(ny) \textbf{ fi}
\end{array}
$$

Then we have altogether:

3.27
$fact(\textbf{nat } x)\textbf{nat}$
$= factp2(factp1(0), 0)$
  where
$factp1(\textbf{nat } y)\textbf{seq}$
$=$ **if** $y = x$
  then $<>$
  else $facthalf(toodd(ny)) \# factp1(ny)$
   where $ny = K^{-1}(x,y)$ **fi**,
$factp2(\textbf{seq } s, \textbf{nat } y)\textbf{nat}$
$=$ **if** $y = x$ **then** 1
  else $2^y \times \textbf{first } s \times factp2(\textbf{rest } s, ny)$
  where $ny = K^{-1}(x,y)$ **fi**,
$facthalf(\textbf{nat } y)\textbf{nat}$
$=$ **if** $y = 1$ **then** 1
  else $y \times facthalf(y-2)$ **fi**.

The functions *factp1* and *factp2* can be optimized similarly to *fact* and *facthalf* in the previous section:

Optimize *factp1*:

- Define (finite differencing)

$$\begin{aligned} factp1(y) &= factp1'(y,1), \\ factp1'(y, z : y \neq 0 \Rightarrow z = facthalf(toodd(y))) &= factp1(y). \end{aligned}$$

- Use lemma 3.8 to derive

3.28 *factp1'*(y,z)
$=$ **if** $y = x$ **then** $<>$
else $nz \# factp1'(ny, nz)$
where $ny = K^{-1}(x,y), nz = fhf(toodd(y), toodd(ny), z)$
fi.

Optimize *factp2*:

- Define (accumulation)

$$\begin{aligned} factp2(s,y) &= factp2'(s,y,0,1), \\ factp2'(s,y,two,res) &= 2^{two} \times res \times factp2(s,y). \end{aligned}$$

- Derive

3.29 *factp2'*(s,y,two,res)
$=$ **if** $y = x$
then $2^{two} \times res$
else *factp2'*(**rest** $s, ny, two + y, res \times$ **first** s)
where $ny = K^{-1}(x,y)$ **fi**.
fi

- Implement K^{-1} as in the previous section.

It is clear that the sequence s produced by *factp1* and used in *factp2* may also be viewed as a one-way communication channel. See [BB84] for a discussion of this kind of consumer-producer programs. The function *factp1* may be implemented on one processor, and send the computed successive first elements of s to the other processor on which *factp2* is implemented.

It is even possible to use a third processor for computing the sequence of K^{-1} values. In the current version, these are computed by both *factp1* and *factp2*.

3.8 Concluding remarks

Using well-known techniques and a number of rules for inverting the order of evaluation, guided by a few simple heuristics for the application of these rules, a previously unknown algorithm was derived. The algorithm appears very complicated, and is in our opinion only intelligible by way of its derivation.

In [Bor85] a factorial algorithm is presented which is based on factoring out *all* prime factors. Its time complexity is better than that of our algorithm. However, it needs more space, viz. for a table of all prime numbers up to the argument of *fact*.

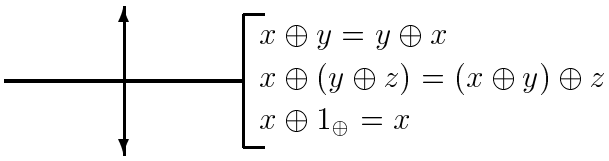
Using conventional program transformation techniques, a different version of the program was derived. This version has a particular shape, viz. that of a consumer/producer program [BB84], and thus it can be interpreted as a program for two cooperating processors. This supports the claim that programs for various unconventional architectures can be derived by using conventional program transformation techniques to arrive at programs of particular forms [DFH+91].

Appendix 1: Proof of the transformation rule"splitting linear recursion"

For convenience, transformation rule 3.3 is given again, with the second version of f labeled as f'.

Transformation 3.30

$f(x) =$ **if** $T(x)$
then $H(x)$
else $Q(x) \oplus f(K(x))$ **fi**



$f'(x) =$ **if** $T(x)$
then $\bigoplus_{j=0}^{-1}\ Q(K^j(x)) \oplus H(x)$
elsf $T(K(x))$
then $\bigoplus_{j=0}^{0}\ Q(K^j(x)) \oplus H(K(x))$
$\ldots$
elsf $T(K^{n-1}(x))$
then $\bigoplus_{j=0}^{n-2}\ Q(K^j(x)) \oplus H(K^{n-1}(x))$
else $\bigoplus_{i=0}^{n-1}\ fn(K^i(x))$
fi

where
$fn(x) =$ **if** $T(x)$ **then** $H(x)$

elsf $\exists_{i=1}^{n-1}\ T(K^i(x))$
then $Q(x)$
else $Q(x) \oplus fn(K^n(x))$
fi.

First, due to definedness of the function f,

$$\forall x\ \exists k \geq 0 : T(K^k(x))$$

holds. Let k_x, where subscript x will occasionally be dropped for convenience, be defined by

$$k_x = \mathbf{min}\{p \mid T(K^p(x))\}.$$

k_x denotes the number of recursive calls in the evaluation of $f(x)$, and thus the number of times that $f(x)$ may be unfolded. In order to prove the correctness of the rule, we will need two lemmas.

Lemma 3.31 Unfold 1

$$f(x) = \bigoplus_{j=0}^{k-1} Q(K^j(x)) \oplus H(K^k(x))$$

Proof: By unfolding f k_x times. □

Lemma 3.32 Unfold 2

$$k_x \geq n \Rightarrow f(x) = \bigoplus_{j=0}^{n-1} Q(K^j(x)) \oplus f(K^n(x))$$

Proof: By unfolding f n times, which is allowed because $k_x \geq n$. □

Proof of the rule: By induction on k_x.

$0 \leq k_x < n$**:**

$$\begin{aligned} f'(x) &=_{\{\text{definition}\}} \bigoplus_{j=0}^{k-1} Q(K^j(x)) \oplus H(K^k(x)) \\ &=_{\{\text{Lemma 3.31}\}} f(x) \end{aligned}$$

$n \leq k_x < 2n$**:**

Set $p = k - n$, so $0 \leq p < k$.

$$
\begin{aligned}
f'(x) &=_{\{\text{definition}\}} \bigoplus_{i=0}^{n-1} fn(K^i(x)) \\
&=_{\{\text{unfold } fn\}} \bigoplus_{i=0}^{p} (fn(K^{n+i}(x)) \oplus Q(K^i(x))) \oplus \bigoplus_{i=p+1}^{n-1} Q(K^i(x)) \\
&=_{\{\text{commutativity } \oplus\}} \bigoplus_{i=0}^{n-1} Q(K^i(x)) \oplus \bigoplus_{i=0}^{p} fn(K^{n+i}(x)) \\
&=_{\{\text{unfold } fn\}} \bigoplus_{i=0}^{n-1} Q(K^i(x)) \oplus \left(\bigoplus_{i=0}^{p-1} Q(K^{n+i}(x)) \right) \oplus H(K^{n+p}(x)) \\
&= \bigoplus_{i=0}^{k-1} Q(K^i(x)) \oplus H(K^k(x)) \\
&=_{\{\text{Lemma 3.32}\}} f(x)
\end{aligned}
$$

$k_x \geq 2n$**:**

$$
\begin{aligned}
f'(x) &=_{\{\text{definition}\}} \bigoplus_{i=0}^{n-1} fn(K^i(x)) \\
&=_{\{\text{unfold } fn\}} \bigoplus_{i=0}^{n-1} (Q(K^i(x)) \oplus fn(K^{n+i}(x))) \\
&=_{\{\text{commutativity } \oplus\}} \bigoplus_{i=0}^{n-1} Q(K^i(x)) \oplus \bigoplus_{i=0}^{n-1} fn(K^{n+i}(x)) \\
&=_{\{\text{fold } f', \text{ induction}\}} \bigoplus_{i=0}^{n-1} Q(K^i(x)) \oplus f(K^n(x)) \\
&=_{\{\text{Lemma 3.32}\}} f(x) \qquad \square
\end{aligned}
$$

Appendix 2: Testing the algorithm

Algorithm **3.25** was tested on a VMS VAX, in VAX Pascal. The Pascal program is included at the end of this appendix.

The table below contains some test results. The first column shows the argument, the second one shows the number of tests. In the third column the time used by the traditional factorial algorithm is shown (in seconds). The last column contains the time used by our algorithm.

Argument	#	trad.	new
10	100	0.11	0.13
20		0.23	0.20
30		0.45	0.27
40		0.58	0.37
50		0.81	0.48
60		1.01	0.59
70		1.34	0.79
80		1.86	0.93
90		2.39	1.39
100		2.87	1.63
150	10	0.52	0.29
200		0.92	0.53
250		1.37	0.83
300		1.98	1.28
350		2.79	1.71
400		3.57	2.35

Since arbitrary length natural numbers are not available in VAX Pascal, we had to code them (together with their operations) ourselves. We only implemented arrays of 100 natural numbers, thus extending the natural numbers to 2^{3200}. The operation `xdivmaxm`, which is supposed to calculate the product of two numbers modulo 2^{32}, is not correct as it is, but it introduces relatively small errors only. We did not implement a correct version, since it would consume enormous amounts of time, whereas on the machine language level the "overflow" of a multiplication is usually available as a by-product.

```
PROGRAM fact(input,output);
{
MODULE bignumbers: type big with multiplication and exponentiation
operations
}
CONST maxbig=100;
TYPE big=RECORD fill:1..maxbig;
                nr:ARRAY[1..maxbig]OF UNSIGNED
         END;
{    UNSIGNED = 1..(2**32)-1 }
VAR big1: big := (1,(1,(maxbig-1) OF 0));
FUNCTION xdivmaxm(a,b:unsigned):unsigned; { a*b DIV (2**32) }
BEGIN
  xdivmaxm := ((a DIV (2**16)) * (b DIV (2**16)))
           + (((a DIV (2**16)) * (b MOD (2**16)))DIV (2**16))
           + (((a MOD (2**16)) * (b DIV (2**16)))DIV (2**16))
END;
```

```
FUNCTION bigxsmall(x:unsigned;y:big):big; {x*y}
VAR i:1..maxbig; bigxsmallx:big;
BEGIN
  bigxsmallx := big1;
  bigxsmallx.nr[1] := y.nr[1]*x;
  FOR i := 1 TO min(y.fill,maxbig-1) DO
    bigxsmallx.nr[i+1] := xdivmaxm(y.nr[i],x)
                        + (y.nr[i+1]*x);
  bigxsmallx.fill := y.fill;
  IF y.fill<maxbig
  THEN IF bigxsmallx.nr[y.fill+1]>0
       THEN bigxsmallx.fill := y.fill+1;
  bigxsmall := bigxsmallx
END;
FUNCTION ptwo(f:unsigned):big; {2**f}
VAR res:big; i:unsigned;
BEGIN i := f; res := big1;
  WHILE i>=32 DO BEGIN
    res.nr[res.fill] := 0;
    res.fill := res.fill+1;
    i := i-32
  END;
  res.nr[res.fill] := 2**i;
  ptwo := res
END;
FUNCTION bigxbig(x,y:big):big; {x*y}
VAR res:big;
    i,j:1..maxbig;
BEGIN
  res := big1;
  res.nr[1] := x.nr[1]*y.nr[1];
  res.fill := min(x.fill+y.fill,maxbig);
  FOR i := 2 TO res.fill DO BEGIN
    res.nr[i] := 0;
    FOR j := 1 TO i-1
    DO res.nr[i] := res.nr[i] + xdivmaxm(x.nr[j],y.nr[i-j]);
    FOR j := 1 TO i
    DO res.nr[i] := res.nr[i] + (x.nr[j] * y.nr[i-j+1])
  END;
  WHILE res.nr[res.fill]=0 DO res.fill := res.fill-1;
  bigxbig := res
END;
```

```
{
MODULE traditional factorial
}
FUNCTION bigfact(x:unsigned):big; {usual fact algorithm}
VAR res:big; y:unsigned;
BEGIN
  y := x; res := big1;
  WHILE y>0 DO BEGIN
    res := bigxsmall(y,res);
    y := y-1
  END;
  bigfact := res
END;
PROCEDURE testfact(x,n:unsigned); {test usual for x, n times}
VAR y:unsigned; z:big;
BEGIN
  FOR y := 1 TO n DO z := bigfact(x)
END;
{
MODULE new factorial
}
FUNCTION newfact(x:unsigned):big; {new fact algorithm}
VAR vy,voddy,vtwo,vn,tood: unsigned;
    vz,vres:big;
BEGIN
  IF x=0 THEN newfact := big1 ELSE BEGIN
  vy := 0; vz := big1; voddy := 1; vres := big1; vtwo := 0;
  vn := trunc(ln(x)/ln(2))+1;
  WHILE vy <> x DO BEGIN
    vtwo := vtwo+vy;
    vn := vn-1;
    vy := x DIV (2**vn);
    IF odd(vy) THEN tood := vy ELSE tood := vy-1;
    WHILE voddy<>tood DO BEGIN
      voddy := voddy+2;
      vz := bigxsmall( voddy,vz)
    END;
    vres := bigxbig(vres,vz)
  END;
  newfact := bigxbig(vres,ptwo(vtwo))
END END;
```

```
PROCEDURE testnewfact(x,n:unsigned); {test new for x, n times}
VAR y:unsigned; z:big;
BEGIN
  FOR y := 1 TO n DO z := newfact(x)
END;
{
test program
}
VAR i,timex,timey,tests,testfrom,testto:unsigned;
BEGIN {main}
write('Enter #tests, from, to:');
readln(tests,testfrom,testto);
FOR i := testfrom TO testto DO
BEGIN
  timex := clock;
  testfact(i,tests);
  writeln('Time:',clock-timex,' ');
  timey := clock;
  testnewfact(i,tests);
  writeln('Time:',clock-timey)
END
END.
```

Chapter 4

A note on similarity of specifications and reusability of transformational developments

4.1 Introduction

Formal specification and transformational programming are advertised to solve major problems in program development. Reusability is claimed to be one of the promising advantages of this new methodology. In fact, there are several convincing aspects that support this claim:

- formalizations of data structures by means of *algebraic types*, together with collections of possible implementations, provide reusable "software components";
- *transformation rules*, *tactics*, and *strategies* are formalizations of "programming knowledge" to be reused in the development of algorithms;
- recorded *transformational developments* are reusable as guidelines in deriving algorithms for modified, "similar" specifications ("replay").

The potential benefits of these aspects are obvious. Numerous case studies exist to illustrate and verify the first two arguments. However, admittedly, little experience with respect to reusability of transformational developments has been gained so far.

In this note we try to illuminate the role of similarity of specifications with respect to reusability of transformational developments by means of a very simple example. The purpose of this case study is to shed some light on this particular form of reuse, which might help in building up an appropriate theory.

In particular, we would like to get some better understanding of the possibilities, the potentials as well as the limitations of reuse of transformational developments. More ambitious goals, such as a theory of reuse or a methodology (based on reuse) for deriving new algorithms, are outside the focus of this treatment.

The term "reuse of transformational developments" is used to characterize a procedure where the knowledge from an existing transformational development

$$S_1 \xrightarrow{T_1} S_2 \xrightarrow{T_2} \ldots S_n \xrightarrow{T_n} S_{n+1}$$

(where specification S_{i+1} is obtained from specification S_i by applying transformation T_i) for a certain problem statement S_1 is somehow used to facilitate a development for a "similar" problem statement S_1'. Of course, the crucial aspect is the way how "similar" is defined, because it essentially determines the net gain in reusability and the ease of doing so.

One can think of several interpretations of the notion "similar". One possibility is obviously the informal one which relies on an intuitive, pragmatic understanding of the word. If, instead, a formal definition is aimed at, again two possibilities exist: a descriptive way by exhibiting characteristic properties of similar specifications, or a more operational one by indicating how to construct similar specifications from existing ones. Of course, if mechanization of transformational developments is a final goal, the latter possibility is the one to aim at.

In the following sections we will make these different ways of looking at similarity more precise and illustrate, by means of an example, what the consequences and benefits are with respect to reuse. In particular, we will demonstrate that quite complicated algorithms, such as the Boyer-Moore and Knuth-Morris-Pratt pattern matching algorithms, can be derived straightforwardly through reuse of appropriately generalized, simple developments.

For the formulation of specifications, the ALGOL-variant of the language CIP-L [BBB+85] is used throughout this paper. Most of the constructs appearing in the text are self-explanatory. The symbols Δ, ∇ and $\Rightarrow$ denote sequential conjunction, disjunction and implication, respectively. They are mainly used to abbreviate conditionals. In addition to these sequential boolean operators, also the usual ones will appear – in particular, in order to stress collaterality of (boolean) subexpressions.

The transformation rules in our sample derivations are given informally on purpose: First, most of them are obvious and well-known anyhow. Second, we did not want to commit ourselves to a particular formal notation before having more profound knowledge on the impact of issues related to reusability on a suitable formal notation. Third, a formal treatment of transformations is outside the focus of this case study, would unnecessarily burden the presentation of our examples, and, furthermore, can be found elsewhere (e.g., [Par90]). Also for this latter reason, we omit justifications of applicability conditions for the transformation rules. Thus, e.g., proving well-foundedness of a recursion introduced by folding will not be commented on explicitly. In all cases occurring it will be trivial anyhow.

4.2 A simple derivation

As a starting point for our further considerations we consider the following simple problem:

Check whether a natural number occurs in a non-empty array of natural numbers.

Assuming a suitable definition of arrays, with

- natural numbers as indices,
- dom yielding the interval of all indices in the domain of the array,
- $\downarrow m$ and $\uparrow m$ denoting the smallest and largest index in the domain of the array m, and
- [.] denoting, as usual, indexed access to an array,

a formal specification of this problem is straightforward:

4.1 **funct** $ex = (\textbf{array}\ m, \textbf{nat}\ x : dom(m) \neq \emptyset)\textbf{bool}$:
$\exists\, \textbf{nat}\ i : \downarrow m \leq i \leq \uparrow m \,\Delta\, m[i] = x.$

As straightforwardly, an algorithm that solves this problem can be derived by means of elementary transformations along a "generalized unfold-fold strategy" (cf. [Par90]) as follows:

$\exists\, \textbf{nat}\ i : \downarrow m \leq i \leq \uparrow m \,\Delta\, m[i] = x$

$\equiv$ ① – [splitting of the "search space"
(case introduction on $\leq$, distributivity, simplification)]

$m[\downarrow m] = x \,\nabla\, \exists\, \textbf{nat}\ i : \downarrow m < i \leq \uparrow m \,\Delta\, m[i] = x$

$\equiv$ ② – [embedding (generalization of $\downarrow m$); focus on auxiliary function]

$m[\downarrow m] = x \,\nabla\, ex'(\downarrow m)$ **where**
funct $ex' = (\textbf{nat}\ l)\textbf{bool}$:
$\exists\, \textbf{nat}\ i : l < i \leq \uparrow m \,\Delta\, m[i] = x$

$\equiv$ ③ – [conjunction of a consequence]

$l < \uparrow m \,\Delta\, \exists\, \textbf{nat}\ i : l < i \leq \uparrow m \,\Delta\, m[i] = x$

$\equiv$ ④ – ["reduction" of the search space (introduction of n; simplification)]

$l < \uparrow m \,\Delta\, (\exists\, \textbf{nat}\ i : n \leq i \leq \uparrow m \,\Delta\, m[i] = x$ **where** $n = l + 1)$

$\equiv$ ⑤ – [splitting of the search space]

$l < \uparrow m \,\Delta\, (m[n] = x \,\nabla\, \exists\, \textbf{nat}\ i : n < i \leq \uparrow m \,\Delta\, m[i] = x$ **where** $n = l + 1)$

$\equiv$ ⑥ – [fold auxiliary function introduced in step ②]

$l < \uparrow m \,\Delta\, (m[n] = x \,\nabla\, ex'(n)$ **where** $n = l + 1).$

If the auxiliary function ex' is looked at as an independent function, it will be undefined for arguments $l < \downarrow m$. This problem, however, will not occur here due to the particular context.

In this derivation transformations are informally described by mnemonic names and additional hints (given in parentheses). Furthermore, all transformations are labeled for later reference.

It is a general methodological advice to couple embedding (through adding new arguments) as in step ② with the introduction of assertions that relate the new arguments to the old ones. In order to (be able to) profit from these assertions also in later simplifications and improvements, they should be formulated as strong as possible. Thus, above we had better couple the embedding with an assertion such as

$miss(\downarrow m, l)$ **where**
$miss(i, j) \equiv \forall\,\mathbf{nat}\ k : (\downarrow m \leq i, j \leq \uparrow m\,\Delta\, i \leq k \leq j) \mapsto m[k] \neq x.$

This assertion records the information that for all indices equal to or less than the "current" one the search has not been successful. Obviously, the other steps of the derivation are not affected by this assertion, since we assume that folding (step ⑥) also takes care of assertions in case there are any.

Using this modified embedding step in place of the one above we obtain the algorithm

4.2 **funct** $ex = (\mathbf{array}\ m, \mathbf{nat}\ x : dom(m) \neq \emptyset)\mathbf{bool}$:
$m[\downarrow m] = x\,\nabla\, ex'(\downarrow m)$ **where**
funct $ex' = (\mathbf{nat}\ l : miss(\downarrow m, l))\mathbf{bool}$:
$l < \uparrow m\,\Delta\,(m[n] = x\,\nabla\, ex'(n)$ **where** $n = l + 1)$

which obviously reflects a simple linear search through m "from left to right".

4.3 Reuse by analogy

Using a pragmatic, intuitive notion of similarity, an informal way of reuse seems to be obvious: "reuse by analogy". This is actually the kind of reuse that is usually employed in connection with paper-and-pencil derivations for analogous problems, which comes close to the intuitive idea of a "strategy". The underlying idea is as follows:

Given a derivation $S_1 \xrightarrow{T_1} S_2 \xrightarrow{T_2} \ldots S_n \xrightarrow{T_n} S_{n+1}$ and a "similar" problem specification S'_1, then for all i, $1 \leq i \leq n$, do the following steps:

- find a transformation T'_i (exploiting the information contained in S'_i, S_i, and T_i) which is applicable to S'_i;
- apply transformation T'_i to S'_i to obtain S'_{i+1}.

The word "find" above is used to convey the informality of this step and to stress the use of intuition in performing this action. The use of "T'_i" to denote the transformations in the analogous development is to express a certain correspondence with transformation T_i. In fact, T'_i may be the identity transformation (i.e., "do nothing"), a single transformation step, but also a sequence of steps.

A simple example for this kind of reuse can be obtained for our introductory example, if we assume a slightly modified original specification

4.3 **funct** $ex = (\mathbf{array}\ m, \mathbf{nat}\ x : dom(m) \neq \emptyset)\mathbf{bool}$:
$\exists\,\mathbf{nat}\ i : \uparrow m \geq i \geq \downarrow m\,\Delta\, m[i] = x.$

In order to solve this obviously similar problem, a derivation analogous to the one in section 4.2 can be given that uses exactly the same steps ① up to ⑥. The only difference is in step ④ where, due to the difference in ordering, another successor operation, viz. $n = l - 1$, is used in reducing the search space. The (obvious) result of this analogous derivation is

4.4 **funct** $ex = (\textbf{array}\ m, \textbf{nat}\ x : dom(m) \neq \emptyset)\textbf{bool}$:
$m[\uparrow m] = x \nabla ex'(\uparrow m)$ **where**
funct $ex' = (\textbf{nat}\ l : miss(l, \uparrow m))\textbf{bool}$:
$l > \downarrow m \,\Delta\, (m[n] = x \nabla ex'(n)$ **where** $n = l - 1)$.

Another, maybe more interesting example for illustrating reuse by analogy is provided by dealing with the following problem:

> *Check whether a natural number occurs in an ordered binary tree of natural numbers.*

Assuming a suitable definition of ordered binary trees (type **ordtree**) with

- an empty tree constructor *et*, and
- a partial tree constructor $node(\textbf{ordtree}\ l, \textbf{nat}\ i, \textbf{ordtree}\ r)$,

as well as a type **path**, defined as sequences of **L** and **R**, with

- $\diamond$ denoting the empty path,
- an operation $+$, denoting concatenation of paths,
- an is-prefix predicate $\preceq$ (on paths),
- a (partial) function *get*, that returns for a tree t and a path p the value that is found at the end of p in t,
- a function *isp*, that determines whether a path leads from the root to a node,

a formal specification of this problem is:

4.5 **funct** $find = (\textbf{ordtree}\ t, \textbf{nat}\ x)\textbf{bool}$:
$\exists\, \textbf{path}\ p : isp(t, p) \,\Delta\, get(t, p) = x$.

Using analogous steps to the derivation above, we derive:

$\exists\, \textbf{path}\ p : isp(t, p) \,\Delta\, get(t, p) = x$
$\equiv$ ①
$(isp(t, \diamond) \,\Delta\, get(t, \diamond) = x) \nabla (\exists\, \textbf{path}\ p : p \neq \diamond \,\Delta\, isp(t, p) \,\Delta\, get(t, p) = x)$
$\equiv$ ②
$(isp(t, \diamond) \,\Delta\, get(t, \diamond) = x) \nabla check(\diamond)$ **where**
funct $check = (\textbf{path}\ p : \forall\, \textbf{path}\ p' : isp(t, p') \,\Delta\, get(t, p') = x \Rightarrow p \preceq p')\textbf{bool}$:
$\exists\, \textbf{path}\ p' : p' \neq \diamond \,\Delta\, isp(t, p + p') \,\Delta\, get(t, p + p') = x$
$\equiv$ ③
$isp(t, \diamond) \,\Delta\, \exists\, \textbf{path}\ p' : p' \neq \diamond \,\Delta\, isp(t, p + p') \,\Delta\, get(t, p + p') = x$
$\equiv$ ④

$isp(t, \diamond)\,\Delta\,\exists\,\mathbf{path}\ p' : isp(t, p+n+p')\,\Delta\,get(t, p+n+p') = x$
$\quad\mathbf{where}\ n = \mathbf{if}\ get(t,p) < x\ \mathbf{then\ R\ else\ L\ fi}$

$\equiv$ ⑤

$isp(t, \diamond)\,\Delta\,((isp(t, p+n)\,\Delta\,get(t, p+n) = x)\,\nabla$
$\quad\exists\,\mathbf{path}\ p' : p' \neq \diamond\,\Delta\,isp(t, p+n+p')\,\Delta\,get(t, p+n+p') = x$
$\quad\mathbf{where}\ n = \mathbf{if}\ get(t,p) < x\ \mathbf{then\ R\ else\ L\ fi})$

$\equiv$ ⑥

$isp(t, \diamond)\,\Delta\,((isp(t, p+n)\,\Delta\,get(t, p+n) = x)\,\nabla\,check(p+n)$
$\quad\mathbf{where}\ n = \mathbf{if}\ get(t,p) < x\ \mathbf{then\ R\ else\ L\ fi}).$

In both examples we used exactly the same transformation steps as in section 4.2. Other examples for analogous derivations, where the steps of the derivation differ, are provided for our initial problem, if we assume that a successor and a predecessor operation are also defined for $\uparrow m$:

4.6 $\exists\,\mathbf{nat}\ i : \downarrow m \leq i \leq \uparrow m\,\Delta\,m[i] = x$

$\equiv$ [**id**; ②; **id**; **id**; (⑤; ③; ④); ⑥]

$ex'(\downarrow m)$ **where**
funct $ex' = (\mathbf{nat}\ l : miss(\downarrow m, l-1))\mathbf{bool}:$
$l \leq \uparrow m\,\Delta\,(m[l] = x\,\nabla\,ex'(n)\ \mathbf{where}\ n = l+1)$

or

4.7 $\exists\,\mathbf{nat}\ i : \downarrow m \leq i \leq \uparrow m\,\Delta\,m[i] = x$

$\equiv$ [**id**; ②; ③; ④; **id**; ⑥]

$ex'(\downarrow m)$ **where**
funct $ex' = (\mathbf{nat}\ l : miss(\downarrow m, l-1))\mathbf{bool}:$
$m[l] = x\,\nabla\,(l < \uparrow m\,\Delta\,ex'(n)\ \mathbf{where}\ n = l+1).$

In order to stress the analogy with the previous derivations, we have used an ad hoc notation for composing single transformation steps: **id** denotes the identity transformation, ";" denotes sequential composition, and parentheses are used to visualize substructures. Obviously, **id** may be left out; it is only included in order to stress the similarity between the various developments.

It should be a straightforward exercise to convert these descriptions into existing formalisms such as DEVA (cf. [Sin87]).

4.4 Reuse by generalization based on instantiation

A possibility of formally capturing the notion of similarity is the following one that uses the auxiliary notions of generalization (and specialization) based on instantiation.

If, as in [BEH+87], a programming language is defined by an algebraic type with signature PL, and programs (or specifications) are viewed as well-formed terms from the term algebra

$W(PL)$, program schemes can be defined as terms from $W(PL \cup X)$, where X is a countable set of (typed) free variables, called scheme variables. Furthermore, instances are finite, partial mappings $\Theta : X \to W(PL \cup X)$. For a program scheme p and an instance Θ, the instantiation $p\Theta$ of p with Θ is obtained by simultaneously replacing all scheme variables in p by program schemes according to Θ, with the additional assumption that instantiation always results in well-formed terms.

If, for two specifications S and S', there exist a specification S'' and instances Θ, Θ', such that $S \equiv S''\Theta$ and $S' \equiv S''\Theta'$ hold, we call S'' a common generalization of S and S', and S, S' specializations of S''. In that case S and S' are said to be similar.

With respect to reusability, rather than solving the similar problem S' by redoing the existing development for S in analogous steps, we now aim at generalizing the available development, in order to cover a whole class of related problems which is characterized by S''.

A careful analysis of our formal specification from section 4.2 suggests various obvious possibilities for generalizing the specification in the above sense:

- an arbitrary element type **m** instead of **nat**;
- an arbitrary, non-empty, finite index type **ind** (with a strict linear ordering $<$ and an equality $=$);
- an arbitrary predicate $P(i, x)$ (instead of $m[i] = x$).

Due to finiteness and nonemptyness of the index type and the (assumed) existence of a linear ordering on it, the least and the greatest element of type **ind** are well-defined and corresponding operators $min_<$ and $max_<$ on **ind** can be defined.

With these prerequisites, the original problem stated above may be generalized (in the above sense) as follows:

funct $ex = (\mathbf{m}\ x)\mathbf{bool}$:
  $\exists\,\mathbf{ind}\ i : P(i, x)$

which is a direct formalization of the problem

> *Check whether an element of type* **m** *and an index i from a non-empty, linearly ordered, finite domain* **ind** *satisfy a predicate P.*

Note that the condition $min_< \leq j \leq max_<$ holds for any j of type **ind**, and thus can be added at all relevant positions in derivations.

When attempting to generalize the derivation for **4.2** given above, one realizes that all steps are exactly the same, except for ④ where the use of a successor operation is dependent on the actual ordering used. Of course, for our generalized derivation we also need a successor operation, i.e., an operation on **ind** to yield the "next" element as a generalization of the operation "+1" on **nat**. This operation on **ind** has to be monotonic with respect to the underlying ordering. Furthermore, it has to satisfy an assertion that generalizes *miss*, i.e. it is dependent on the predicate P.

In order to have greater flexibility, we prefer not to add an additional explicit operation to our generalized type **ind**. Instead, we aim at an implicit characterization of a suitable successor operation to yield a next element. To this end we introduce

$next_<(i, x, P) \equiv$ **some ind** $k : i < k \,\Delta\, \forall$ **ind** $k' : i \leq k' < k \Rightarrow miss(i, k', x, P)$
where $miss(i, j, x, P) \equiv \forall$ **ind** $k : i \leq k \leq j \Rightarrow \neg P(k, x)$.

Using the choice operator **some** here is motivated by the intention to delay the decision on a suitable choice until further information is available to make a "clever" choice.

With this additional definition, all steps from section 4.2 can be reused in a straightforward way, each of them resulting in a generalized version of the respective (intermediate) specification of the original development:

$\exists$ **ind** $i : P(i, x)$

$\equiv$ ①

$P(min_<, x) \,\nabla\, \exists$ **ind** $i : min_< < i \,\Delta\, P(i, x)$

$\equiv$ ②

$P(min_<, x) \,\nabla\, ex'(min_<)$ **where**
funct $ex' =$ (**ind** $l : miss(min_<, l, x, P)$)**bool** :
$\exists$ **ind** $i : l < i \,\Delta\, P(i, x)$

$\equiv$ ③

$l < max_< \,\Delta\, \exists$ **ind** $i : l < i \,\Delta\, P(i, x)$

$\equiv$ ④

$l < max_< \,\Delta\, (\exists$ **ind** $i : n \leq i \,\Delta\, P(i, x)$ **where** $n = next_<(l, x, P))$

$\equiv$ ⑤

$l < max_< \,\Delta\, (P(n, x) \,\nabla\, \exists$ **ind** $i : n < i \,\Delta\, P(i, x)$ **where** $n = next_<(l, x, P))$

$\equiv$ ⑥

$l < max_< \,\Delta\, (P(n, x) \,\nabla\, ex'(n)$ **where** $n = next_<(l, x))$.

Thus, we finally obtain the "schematic algorithm"

4.8 **funct** $ex =$ (**m** x)**bool** :
$P(min_<, x) \,\nabla\, ex'(min_<)$ **where**
funct $ex' =$ (**ind** $l : miss(min_<, l, x, P)$)**bool** :
$l < max_< \,\Delta\, (P(n, x) \,\nabla\, ex'(n)$ **where** $n = next_<(l, x, P))$.

Our algorithms **4.2** and **4.4** are now just instantiations of this schematic algorithm. But our schematic algorithm **4.8** also covers instantiations that do not appear so obvious from the trivial result obtained in the beginning. Thus, e.g. any descendant of $next_<$ which skips certain elements of **ind** when checking P results in a "fast linear search" algorithm.

If, as in the derivation of **4.6**, we assume that $next_<$ is also defined for $max_<$ (which is often the case, since **ind** may very well be a submode or a restricted mode), then, of course,

generalizations of versions **4.6** and **4.7** can be derived, too. This, however, also requires a definition of a total operation similar to $next_<$ to yield the "previous" element:

$$prev_<(i, x, P) \equiv \textbf{some ind } k : k < i \,\Delta\, \forall \textbf{ind } k' : k < k' \leq i \mapsto miss(k', i, x, P).$$

Thus, e.g., as a generalized counterpart of **4.6** we obtain

4.9 **funct** $ex = (\textbf{m } x)\textbf{bool}$:
$ex'(min_<)$ **where**
funct $ex' = (\textbf{ind } l : miss(min_<, prev_<(l, x, P), x, P))\textbf{bool}$:
$l \leq max_< \,\Delta\, (P(l, x) \,\nabla\, ex'(n)$ **where** $n = next_<(l, x, P))$.

4.5 Reuse by generalization based on descendance

An even more powerful notion of generalization is obtained, if we extend the definition from section 4.4 by the descendance relation (cf. [BBB+85]) instead of equivalence. A specification S'' is now said to be a generalization of a specification S, if there exists an instantiation Θ such that $S \subseteq S''\Theta$ (where $\subseteq$ denotes the descendance relation). As a consequence of introducing this kind of non-determinism in generalizing a specification, often the transformation steps to be reused will have to be performed more than once.

In order to give an example for this kind of generalization, we could, e.g., start with a specification that is generalized by introducing a choice. Step ① then is the same; however, it has to be performed twice, i.e., for each element of the choice, with different orderings on **ind**. Step ② remains essentially the same, however, generalizing both $min_<$ and $max_<$. The remaining steps are again performed twice. Furthermore, step ⑤ has to be supplemented by another introduction of a choice (which is motivated by the intention of folding in step ⑥):

$\exists \textbf{ind } i : P(i, x) \,[\,]\, \exists \textbf{ind } i : P(i, x)$

$\equiv$ [①, twice]

$(P(min_<, x) \vee P(max_<, x)) \,\nabla$
$(\exists \textbf{ind } i : min_< < i < max_< \,\Delta\, P(i, x) \,[\,]\, \exists \textbf{ind } i : min_< < i < max_< \,\Delta\, P(i, x))$

$\equiv$ ②

$(P(min_<, x) \vee P(max_<, x)) \,\nabla\, ex'(min_<, max_<)$ **where**
funct $ex' = (\textbf{ind } l, r : miss(min_<, l, x, P) \wedge miss(r, max_<, x, P))\textbf{bool}$:
$\exists \textbf{ind } i : l < i < r \,\Delta\, P(i, x) \,[\,]$
$\exists \textbf{ind } i : l < i < r \,\Delta\, P(i, x)$

$\equiv$ [③; ④; (⑤; introduction of a choice); ⑥, each twice]

$(l < max_< \,\Delta\, (P(n, x) \,\nabla\, ex'(n, r)$ **where** $n = next_<(l, x, P))) \,[\,]$
$(min_< < r \,\Delta\, (P(p, x) \,\nabla\, ex'(l, p)$ **where** $p = prev_<(r, x, P)))$.

Thus, we obtain as a result

4.10 **funct** $ex = (\mathbf{m}\ x)\mathbf{bool}:$
$(P(min_<, x) \vee P(max_<, x))\, \nabla\, ex'(min_<, max_<)$ **where**
funct $ex' = (\mathbf{ind}\ l, r : miss(min_<, l, x, P) \wedge miss(r, max_<, x, P))\mathbf{bool}:$
$(l < max_< \,\Delta\, (P(n, x)\, \nabla\, ex'(n, r)$ **where** $n = next_<(l, x, P)))\,[\,]$
$(min_< < r \,\Delta\, (P(p, x)\, \nabla\, ex'(l, p)$ **where** $p = prev_<(r, x, P))).$

This algorithm describes a non-deterministic scanning of the search space from either side ("outside-in"). The function ex' above comprises as descendants obviously ex' in algorithms **4.2** and **4.4**, but also a version where the search space is scanned symmetrically from both sides, or even one where scanning is done in parallel.

Of course, there are other possibilities for generalizing the initial specification in the above sense. Another example is obtained by using a (non-deterministically) selected element in the initial specification. Splitting of the search space in step ① then is done dependent on this choice. Again, step ② remains the same, the remaining ones are performed twice (for each disjunct), as before:

$\exists\mathbf{ind}\ i : (min_< \le i \le j \vee j \le i \le max_<)\, \Delta\, P(i, x)$ **where**
ind $j = \mathbf{some}\ \mathbf{ind}\ j' : min_< \le j' \le max_<$

$\equiv$ ①

$P(j, x)\, \nabla\, ((\exists\mathbf{ind}\ i : i < j\, \Delta\, P(i, x)) \vee (\exists\mathbf{ind}\ i : j < i\, \Delta\, P(i, x)))$ **where**
ind $j = \mathbf{some}\ \mathbf{ind}\ j' : min_< \le j' \le max_<$

$\equiv$ ②

$P(j, x)\, \nabla\, ex'(j, j)$ **where**
funct $ex' = (\mathbf{ind}\ l, r : miss(l, r, x, P))\mathbf{bool}:$
$(\exists\mathbf{ind}\ i : i < l\, \Delta\, P(i, x)) \vee (\exists\mathbf{ind}\ i : r < i\, \Delta\, P(i, x))$
$\equiv$ [③; ④; ⑤; ⑥, each twice]
$(min_< < l \,\Delta\, (P(p, x)\, \nabla\, ex'(p, r)$ **where** $p = prev_<(l, x, P)))\vee$
$(r < max_< \,\Delta\, (P(n, x)\, \nabla\, ex'(l, n)$ **where** $n = next_<(r, x, P))).$

Thus, we obtain as a result

4.11 **funct** $ex = (\mathbf{m}\ x)\mathbf{bool}:$
$P(j, x)\, \nabla\, ex'(j, j)$ **where**
ind $j = \mathbf{some}\ \mathbf{ind}\ j' : min_< \le j' \le max_<,$
funct $ex' = (\mathbf{ind}\ l, r : miss(l, r, x, P))\mathbf{bool}:$
$(min_< < l \,\Delta\, (P(p, x)\, \nabla\, ex'(p, r)$ **where** $p = prev_<(l, x, P)))\vee$
$(r < max_< \,\Delta\, (P(n, x)\, \nabla\, ex'(l, n)$ **where** $n = next_<(r, x, P))).$

This algorithm now describes a scanning of the search space "inside-out". Obviously, by choosing $min_<$ (or $max_<$) for j, algorithms **4.2** and **4.4**, respectively, are obtained. Furthermore, additionally assuming the search space to be ordered, algorithm **4.11** also comprises a "binary search algorithm" as a particular descendant.

4.6 Reuse by calculation

Instead of using a descriptive characterization of the notion of similarity as done in the previous sections, we also could use a more constructive characterization. Two specifications S and S' are defined to be similar, if there is an (operational) expression E such that $S' \equiv E(S)$ holds (or vice versa). In a language whose constructs are monotonic with respect to equivalence $\equiv$, then the result of a development starting from S' is immediately obtained by $S'_{n+1} \equiv E(S_{n+1})$, i.e., by substituting S_{n+1} for S in $S' \equiv E(S)$. If S'_{n+1} is still defined in terms of S_{n+1}, it can be made independent by a simple calculation – most notably composition via unfold-fold steps.

As an example we consider the problem:

> *Check whether an element of type* **m** *and all indices i in the non-empty, finite domain* **ind** *satisfy a predicate R.*

An obvious formalization is

funct $all = (\mathbf{m}\ x)\mathbf{bool}$:
$\forall\, \mathbf{ind}\ i : R(i, x)$.

Using the well-known relationship between existential and universal quantification, the above specification can be transformed into

funct $all = (\mathbf{m}\ x)\mathbf{bool}$:
$\neg(\exists\, \mathbf{ind}\ i : \neg R(i, x))$.

Now we may replace the existentially quantified subexpression by any of the algorithms for *ex*. Using, e.g., version **4.8** yields after obvious simplification

$R(min_<, x)\, \Delta\, \neg ex'(min_<)$ **where**
funct $ex' = (\mathbf{ind}\ l : miss(min_<, l, x, \neg R))\mathbf{bool}$:
$l < max_<\, \Delta\, (\neg R(n, x)\, \nabla\, ex'(n)$ **where** $n = next_<(l, x, \neg R))$.

In order to obtain a result independent of ex', we introduce

funct $all' = (\mathbf{ind}\ l : miss(min_<, l, x, \neg R))\mathbf{bool}$:
$\neg ex'(l)$

and calculate a new body for all' as follows:

$\neg ex'(l)$
$\equiv$ [unfold ex']
$\neg(l < max_<\, \Delta\, (\neg R(n, x)\, \nabla\, ex'(n)$ **where** $n = next_<(l, x, \neg R))$
$\equiv$ [de Morgan's law]
$\neg(l < max_<)\, \nabla\, \neg(\neg R(n, x)\, \nabla\, ex'(n)$ **where** $n = next_<(l, x, \neg R))$
$\equiv$ [simplification; de Morgan's law]
$l \geq max_<\, \nabla\, (R(n, x)\, \Delta\, \neg ex'(n)$ **where** $n = next_<(l, x, \neg R))$
$\equiv$ [fold with assertion]
$l \geq max_<\, \nabla\, (R(n, x)\, \Delta\, all'(n)$ **where** $n = next_<(l, x, \neg R))$.

Thus, after folding all' in *all* and unfolding *miss* in the definition of all', as a final result we obtain

4.12 **funct** $all = (\mathbf{m}\ x)\mathbf{bool}$:
$R(min_<, x)\,\Delta\,all'(min_<)$ **where**
funct $all' = (\mathbf{ind}\ l : \forall\,\mathbf{ind}\ k : k \leq l \Mapsto R(k,x))\mathbf{bool}$:
$l \geq max_<\,\nabla\,(R(n,x)\,\Delta\,all'(n)$ **where** $n = next_<(l, x, \neg R))$.

Of course, rather than **4.8** above, we might have used any of the other algorithms for *ex* which would have resulted in algorithms different from **4.12**. Thus, e.g., using **4.9** results in:

4.13 **funct** $all = (\mathbf{m}\ x)\mathbf{bool}$:
$all'(min_<)$ **where**
funct $all' = (\mathbf{ind}\ l : \forall\,\mathbf{ind}\ k : k < l \Mapsto R(k,x))\mathbf{bool}$:
$l > max_<\,\nabla\,(R(l,x)\,\Delta\,all'(n)$ **where** $n = next_<(l, x, \neg R))$

where the definition of all' is obtained by reusing the above calculation.

Similarly to deriving an algorithm for *all* by exploiting the relationship between $\forall$ and $\exists$, other problems that are definable as expressions over an existentially quantified formula can be treated. Typical examples are:

- **if** $\exists\,\mathbf{ind}\ i : R(i,x)$ **then some ind** $i : R(i,x)$ **else** ... **fi**
- **if** $\exists\,\mathbf{ind}\ i : R(i,x)$ **then min ind** $i : R(i,x)$ **else** ... **fi**
- $\{\mathbf{m}\ x : \exists\,\mathbf{ind}\ i : R(i,x)\}$.

Yet another kind of problem that allows reuse by calculation may informally be stated as follows:

> *Check whether an element x of type* **m** *and indices i and j in the finite, linearly ordered, non-empty domain* **ind** *satisfy a predicate Q.*

Again, a formalization is straightforward:

funct $ex_1 = (\mathbf{m}\ x)\mathbf{bool}$:
$\exists\,\mathbf{ind}\ i, j : Q(i,j,x)$.

Obviously, the body of ex_1 can be transformed into

$\exists\,\mathbf{ind}\ i : (\exists\,\mathbf{ind}\ j : Q(i,j,x))$

and, by abstraction, further into

funct $ex_1 = (\mathbf{m}\ x)\mathbf{bool}$:
$\exists\,\mathbf{ind}\ i : ex_2(i)$ **where**
funct $ex_2 = (\mathbf{ind}\ i)\mathbf{bool}$:
$\exists\,\mathbf{ind}\ j : Q(i,j,x)$.

Now we realize that for both ex_1 and ex_2 any of our generalized derivations (leading to **4.8**, **4.10**, or **4.11**) can be reused. According to the remarks made in the beginning of this section, the reuse in this particular case consists of appropriate substitutions. A subsequent calculation as in the case of the function *all* is not necessary here.

Thus, e.g. again choosing **4.8** for both ex_1 and ex_2, we obtain immediately

funct $ex_1 = (\mathbf{m}\ x)\mathbf{bool}$:
$ex_2(min_<) \nabla\, ex_1'(min_<)$ **where**
funct $ex_1' = (\mathbf{ind}\ i : miss(min_<, i, x, ex_2))\mathbf{bool}$:
$i < max_< \Delta\, (ex_2(i) \nabla\, ex_1'(next_<(i, x, Q)))$,
funct $ex_2 = (\mathbf{ind}\ i)\mathbf{bool}$:
$Q(i, min_<, x) \nabla\, ex_2'(i, min_<)$ **where**
funct $ex_2' = (\mathbf{ind}\ i, \mathbf{ind}\ j : miss(i, j, x, Q))\mathbf{bool}$:
$j < max_< \Delta\, (Q(i, j, x) \nabla\, ex_2'(i, next_<(j, x, Q)))$

which can be further simplified (by unfolding ex_2 and *miss*) into

4.14 **funct** $ex_1 = (\mathbf{m}\ x)\mathbf{bool}$:
$Q(min_<, min_<, x) \nabla\, ex_2'(min_<, min_<) \nabla\, ex_1'(min_<)$ **where**
funct $ex_1' = (\mathbf{ind}\ i : \forall\, \mathbf{ind}\ k : k \leq i \mapsto \neg \exists\, \mathbf{ind}\ j : Q(k, j, x))\mathbf{bool}$:
$i < max_< \Delta\, (Q(i, min_<, x) \nabla\, ex_2'(i, min_<) \nabla\, ex_1'(next_<(i, x, Q)))$,
funct $ex_2' = (\mathbf{ind}\ i, \mathbf{ind}\ j : \forall\, \mathbf{ind}\ k : k \leq j \mapsto \neg Q(i, k, x))\mathbf{bool}$:
$j < max_< \Delta\, (Q(i, j, x) \nabla\, ex_2'(i, next_<(j, x, Q)))$.

Note that we also might have used different algorithms for ex_1 and ex_2 which would have resulted in algorithms different from **4.14**. Note also that in algorithm **4.14** itself, we still have a freedom of choice, in particular with respect to instantiating the ordering $<$ on **ind**. Since the occurrences of this ordering in ex_1' and ex_2' are independent, even different instances of the ordering can be used. In this way, various different "search strategies" are comprised as special instances.

Also, related problems containing existential and universal quantification can be treated in exactly the same way, e.g. problems of the form

4.15 **funct** $exall = (\mathbf{m}\ x)\mathbf{bool}$:
$\exists\, \mathbf{ind}\ i : (\forall\, \mathbf{ind}\ j : Q(i, j, x))$.

If, for instance, we use the schematic derivations leading to **4.9** and **4.13**, respectively, and do simplifications similar to the ones above, such a kind of problem is solved by

4.16 **funct** $exall = (\mathbf{m}\ x)\mathbf{bool}$:
$ex'(min_<)$ **where**
funct $ex' = (\mathbf{ind}\ i : \forall\, \mathbf{ind}\ k : min_< \leq k < i \mapsto \exists\, \mathbf{ind}\ j : \neg Q(k, j, x))\mathbf{bool}$:
$i \leq max_< \Delta\, (all'(i, min_<) \nabla\, ex'(next_<(i, x)))$,
funct $all' = (\mathbf{ind}\ i, j : \forall\, \mathbf{ind}\ k : min_< \leq k < j \mapsto Q(i, k, x))\mathbf{bool}$:
$j > max_< \nabla\, (Q(i, j, x) \Delta\, all'(i, next_<(j, x)))$.

4.7 Extended example: pattern matching

So far, by various ways of reusing developments, we were able to derive a whole variety of schematic search algorithms. Now we are going to consider as a concrete problem (the essential part of) the problem of pattern matching in strings which reads, when we restrict it to non-trivial cases,

funct $occurs = (\mathbf{string}\ s, \mathbf{string}\ p : 0 < |p| \leq |s|)\mathbf{bool}$:
$\exists(\mathbf{nat}\ i : 1 \leq i \leq |s| - |p|) : \forall(\mathbf{nat}\ j : 1 \leq j \leq |p|) : s[i+j] = p[j]$.

Here, a data type **string** is assumed to be available with

- $\diamond$, denoting the empty string;
- an operation $|.|$, yielding the length of a string;
- an operation $.[.]$ for indexed access to the elements of a string; and
- an operation $.[. : .]$ to select a substring characterized by an index interval.

Obviously, *occurs* is an instance of **4.15**. Thus, any of the schematic algorithms derived from **4.15** may be used for obtaining a solution to the pattern matching problem by instantiation, additionally exploiting particular details of the concrete problem. In the following we will use **4.16** (which is allowed, since here, **ind** $\equiv$ **pnat**, and thus $next_<$ may be defined for $max_<$). In particular, we will see that in a simple way the well-known pattern matching algorithms by Boyer and Moore (cf. [BM77]), as well as by Knuth, Morris, and Pratt (cf. [KMP77]) can be derived.

4.7.1 Pattern matching according to Boyer and Moore

By instantiating **4.16**, using in ex' the ordering from low to high and in all' the ordering from high to low, we get:

funct $occurs = (\mathbf{string}\ s, \mathbf{string}\ p : 0 < |p| \leq |s|)\mathbf{bool}$:
$occ(1)$ **where**
funct $occ = (\mathbf{nat}\ i : \forall\,\mathbf{nat}\ k : 1 \leq k < i \mapsto \exists\,\mathbf{ind}\ j : s[k+j] \neq p[k]))\mathbf{bool}$:
$i \leq |s| - |p| \,\Delta\, (pm'(i, |p|) \,\nabla\, occ(next_<(i)))$,
funct $pm' = (\mathbf{ind}\ i, j : \forall\,\mathbf{ind}\ k : j < k \leq |p| \mapsto s[i+k] = p[k])\mathbf{bool}$:
$j < 1 \,\nabla\, (s[i+j] = p[j] \,\Delta\, pm'(i, prev_<(j)))$

(where, additionally, in $next_<$ and $prev_<$ the constant global parameters have been suppressed for the sake of better readability).

This is already the essence of the pattern matching algorithm by Boyer and Moore, if suitable definitions for $next_<$ and $prev_<$ are given. In order to come somewhat closer to the form of the algorithm as given e.g. in [PS90] a few "cosmetic" transformations are needed. The following property (cf. [PS90]) is used in these transformations:

$$((a \,\Delta\, b) \,\nabla\, c) \equiv ((a \,\Delta\, (b \,\nabla\, c)) \,\nabla\, (\neg a \,\Delta\, c)). \tag{4.17}$$

1. Introduction of an auxiliary function

   funct $pm'' = (\mathbf{ind}\ i,j : \forall\,\mathbf{ind}\ k : j < k \leq |p| \mapsto s[i+k] = p[k])\mathbf{bool}$:
   $pm'(i,j) \nabla\, occ(next_<(i))$.

2. In *occ*:

   $i \leq |s| - |p| \,\Delta\, (pm'(i,|p|) \nabla\, occ(next_<(i)))$
   $\equiv$ [unfold pm' ; simplification]
   $i \leq |s| - |p| \,\Delta\, ((s[i+|p|] = p[|p|] \,\Delta\, pm'(i, prev_<(|p|))) \nabla\, occ(next_<(i)))$
   $\equiv$ [property 4.17]
   $i \leq |s| - |p| \,\Delta$
   $((s[i+|p|] = p[|p|] \,\Delta\, (pm'(i, prev_<(|p|)) \nabla\, occ(next_<(i)))) \nabla$
   $(s[i+|p|] \neq p[|p|] \,\Delta\, occ(next_<(i))))$
   $\equiv$ [fold pm'']
   $i \leq |s| - |p| \,\Delta$
   $((s[i+|p|] = p[|p|] \,\Delta\, pm''(i, prev_<(|p|))) \nabla$
   $(s[i+|p|] \neq p[|p|] \,\Delta\, occ(next_<(i))))$.

3. In pm'':

   $pm'(i,j) \nabla\, occ(next_<(i))$
   $\equiv$ [unfold pm']
   $(j < 1 \nabla\, (s[i+j] = p[j] \,\Delta\, pm'(i, prev_<(j)))) \nabla\, occ(next_<(i))$,
   $\equiv$ [associativity of ∇ ; property 4.17]
   $j < 1 \nabla$
   $(s[i+j] = p[j] \,\Delta\, (pm'(i, prev_<(j)) \nabla\, occ(next_<(i)))) \nabla$
   $(s[i+j] \neq p[j] \,\Delta\, occ(next_<(i)))$
   $\equiv$ [fold pm'']
   $j < 1 \nabla$
   $(s[i+j] = p[j] \,\Delta\, pm''(i, prev_<(j))) \nabla$
   $(s[i+j] \neq p[j] \,\Delta\, occ(next_<(i)))$.

Putting things together, we have

funct $occurs = (\mathbf{string}\ s, \mathbf{string}\ p : 0 < |p| \leq |s|)\mathbf{bool}$:
  $occ(1)$ **where**
  funct $occ = (\mathbf{nat}\ i : \forall\,\mathbf{nat}\ k : 1 \leq k < i \mapsto \exists\,\mathbf{ind}\ j : s[k+j] \neq p[k])\mathbf{bool}$:
    $i \leq |s| - |p| \,\Delta$
      $((s[i+|p|] = p[|p|] \,\Delta\, pm''(i, prev_<(|p|))) \nabla$
      $(s[i+|p|] \neq p[|p|] \,\Delta\, occ(next_<(i)))),$

funct $pm'' = (\textbf{ind}\ i,j : \forall\,\textbf{ind}\ k : j < k \leq |p| \mapsto s[i+k] = p[k])\textbf{bool}$:
$j < 1\,\nabla$
$(s[i+j] = p[j]\,\Delta\, pm''(i, prev_<(j)))\,\nabla$
$(s[i+j] \neq p[j]\,\Delta\, occ(next_<(i)))$.

This is essentially the version of Boyer and Moore's algorithm given in [PS90], if we additionally instantiate $prev_<(j)$ by $j-1$ and use for $next_<$ (according to our schematic definition)

4.18 $next_<(i) \equiv \textbf{some nat}\ k : i < k\,\Delta\,\forall\,\textbf{ind}\ k' : i \leq k' < k \mapsto miss(i,k')$ **where**
$miss(i,j) \equiv \forall\,\textbf{ind}\ k : i \leq k \leq j \mapsto \exists\,(\textbf{ind}\ k' : 1 \leq k' \leq |p|) : s[k+k'] \neq p[k']$,

from which a good operational approximation can be calculated by taking into account the respective context (cf. [PS90]).

4.7.2 Pattern matching according to Knuth, Morris, and Pratt

In a similar way the algorithm by Knuth, Morris, and Pratt can be derived, if we use **4.16** and choose orderings from low to high in both ex' and all', but maintain the flexibility with respect to $next_<$ by using two different operations for $next_<$ in ex' and all', respectively. By simply instantiating **4.16** we get

funct $occurs = (\textbf{string}\ s, \textbf{string}\ p : 0 < |p| \leq |s|)\textbf{bool}$:
$kmp(1)$ **where**
funct $kmp = (\textbf{nat}\ i : \forall\,\textbf{nat}\ k : 1 \leq k < i \mapsto \exists\,\textbf{ind}\ j : s[k+j] \neq p[k]))\textbf{bool}$:
$i \leq |s| - |p|\,\Delta\,(pm'(i,1)\,\nabla\, kmp(next_<(i)))$,
funct $pm' = (\textbf{ind}\ i,j : \forall\,\textbf{ind}\ k : 1 \leq k \leq j \mapsto s[i+k] = p[k])\textbf{bool}$:
$j > |p|\,\nabla\,(s[i+j] = p[j]\,\Delta\, pm'(i, ne_<(j)))$.

Again, in order to come closer to a form of the algorithm as known from the literature, a few "cosmetic" transformations are needed:

1. Introduction of an auxiliary function

   funct $pm'' = (\textbf{ind}\ i,j : \forall\,\textbf{ind}\ k : 1 \leq k \leq j \mapsto s[i+k] = p[k])\textbf{bool}$:
   $pm'(i,j)\,\nabla\, kmp(next_<(i))$.

2. In kmp:

   $i \leq |s| - |p|\,\Delta\,(pm'(i,1)\,\nabla\, kmp(next_<(i)))$
   $\equiv$ [fold pm'']
   $i \leq |s| - |p|\,\Delta\, pm''(i,1)$.

3. In pm'':

$pm'(i,j) \nabla kmp(next_<(i))$
$\quad \equiv$ [unfold pm']
$(j > |p| \nabla (s[i+j] = p[j] \Delta pm'(i, ne_<(j)))) \nabla kmp(next_<(i))$
$\quad \equiv$ [associativity of ∇; property 4.17]
$j > |p| \nabla$
$(s[i+j] = p[j] \Delta (pm'(i, ne_<(j)) \nabla kmp(next_<(i)))) \nabla$
$(s[i+j] \neq p[j] \Delta kmp(next_<(i)))$
$\quad \equiv$ [fold pm'']
$j > |p| \nabla$
$(s[i+j] = p[j] \Delta pm''(i, ne_<(j))) \nabla$
$(s[i+j] \neq p[j] \Delta kmp(next_<(i)))$.

4. In *occurs*:

$kmp(1)$
$\quad \equiv$ [unfold kmp]
$1 \leq |s| - |p| \Delta pm''(1,1)$.

Together, we have a version of the algorithm by Knuth, Morris, and Pratt which is essentially the algorithm (5.5′) in [PV91] (if we additionally define $ne_<(j)$ by $j+1$, and $next_<$ similar to **4.18**:

funct $occurs = (\textbf{string}\ s, \textbf{string}\ p : 0 < |p| \leq |s|)\textbf{bool}$:
$1 \leq |s| - |p| \Delta pm''(1,1)$ **where**
funct $pm'' = (\textbf{ind}\ i,j : \forall\, \textbf{ind}\ k : 1 \leq k \leq j \mapsto s[i+k] = p[k])\textbf{bool}$:
$j > |p| \nabla$
$(s[i+j] = p[j] \Delta pm''(i, j+1)) \nabla$
$(s[i+j] \neq p[j] \Delta next_<(i) \leq |s| - |p| \Delta pm''(next_<(i), 1))$.

4.7.3 Comparison with other derivations of pattern matching algorithms

In the literature several formal derivations of pattern matching algorithms can be found (cf., e.g., [BGJ89, Dij76a, Mor90, PS90, PV91, Pep91, Völ92, vdW89]). Apart from technical details, our derivations above differ from most of these in two essential respects:

- By looking at pattern matching as an instance of a more general search problem (essentially a fast, two-level, linear search), the overall structure of the developments is much clearer, since the general aspects (i.e., fast linear search) and the particular ones (e.g., orderings used in searching) are well separated.

- By our derivations the commonalities as well as the differences of the Boyer-Moore algorithm and the Knuth-Morris-Pratt algorithm are formally exhibited which allows simple comparisons.

4.8 Final remarks

One of the crucial aspects with respect to reusability of transformational developments is an appropriate definition of the notion "similar". We have seen that, depending on different definitions of similarity, different kinds of reuse are possible, leading to a whole family of substantially different algorithms. We have also seen, that most profit can be gained by reuse, if the respective specifications are similar in the constructive sense of section 4.6, since then the new development can be calculated from the available one. A thorough theoretical investigation of several formal definitions of the notion "similarity" and their properties are the topic of ongoing research.

In our considerations it also should have become obvious that reusability of transformational developments strongly depends on the way how transformational developments are formally expressed. Thus, for instance, references to subexpressions of specifications within transformation steps are certainly an obstacle for reusability. In order to avoid this difficulty, we have deliberately used an informal description of transformations. In the long run, however, it will be necessary to have a formalism for describing the transformation process which is as independent as possible of concrete specifications.

Furthermore, it also should have become apparent that the various ways of reusability reflect idealistic views. In practice one will encounter cases that allow "partial" reuse and require an additional adaptation by hand.

Thus, summing up, it seems to be obvious that the field of reusability of transformational developments needs further investigation and further experience through appropriate case studies. "Automatic replay" as often advertized in the literature is still a dream.

Chapter 5

Intersections of sets and bags of extended substructures – a class of problems

5.1 Introduction

An important part of research in transformational programming has concentrated on the development of general techniques for handling large problem domains. Examples of this are the category-theoretical foundation of BMF by Malcolm [Mal90] and others, Smith and Lowry's algorithm theories and design tactics [SL90], and the experiments in reuse of transformational developments by our own group in Nijmegen [PV91, PB91].

In the present chapter, we consider a class of problems which can be viewed as a generalization of the pattern matching problem, which is currently a popular subject in the area of transformational programming (cf. [BGJ89, PS90, Völ92, PB91]). Our long term goal is to provide solutions to a large class of generalized pattern matching problems. This note is concerned mainly with the identification and description of the important concepts of the general description of this problem class. Although we present some ideas about solving these problems, much research still needs to be done on this subject.

In our view the essence of pattern matching problems is: the intersection of a set and a bag consisting of all extended substructures of a structured object.

In this section, we illustrate some of the important concepts by stepwise generalization of the string matching problem. Section 5.2 gives more detail about the notation used. In section 5.3, we briefly present bags with their corresponding operations and notations. Then in sections 5.4 and 5.5 we discuss substructures and extended substructures. The ISBES class of problems is formally defined in section 5.6. The remaining sections contain examples of ISBES specifications and applicable transformations and strategies.

Informally, the string matching problem can be stated as follows:

> *given strings s and p, determine whether p occurs in s.*

A formal definition is

$$occurs\ (p, s) = \exists\, x, y : s = x \# p \# y$$

where $\#$ denotes concatenation.

A first generalization is to consider a set of patterns P. Then we have

$$occur\ (P, s) = \exists\, p \in P : \exists\, x, y : s = x \# p \# y$$

. Of course, $occurs\ (p, s) = occur\ (\{p\}, s)$.

A simple abstraction introduces the concept of substrings (or *segments*, cf. [BGJ89]). Then we have

$$\begin{aligned} occur\ (P, s) &= \exists\, p \in P : p \in segs\ (s) \\ segs\ (s) &= \{\!|\ p |\ \exists\, x, y : s = x \# p \# y |\!\} \end{aligned}$$

where $\{\!|\ |\!\}$denotes bag comprehension (cf. section 5.3). But then also

$$occur\ (P, s) = (P \cap segs\ (s) \neq \emptyset)$$

where the operator $\cap$ is used to denote a particular kind of intersection, viz. that between a set and a bag which eliminates from the bag all elements that do not occur in the set.

In the above example, sets would suffice (instead of bags), since we are not interested in the number of occurrences of the pattern string. The usual definitions of *segs* [BGJ89] yield *lists* of substrings. This usually leads to design decisions made already in the *specification* stage (viz. whether the segments are defined as the prefixes of all suffixes, or vice versa). Hence our insistence on bags: the number of identical substructures may very well be relevant, but an order on the substructures is usually not imposed before *implementations* are considered.

So, the string matching problem can be described as follows:

- consider the bag D consisting of all substructures (viz. segments) of the given structure s,
- intersect D with the given set $\{p\}$,
- the result is determined by the emptiness of the resulting bag.

Usually, in pattern matching problems one wants to know not only *that* the pattern occurs, but also "where" it occurs. This implies that to each substructure some information must be added which relates the substructure to the original structured object, resulting in what we will call *extended substructures* (cf. section 5.5). For strings, obviously *indices* constitute the required extra information.

The string matching problem with indices can be described as follows:

> *given strings s and p, determine whether p occurs in s, and if so, give indices i and j such that $p = s[i..j]$*[1] *$(0 \leq i, j \leq |s|)$.*

[1] For indexing, or *slicing*, we use the convention of not numbering the elements themselves, but the "borders" between them [Mei86]. This combines the advantages of numbering the elements from 0 and from 1 onwards: no 1's have to be added or subtracted, and the difference between highest and lowest index equals the length of the list.

A number of generalizations lead from the previous problem to this one:

- instead of the bag *segs* (s) consider the bag

$$segsi\ (s) = \{\![\ < p, < i, j >> \mid p = s[i..j]]\!\}$$

  (which actually is a set, see section 5.5 for a discussion on this issue) where $<>$ is the tuple former;

- in order for the pattern string to match the type of the indexed segments, one needs to consider all possible combinations of the pattern p and two indices i and j such that, for some string s, $p = s[i..j]$ might hold. So, as the pattern set we take

$$\{< p, < n, n + |p| >> \mid n \geq 0\}.$$

Then a formal specification of this problem is:

$$\begin{array}{lll} occuri\ (p, s) = & \textbf{if } B = \emptyset & \textbf{then false} \\ & & \textbf{else some } \ < i, j >:< p, < i, j >> \ \in B \textbf{ fi} \\ & \textbf{where } B & = \{< p, < n, n + |p| >> \mid n \geq 0\} \cap segsi(s). \end{array}$$

5.2 About notation

We will present *specifications* for a large class of problems, and thus it is important to state the problems in such a way that all possible solutions can be derived from them. Thus, we mainly use *descriptive* (as opposed to *operational*) notation. The following descriptive constructs, mostly known from CIP-L [BBB+85], will be used:

- **some** $x : P(x)$, denoting an arbitrary value satisfying P;
- **that** $x : P(x)$, denoting the unique value satisfying P;
- set comprehension $\{x \mid P(x)\}$;
- a limited form of bag comprehension (cf. section 5.3);
- existentially and universally quantified predicates.

Furthermore, we also use some of the primitive operations of Squiggol (or BMF [Bir87, Mee86, STO89]) on finite lists, bags and sets (part of the *Boom-hierarchy* [Mee89a]). The meaning of these operators will be given in a polymorphic way. ε denotes the empty structure, τ denotes the singleton structure former, and $\sqcup$ denotes the binary structure former (e.g. concatenation, union). These types are defined as follows:

$$\frac{}{\varepsilon : Struct(\mathbf{t})} \qquad \frac{x : \mathbf{t}}{\tau(x) : Struct(\mathbf{t})} \qquad \frac{a, b : Struct(\mathbf{t})}{a \sqcup b : Struct(\mathbf{t})}$$

ε is the unit of $\sqcup$. If $\sqcup$ is associative, we have the type *List* of finite lists. If it is also

commutative, we get the type *Bag* of finite bags. When, additionally, $\sqcup$ is idempotent, the type *Set* of finite sets is obtained. In concrete cases, different symbols may be used for the operations.

Inductive definitions on *Struct* can be given in two ways: either by defining an operation for ε, $\tau(x)$, and $a \sqcup b$ (the "*join*-view"), or by defining it for ε and $\tau(x) \sqcup a$ (the "*cons*-view"). We use the most convenient view in each case.

For a, b of type *Struct*($\mathbf{t}_1$) and c, d of type *Struct*($\mathbf{t}_2$), we have operations:

- $p \lhd a$ (filter), denoting the structure containing all elements, as many copies (for list/bag) and in the same order (for list) of a that satisfy predicate p:

$$\begin{array}{rcl} p \lhd \varepsilon & = & \varepsilon \\ p \lhd (\tau(x) \sqcup a) & = & \textbf{if } p(x) \textbf{ then } \tau(x) \sqcup (p \lhd a) \\ & & \textbf{else } p \lhd a \textbf{ fi} \end{array}$$

- $f * a$ (map), denoting the structure where every element of a is replaced by its f-value:

$$\begin{array}{rcl} f * \varepsilon & = & \varepsilon \\ f * \tau(x) & = & \tau(f\ x) \\ f * (a \sqcup b) & = & (f * a) \sqcup (f * b) \end{array}$$

- $\oplus/a$ (reduce), for operator $\oplus$ of type $\mathbf{t}_1 \times \mathbf{t}_1 \to \mathbf{t}_1$. $\oplus$ must be associative, commutative (for bags and sets) and idempotent (for sets). It denotes the value obtained by "putting $\oplus$ between all elements of a". Thus, for nonempty a we have:

$$\begin{array}{rcl} \oplus/\tau(y) & = & y \\ \oplus/(a \sqcup b) & = & (\oplus/a) \oplus (\oplus/b) \text{ for nonempty } a, b \end{array}$$

  If $\oplus$ has a unit element $1_\oplus$, additionally we have

$$\oplus/\varepsilon = 1_\oplus \tag{5.1}$$

  In that case, the law for $\sqcup$ holds for empty a or b as well.

- $a \mathsf{X}_\oplus\ c$ (cross), for operator $\oplus$ of type $\mathbf{t}_1 \times \mathbf{t}_2 \to \mathbf{t}_3$, returning an object of type *Struct*($\mathbf{t}_3$). Informally, $a \mathsf{X}_\oplus\ c$ denotes the structure containing for all possible combinations x, y from a and c, respectively, the values $x \oplus y$. Formally, X is defined by:

$$\begin{array}{rcl} a \mathsf{X}_\oplus\ \varepsilon & = & \varepsilon \\ a \mathsf{X}_\oplus\ \tau(x) & = & (\oplus x) * a \\ a \mathsf{X}_\oplus\ (c \sqcup d) & = & (a \mathsf{X}_\oplus\ c) \sqcup (a \mathsf{X}_\oplus\ d) \end{array}$$

In the sequel, we freely mix the CIP-L and Squiggol styles of notation, using the notations we find most convenient in each case. On one hand, recursive functions with brackets for application will be used as in CIP-L. On the other hand, application of Squiggol functions will often be denoted by juxtaposition. As in the definition of $\mathsf{X}_{\oplus}$, *sectioning* will be used, i.e. we write $(x +)$ for f, where $f(y) = (x + y)$ etc. Types will not be denoted, in general, because they can often be deduced from the context, and because some expressions are polymorphic.

In order to avoid confusion, () will be used for application and disambiguation, not for forming tuples. Tuples are denoted using $<\ >$. Their components can be accessed by projection functions π_i. For functions f and g, the function f, g is defined by: $(f, g)x = < f\ x, g\ x >$.

5.3 More notation: Bags

The notation we use for sets is, of course, the conventional one. For bags, however, a widely used standard notation does not seem to exist.

The type of bags over a type $\mathbf{t}$, denoted as $Bag(\mathbf{t})$, is a particular instance of the type *Struct* defined before. In order to introduce the particular notations in this case, we give a complete definition:

$$\frac{}{\emptyset : Bag(\mathbf{t})} \qquad \frac{x : \mathbf{t}}{\{\![\ x\ ]\!] : Bag(\mathbf{t})} \qquad \frac{y, z : Bag(\mathbf{t})}{y \uplus z : Bag(\mathbf{t})}$$

As stated in the discussion on *Struct*, $\uplus$ is associative and commutative, and has neutral element $\emptyset$.

We assume the usual definitions for operations $\in$ (element), # (number of elements) and $-_1 x$ (elimination of one occurrence of x).

Three more operations that will be used are:

- $B_1 \cap B_2$ (intersection of *two bags*) with laws

$$\begin{array}{rcl} B \cap \emptyset & = & \emptyset \\ B_1 \cap (\{\![\ x\ ]\!] \uplus B_2) & = & \textbf{if } x \in B_1 \\ & & \textbf{then } ((B_1 -_1 x) \cap B_2) \uplus \{\![\ x\ ]\!] \\ & & \textbf{else } B_1 \cap B_2 \textbf{ fi}. \end{array}$$

- a limited form of bag comprehension $\{\![\ x \mid P(x)]\!]$, defined only in the cases below:

$$\begin{array}{rcl} \{\![\ x \mid x \in B \wedge P(x)]\!] & = & P \lhd B \quad \text{for bags } B \\ \{\![\ x \mid \exists y : Q(y, x)]\!] & = & \uplus_y(\textbf{if } \exists\, x : Q(y, x) \ \textbf{ then } \{\![\ \textbf{that } x : Q(y, x)]\!] \\ & & \qquad\qquad\qquad\qquad \textbf{else } \emptyset \textbf{ fi}) \\ & & \text{if} \quad \forall x, y, z : (Q(y, x) \wedge Q(y, z)) \Rightarrow x = z. \end{array}$$

  This means that for each y of the appropriate type the unique value x such that $Q(y, x)$ holds is included in $\{\![\ x \mid \exists\, y : Q(y, x)]\!]$. Such a value x occurs exactly as often as the

number of different values y such that $Q(y, x)$ holds. If, for some y, there are multiple x such that $Q(y, x)$, the bag comprehension is not defined.

The general notion of bag comprehension is hard to define formally in such a way that it coincides with the intuition *and* that it is monotonic w.r.t. the predicate (i.e., $\{\!\![\ x \mid P(x)\ ]\!\!\} \subseteq \{\!\![\ x \mid Q(x)\ ]\!\!\}$ whenever $P \Rightarrow Q$). A discussion of this problem may be found in [Boi91a].

- $S \cap\!\!\!\!\cap B$ (intersection of set and bag, cf. section 5.1) which returns a *bag*, with laws

$$\begin{aligned} S \cap\!\!\!\!\cap \emptyset &= \emptyset \\ S \cap\!\!\!\!\cap \{\!\![\ x\ ]\!\!\} &= \textbf{if } x \in S \textbf{ then } \{\!\![\ x\ ]\!\!\} \textbf{ else } \emptyset \textbf{ fi} \\ S \cap\!\!\!\!\cap (a \uplus b) &= (S \cap\!\!\!\!\cap a) \uplus (S \cap\!\!\!\!\cap b) \end{aligned}$$

or, more concisely $S \cap\!\!\!\!\cap B = (\in S) \lhd B$.

5.4 Substructures

By "structured types" we mean algebraic types that include, in some sense, information from one or more underlying "base types". Examples are sets, records, trees and arrays. As in the previous sections, structured types will be introduced by giving a number of introduction rules and some laws (equivalences) on the terms of the type. Restrictions on the type (e.g. ordered trees) are assumed to be modeled by restrictions in the introduction rules[2].

For objects of a structured type, often *substructures* can be identified: objects of the structured type that include only part of the "information" (from the base types) and structure. Examples of this are subsets, subtrees and "slices" (of arrays).

We define a set of substructure forming functions for a data type $S(\mathbf{t})$. This set is chosen in such a way that any substructure of an object x contains only "information" that is present in x, and that it does not duplicate information present in x. This, at first sight, somewhat arbitrary decision (ordering *may* be changed, number of occurrences may not) is motivated by our intuition (note that, for bags, this is exactly what one would expect).

Note that, without loss of generality, the introduction rules for $S(\mathbf{t})$ may be represented by a set of constructor functions, indexed by some labeling set I, which first take all their arguments of type $S(\mathbf{t})$ and then all their arguments of type $\mathbf{t}$. $\vec{\alpha}_m$ denotes $\alpha_1, \ldots, \alpha_m$.

Definition 5.2 Let the collection of all constructor functions leading into $S(\mathbf{t})$ be

$$C = \{c_i : S(\mathbf{t})^{m_i} \times \mathbf{t}^{n_i} \to S(\mathbf{t}) \mid i \in I\}.$$

The substructure forming functions (sff's) for a structured type $S(\mathbf{t})$ are defined as the smallest set of (partial) functions $(\in S(\mathbf{t}) \to S(\mathbf{t}))$ satisfying the conditions below.

[2]This requires some "compositionality" of the restrictions – e.g. lists of a prime length cannot be easily modeled this way, but for such types it does not make much sense to talk about substructures anyway.

1. The identity function on $S(\mathbf{t})$ is a sff.

2. If g is a sff, then for all $i \in I$, $1 \leq j \leq m_i$ such that g is defined for α_j, f defined by

$$f(c_i(\vec{\alpha}_{m_i}, \vec{\beta}_{n_i})) = g(\alpha_j)$$

   is a sff. Note that α_j are substructures by taking g to be the identity function.

3. For all $i \in I$ and $j \in I$ and sff's $g_1, \ldots, g_{m_j}$, f defined by

$$f(c_i(\vec{\alpha}_{m_i}, \vec{\beta}_{n_i})) = c_j(g_1(\alpha_{p(1)}), \ldots, g_{m_j}(\alpha_{p(m_j)}), \beta_{q(1)}, \ldots, \beta_{q(n_j)})$$

   where p is an injective mapping from $\{1, \ldots, m_j\}$ to $\{1, \ldots, m_i\}$ and q is an injective mapping from $\{1, \ldots, n_j\}$ to $\{1, \ldots, n_i\}$, is a sff, which is only defined on terms $c_i(\vec{\alpha}_{m_i}, \vec{\beta}_{n_i})$ where the functions g_i are defined on $\alpha_{p(i)}$ and where c_j is defined. The requirement that p and q be injective mappings implies that none of the α_k or β_k is "chosen twice" by c_j, and that $m_j \leq m_i$ and $n_j \leq n_i$.

   The above rule implies that a substructure may be constructed by applying a different function of the type to a subset of (substructures of) the subterms of a term.

Definition 5.3 For a given (sub)set of sff's F, the *bag of substructures* D of a structured object x is defined by:

$$D(x) = (.x) * (\text{DEF}_x) \lhd F$$

where . denotes application and predicate DEF_x denotes the definedness of a function for argument x.

Any definition of a sensible kind of substructures can be expressed by way of a particular subset of the substructure forming functions defined above.

For example, consider the type $\mathit{Tree}(\mathbf{t})$ of binary trees over a type $\mathbf{t}$, defined by:

$$\frac{}{\varepsilon : \mathit{Tree}(\mathbf{t})} \qquad \frac{y, z : \mathit{Tree}(\mathbf{t}), x : \mathbf{t}}{y \swarrow x \searrow z : \mathit{Tree}(\mathbf{t})}$$

The (infinite) set of substructure forming functions one takes to create all subtrees of a tree is the smallest set F satisfying

1. $(I(t) = t) \in F$ (identity function),

2. $f \in F \Rightarrow (g(t_1 \swarrow x \searrow t_2) = f(t_1)) \in F$,

3. $f \in F \Rightarrow (g(t_1 \swarrow x \searrow t_2) = f(t_2)) \in F$.

By taking *all* allowed substructure forming functions, one would get all trees where the bag of leaves forms a subbag of the bag of leaves of the original tree.

5.5 Extended substructures

In section 5.1 we considered two example problems. In the second one (string matching with indices), we needed more than just the substructures of the string: the indices denoting the positions of the substrings were also needed. In a more general sense, we distinguish between substructures by adding a kind of *labels* to them, that give some "information" about the "position" of the substructures. Any particular notion of extended substructures can be defined using a notion of substructures and a function ϕ which, given an object S and a label E, returns the bag of all substructures of S that get labeled with E.

Definition 5.4 *Given a function* $\phi : S(\mathbf{t}) \times L \to Bag(S(\mathbf{t}))$, *an* extended substructure *of a structured object* S *is a pair* $< S', E >: S(\mathbf{t}) \times L$ *such that* S' *is a substructure of* S, *and* $S' \in \phi(S, E)$.

A condition for such a ϕ to be useful is that $\uplus_E \phi(S, E) = D(S)$, i.e. all substructures as in the previous section can be extended. Note that the bag containing all extended substructures is a set whenever the range of ϕ consist of singleton bags and empty bags only, which is often the case in practice. If $\phi(S, E)$ contains more than one element, this means that E gives "incomplete information" about the "position" of the elements of $\phi(S, E)$ in S.

The bag of all extended substructures D^+ can now be defined as follows:

$$D^+(S) = \biguplus_E (\phi(S, E) \ \mathsf{X}_{<,>} \ \{\!|\, E \,|\!\}).$$

This yields an implicit characterization of extended substructure forming functions; usually an obvious definition of the extended substructure forming functions can be given based on the sff's.

For the subtree example above, one can imagine at least two kinds of labels: the path leading to (the root of) the subtree, or the depth of the (root of) the subtree.

If we define paths as lists of **L** and **R**, for subtrees with paths, the set of all extended substructure forming functions can be defined to be the smallest set F satisfying

1. $(f(t) =< t, \varepsilon >) \in F$ (identity function),
2. $(f(t_1) =< t', p >) \in F \Rightarrow (g(t_1 \swarrow x \searrow t_2) =< t', [\mathbf{L}] +\!\!+ p >) \in F$,
3. $(f(t_2) =< t', p >) \in F \Rightarrow (g(t_1 \swarrow x \searrow t_2) =< t', [\mathbf{R}] +\!\!+ p >) \in F$.

Obviously, the position of a subtree in a tree is uniquely determined by its path, and thus the corresponding function ϕ has at most singletons in its domain:

$$\begin{aligned} \phi(t, \varepsilon) &= \{\!|\, t \,|\!\} \\ p \neq \varepsilon \Rightarrow \phi(\varepsilon, p) &= \emptyset \\ \phi(t_1 \swarrow a \searrow t_2, [\mathbf{L}] +\!\!+ p) &= \phi(t_1, p) \\ \phi(t_1 \swarrow a \searrow t_2, [\mathbf{R}] +\!\!+ p) &= \phi(t_2, p). \end{aligned}$$

If we take subtrees with depth, the set of all extended substructure forming functions can be defined to be the smallest set F satisfying

1. $(f(t) =< t, 0 >) \in F$ (identity function),
2. $(f(t_1) =< t', n >) \in F \Rightarrow (g(t_1 \swarrow x \searrow t_2) =< t', n+1 >) \in F$,
3. $(f(t_2) =< t', n >) \in F \Rightarrow (g(t_1 \swarrow x \searrow t_2) =< t', n+1 >) \in F$.

Here, the label does not uniquely determine the position of a subtree in a tree. This can be seen from the corresponding ϕ, defined by:

$$\begin{aligned} \phi(t,0) &= \{\!|\, t \,|\!\} \\ n > 0 \Rightarrow \phi(\varepsilon, n) &= \emptyset \\ n > 0 \Rightarrow \phi(t_1 \swarrow x \searrow t_2, n) &= \phi(t_1, n-1) \uplus \phi(t_2, n-1). \end{aligned}$$

A similar example may be found in [Spi90], where subterms (subtrees) are defined together with their corresponding "paths".

5.6 Formulation of the problem class *ISBES*

The problem class *ISBES* (Intersection of a Set and a Bag of Extended Substructures) can be specified as follows:

Given

- an object O of the structured type $S(\mathbf{t})$,
- a set P of objects of the type $S^+(\mathbf{t})$, where $S^+(\mathbf{t})$ is the type of tuples from $S(\mathbf{t})$ and some other type. P often depends on other input parameters of the problem. This is represented by defining P to be a function of these parameters.
- a function D^+ (decompose) which yields the bag of extended substructures (of type $S^+(\mathbf{t})$) of an object of type $S(\mathbf{t})$,
- a function R (result) which takes a bag of elements of type $S^+(\mathbf{t})$ and returns the "answer" to the problem,

compute

$$R(P \cap\!\!\!\cap D^+(O)).$$

5.7 Examples of *ISBES*

1. **Standard string matching**
   Assuming we want to know whether p occurs in a string, we have:

$$\begin{aligned} S(\mathbf{t}) &= \mathit{List}(\mathbf{t}) \\ P(p) &= \{p\} \\ D^+ &= \mathit{segs} \\ R &= (\neq \emptyset) \end{aligned}$$

If we want to determine how often p occurs in a string, we have instead for R:

$$R = \#$$

Note that here, the *bag* of substructures is essential.

2. **Standard string matching, leftmost occurrence indexed**
Assuming we want to know where p occurs first in a string, we have:

$$\begin{aligned} S(\mathbf{t}) &= List(\mathbf{t}) \\ P(p) &= \{< p, < n, n + |p| >> \mid n \geq 0\} \\ D^+ &= segsi \\ R &= min/ \cdot (\pi_1 \cdot \pi_2)* \end{aligned}$$

3. **Preprocessing for the Knuth-Morris-Pratt string matching algorithm**
The Knuth-Morris-Pratt string matching algorithm [KMP77] has been the subject of a number of recent studies in transformational programming [BGJ89, Völ92, PV91]. The algorithm requires the construction by preprocessing of a table δ, which contains for each prefix x of the pattern p the length of its longest proper suffix s such that s is a prefix of x.

Note that the structured object p is also used to determine the pattern set P.

$$\begin{aligned} S(\mathbf{t}) &= List(\mathbf{t}) \\ P &= \{< x, < k, l >> \mid p[0 : l - k] = x \wedge 1 \leq k < l \leq |p|\} \\ D^+(p) &= segsi(p) \\ R(B) &= \textbf{some map } \delta : \forall i : \delta[i] = max/\{i - k \mid \exists x :< x, < k, i >> \in B\} \end{aligned}$$

The set P gives all prefixes of the pattern p, with as extra information all possible index pairs k, l where the prefix *might* occur elsewhere in the string.

A definition of the type **map** is not given here; the meaning of the above should be obvious.

4. **Pattern matching with variables**
Let *subst* be the set of all possible ground substitutions for the variables, and let the application of a substitution σ to an expression E be denoted by $E\sigma$. Suppose we want to determine whether any substitution instance of the pattern p can be found in a string. Then we have:

$$\begin{aligned} P(p) &= \{p\sigma \mid \sigma \in subst\} \\ S(\mathbf{t}) &= List(\mathbf{t}) \\ D^+ &= segs \\ R &= (\neq \emptyset) \end{aligned}$$

5. **Finding a value in a tree**
The function $Paths : \alpha \times Tree(\alpha) \to Bag(Path)$, where $Path$ denotes lists of **L** and **R**, is to return all paths in a tree t to a node a. In ISBES-form, this can be specified by:

$$\begin{aligned} S(\mathbf{t}) &= Tree(\mathbf{t}) \\ P(a) &= \{< t_1 \swarrow a \searrow t_2, p > \mid \mathbf{true}\} \\ D^+ &= Subt \\ R &= \pi_2 * \end{aligned}$$

where $Subt$ is the function that returns, for a tree, all subtrees with their paths as in section 5.5. An operational version (obtained by trivial rewriting) is:

$$\begin{aligned} Subt(\varepsilon) &= \{\!\![< \varepsilon, \varepsilon >]\!\!\} \\ Subt(t_1 \swarrow a \searrow t_2) &= \{\!\![< t_1 \swarrow a \searrow t_2, \varepsilon >]\!\!\} \uplus \\ &\quad (\mathbf{L}\odot) * Subt(t_1) \uplus (\mathbf{R}\odot) * Subt(t_2) \\ &\quad \text{where} \quad x\odot < t, p >=< t, [x] \# p > \end{aligned}$$

Thus we have

$$Paths(a, t) = \pi_2 * (P \cap Subt(t))$$

As an example of a calculation with ISBES-style problems, we calculate a recursive definition of $Paths$:

$Paths(a, \varepsilon)$
$\quad =\{$ definition $Paths$, P, $Subt$ $\}$
$\pi_2 * (\{< t_1 \swarrow a \searrow t_2, p > \mid \mathbf{true}\} \cap \{\!\![< \varepsilon, \varepsilon >]\!\!\})$
$\quad =\{$ definition $\cap$ $\}$
$\pi_2 * \emptyset$
$\quad =\{$ definition $*$ $\}$
$\emptyset$.

$Paths(a, t_3 \swarrow b \searrow t_4)$
$\quad =\{$ definitions $Paths$, $Subt$ $\}$
$\pi_2 * (P \cap (\{\!\![< t_3 \swarrow b \searrow t_4, \varepsilon >]\!\!\} \uplus (\mathbf{L}\odot) * Subt(t_3) \uplus (\mathbf{R}\odot) * Subt(t_4))$
$\quad =\{$ $\cap$ distributes over $\uplus$ $\}$
$\pi_2 * ((P \cap \{\!\![< t_3 \swarrow b \searrow t_4, \varepsilon >]\!\!\}) \uplus (P \cap (\mathbf{L}\odot) * Subt(t_3)) \uplus (P \cap (\mathbf{R}\odot) * Subt(t_4)))$
$\quad =\{$ definition P, $\cap$, π_2* distributes over $\uplus$ $\}$
$\pi_2 * (\mathbf{if}\ a = b\ \mathbf{then}\ \{\!\![< t_3 \swarrow b \searrow t_4, \varepsilon >]\!\!\}\ \mathbf{else}\ \emptyset\ \mathbf{fi}) \uplus$

$$\pi_2 * (P{\cap\!\!\!\cap}(\mathbf{L}\odot) * Subt(t_3)) \uplus \pi_2 * (P{\cap\!\!\!\cap}(\mathbf{R}\odot) * Subt(t_4))$$
$$=\{\ \pi_2* \text{ distributes over } \mathbf{if},\ P{\cap\!\!\!\cap} \text{ independent of path}\}$$
$$(\mathbf{if}\ a = b\ \mathbf{then}\ \{\!|\varepsilon|\!\}\ \mathbf{else}\ \emptyset\ \mathbf{fi}) \uplus$$
$$\pi_2 * ((\mathbf{L}\odot) * P{\cap\!\!\!\cap} Subt(t_3)) \uplus \pi_2 * ((\mathbf{R}\odot) * P{\cap\!\!\!\cap} Subt(t_4))$$
$$=\{\ \pi_2 \cdot (x\odot) = ([x]\!\!+\!\!) \cdot \pi_2\}$$
$$(\mathbf{if}\ a = b\ \mathbf{then}\ \{\!|\varepsilon|\!\}\ \mathbf{else}\ \emptyset\ \mathbf{fi}) \uplus$$
$$([\mathbf{L}]\!\!+\!\!) * \pi_2 * (P{\cap\!\!\!\cap} Subt(t_3)) \uplus ([\mathbf{R}]\!\!+\!\!) * \pi_2 * (P{\cap\!\!\!\cap} Subt(t_4))$$
$$=\{\ \text{definition } Paths\ \}$$
$$(\mathbf{if}\ a = b\ \mathbf{then}\ \{\!|\varepsilon|\!\}\ \mathbf{else}\ \emptyset\ \mathbf{fi}) \uplus ([\mathbf{L}]\!\!+\!\!) * Paths(a, t_3) \uplus ([\mathbf{R}]\!\!+\!\!) * Paths(a, t_4).$$

5.8 *ISBES* as the starting point of transformational developments

We have presented a characterization of a class of problems, and a number of examples. Readers familiar with the examples may notice that our specifications of these example problems do not always take the most familiar or simple form. Thus, we need to motivate why it may be useful to specify these problems in such a way.

In our methodology, viz. *transformational programming*, formal specifications mainly serve as starting points of transformational developments. The usefulness of a specification is strongly related to the ease of deriving efficient algorithms from it. Thus, we have to investigate the applicability of relevant program transformation techniques (cf. [Fea87, Par90, SL90] for overviews) to problems specified in this way. From the number of algorithms known for some of our examples it is clear that many such techniques exist.

Although this is clearly a subject for further research, some ideas can already be presented. Following [SL90], we can point out the following:

- In their most general form, our specifications are of a *generate* (decompose) *and test* (intersect) nature. Thus, well-known strategies like *filter promotion* are obviously applicable.

- Because the intersection operator $\cap\!\!\!\cap$ is homomorphic (in the sense of [Bir87]), *divide-and-conquer* strategies may be readily applied, particularly when the decomposition and result functions are homomorphic as well.

- In many cases, the bag of substructures of a certain object can be ordered. Thus, *global search* strategies (binary search, backtracking, etc.) may be applied.

A number of laws that can be used for problems specified in ISBES-form is contained in appendix 1.

Appendix 1: A number of laws for calculations

In this appendix, a number of laws are presented that may prove useful in calculations (program transformations) on bags and ISBES problems. Often, laws only hold for certain types. Types are not indicated explicitly; variables B and C denote bags (of type $Bag(\alpha)$), variables S and T denote sets (of type $Set(\alpha)$). p and q denote predicates on α.

First, a number of properties of the $\Cap$ operator are mentioned (recall that $\cap$ denotes bag intersection).

$$S \Cap \emptyset = \emptyset \tag{5.5}$$
$$S \Cap (B \uplus C) = (S \Cap B) \uplus (S \Cap C) \tag{5.6}$$
$$\emptyset \Cap B = \emptyset \tag{5.7}$$
$$(S \cap T) \Cap B = (S \Cap B) \cap (T \Cap B) \tag{5.8}$$
$$S \Cap (B \cap C) = (S \Cap B) \cap (S \Cap C) \tag{5.9}$$

The first two laws are actually just the definition of $\Cap$.

$\Cap$ is defined in terms of $\lhd$ and thus a number of properties of $\lhd$ are also useful (e.g. for proving the laws above). $\lhd$ on bags enjoys a number of properties, some of which do not hold on lists.

$$(p \wedge q) \lhd B = p \lhd (q \lhd B) \tag{5.10}$$
$$(p \wedge q) \lhd B = (p \lhd B) \cap (q \lhd B) \tag{5.11}$$
$$(p \vee q) \lhd B = ((p \lhd B) \uplus (q \lhd B)) - (p \wedge q) \lhd B \tag{5.12}$$
$$\neg q \lhd B = B - q \lhd B \tag{5.13}$$
$$(p \Rightarrow q) \Rightarrow p \lhd B \subseteq q \lhd B \tag{5.14}$$

For suitable definitions of $-$ and $\subseteq$ on lists, laws 5.13 and 5.14 also hold for lists. 5.12 and 5.11 have no counterparts for lists.

Similarly, also $\mathsf{X}_\oplus$ enjoys a number of special properties. $\widetilde{\oplus}$ is defined by: $a \widetilde{\oplus} b = b \oplus a$.

$$B \mathsf{X}_\oplus C = C \mathsf{X}_{\widetilde{\oplus}} B \tag{5.15}$$
$$B \mathsf{X}_\oplus (C \uplus D) = (B \mathsf{X}_\oplus C) \uplus (B \mathsf{X}_\oplus D) \tag{5.16}$$
$$(B \uplus C) \mathsf{X}_\oplus D = (B \mathsf{X}_\oplus D) \uplus (C \mathsf{X}_\oplus D) \tag{5.17}$$

Law 5.16 is just the definition of $\mathsf{X}_\oplus$. Law 5.15 only holds on bags (and sets), 5.17 can be proved using 5.15 and 5.16.

Several of the bags that occur in this paper "are actually just sets". This information can profitably be used in derivations. We define type conversion functions $\textsc{Bag} : Set(\alpha) \to Bag(\alpha)$ and $\textsc{Set} : Bag(\alpha) \to Set(\alpha)$, and a predicate $\textsc{IsSet}$, that checks whether a bag "is actually a set", by:

$$\textsc{Set}(B) = (\cup/ \cdot \{\cdot\}*)B$$
$$\textsc{Bag}(\emptyset) = \emptyset$$

$$\begin{aligned}
\textsc{Bag}(\{x\}) &= \{\!|\, x \,|\!\} \\
\textsc{Bag}(S \cup T) &= (\textsc{Bag}(S) \uplus \textsc{Bag}(T)) - \textsc{Bag}(S \cap T) \\
\textsc{IsSet}(B) &= (\textsc{Bag}(\textsc{Set}(B)) = B)
\end{aligned}$$

$\textsc{Bag}(S)$ returns a bag containing each element of S once. In the laws below, C is a bag that satisfies IsSet, and f is an injective function.

$$\textsc{Set}(\textsc{Bag}(S)) = S \tag{5.18}$$
$$p \lhd C = \textsc{Bag}(p \lhd \textsc{Set}(C)) \tag{5.19}$$
$$\oplus/C = \oplus/\textsc{Set}(C) \tag{5.20}$$
$$f * C = \textsc{Bag}(f * \textsc{Set}(C)) \tag{5.21}$$
$$\textsc{Set}(p \lhd C) = p \lhd \textsc{Set}(C) \tag{5.22}$$
$$\textsc{Set}(f * C) = f * \textsc{Set}(C) \tag{5.23}$$
$$\textsc{IsSet}(\textsc{Bag}(\textsc{Set}(B))) = \textbf{true} \tag{5.24}$$

5.22 and 5.23 can be proved directly from 5.19 and 5.21, respectively, using 5.18[3]. Obviously, $\Cap$ reduces to $\cap$ for "bags that are actually sets" (here, C satisfies IsSet):

$$S \Cap C = \textsc{Bag}(S \cap \textsc{Set}(C)) \tag{5.25}$$
$$\textsc{Set}(S \Cap C) = S \cap \textsc{Set}(C) \tag{5.26}$$
$$S \Cap \textsc{Bag}(\textsc{Set}(B)) = \textsc{Bag}(S \cap \textsc{Set}(B)) \tag{5.27}$$

5.26 is a trivial consequence of 5.25, using 5.18. 5.27 can be deduced from 5.25 using 5.18 and 5.24.

[3] Actually, they hold for arbitrary bags, but this requires a different proof.

Chapter 6

Solving a combinatorial problem by transformation of abstract data types

6.1 Introduction

In 1965, C.H.A. Koster [Kos65] described an operator for creating permutations of strings. It is denoted by ⊓⊔[1], and has the well-known interpretation from proof reading: ⊓a b⊔ denotes the string *ba*, and ⊓e pl⊓an ex⊔⊔ denotes the string *example*. Koster used the ⊓⊔ operator for the description of transducers using affix grammars. Recently [Kos90], he posed the question whether the set of permutations that can be generated using only ⊓⊔ in a nested fashion (Koster calls these *inversions*; we follow van Leijenhorst [vL90] in calling them *K-permutations*) differs significantly from the set of (ordinary) permutations of a string. For example, $[2,4,1,3]$ and $[3,1,4,2]$ are (the only) two permutations of $[1,2,3,4]$ that are not K-permutations of $[1,2,3,4]$.

An answer to this question appears in [vL90]: it is shown there how the number of K-permutations of a string consisting of n distinct elements is bounded by 9.9179^n and therefore much less than $n!$, using formal power series and estimates of integrals. A more recent analysis [vL91] of these results leads to an upper bound of $\approx 7.1^n$. We will present a very simple proof that 8^n is an upper bound, and a reduction to normal form of K-permutations by way of abstract data type transformations. From this, an exact formula counting the number of K-permutations is derived.

6.2 Preliminaries

The usual BMF (Bird-Meertens Formalism) notation for sets and lists is used [Bir87, Mee89a]. The reader is referred to [Bir87] for formal definitions of the BMF operators used in this paper. Informally, they can be characterized by:

$$[a_1,\ldots,a_n]\#[b_1,\ldots,b_m] \;=\; [a_1,\ldots,a_n,b_1,\ldots,b_m]$$

[1] Or alternatively ⊔⊓

$$
\begin{array}{rcl}
[\,]\#x = x\#[\,] & = & x \\
f * [a_1, \ldots, a_n] & = & [f\ a_1, \ldots, f\ a_n] \\
\oplus/[a_1, \ldots, a_n] & = & a_1 \oplus \ldots \oplus a_n \\
(a\oplus)x & = & a \oplus x \\
(\oplus a)x & = & x \oplus a \\
a\tilde{\oplus}b & = & b \oplus a \\
[a_1, \ldots, a_n]\,\mathsf{X}_{\oplus}\,[b_1, \ldots, b_m] & = & [a_1 \oplus b_1, \ldots, a_n \oplus b_1, \ldots, a_1 \oplus b_m, \ldots, a_n \oplus b_m] \\
(f\hat{\oplus}g)x & = & (f\ x) \oplus (g\ x).
\end{array}
$$

Using this notation, the set of permutations of a list is given by:

$$
\begin{array}{rcl}
perms\ [\,] & = & \{[\,]\} \\
perms\ [a] & = & \{[a]\} \\
perms\ (l\#m) & = & \cup/((perms\ l)\,\mathsf{X}_{\odot}\ (perms\ m)) \\
 & & \text{where} \\
[\,] \odot m & = & \{m\} \\
m \odot [\,] & = & \{m\} \\
([a]\#l) \odot ([b]\#m) & = & ([a]\#) * (l \odot ([b]\#m)) \cup \\
 & & ([b]\#) * (([a]\#l) \odot m)
\end{array}
$$

(the operator $\odot$ takes two lists and merges them in all possible ways).

We also use the inverse operator:

$$f^{-1}\ x = \{y \mid f\ y = x\}.$$

6.3 K-permutation patterns

First we present an informal specification, more or less as given by Koster:

> *A K-permutation of a string can be constructed as follows:*
>
> - *choose two arbitrary adjacent nonempty substrings;*
> - *interchange these.*
> - *Repeat this as necessary within the chosen substrings, or in the unchanged part of the string.*

This can be straightforwardly translated into a formal specification of the type *Kperm*, which is an extension of the type *List* of *nonempty* lists. The element type is denoted by α. We will write $a \sqcap\!\sqcup b$ for $\lceil a\ b \rfloor$, occasionally using brackets for disambiguation.

$$\frac{x : \alpha}{[x] : Kperm(\alpha)} \qquad \frac{a, b : Kperm(\alpha)}{a\#b : Kperm(\alpha)} \qquad \frac{a, b : Kperm(\alpha)}{a \sqcap\!\sqcup b : Kperm(\alpha)}$$

Because we consider *Kperm* as a extension of *List*, the $+\!\!+$ operator of *List* is used here as well, and its associativity is also assumed. So one law certainly holds for *Kperm*:

$$(a+\!\!+b)+\!\!+c = a+\!\!+(b+\!\!+c). \tag{6.1}$$

This allows to leave out brackets in expressions with multiple occurrences of $+\!\!+$.

The intended interpretation of $a \bigsqcap\!\!\bigsqcup b$ is, of course, $b+\!\!+a$. It is possible to add this as an equivalence on the type *Kperm*:

$$a \bigsqcap\!\!\bigsqcup b = b+\!\!+a. \tag{6.2}$$

However, doing so would result in the loss of important structure from the type *Kperm*, viz. what string a *Kperm* term "is a permutation of". This information is retained by including *no* laws that allow changing the order of the basic elements in *Kperm* terms.

Formally, this can be described as follows.

Definition 6.3 Given two[2] functions $F : \alpha \to \beta$ and $G : \alpha \to \gamma$, the (F, G)-induced equivalence $=_{F,G}$ (on α) is defined by

$$x =_{F,G} y \Leftrightarrow ((F\ x) = (F\ y) \wedge (G\ x) = (G\ y)).$$

We add laws L to *Kperm* such that $x =_{I,O} y \Leftrightarrow x =_L y$, where I and O are functions that give for a *Kperm* term the "original" list and the "interpretation", i.e. the permutation that is represented. The "original" list is obtained by replacing all occurrences of $\bigsqcap\!\!\bigsqcup$ by $+\!\!+$, and the string that is actually represented by the K-permutation is obtained by replacing all occurrences of $\bigsqcap\!\!\bigsqcup$ by $\widetilde{+\!\!+}$. The homomorphisms O (for *O*riginal) and I (for *I*nterpretation) are given by:

$$\begin{aligned}
O[a] &= [a] \\
O(l+\!\!+m) &= l+\!\!+m \\
O(l \bigsqcap\!\!\bigsqcup m) &= l+\!\!+m \\
I[a] &= [a] \\
I(l+\!\!+m) &= l+\!\!+m \\
I(l \bigsqcap\!\!\bigsqcup m) &= m+\!\!+l.
\end{aligned}$$

As mentioned before, associativity of $+\!\!+$ is assumed. Another equivalence that must be added to have (I, O)-induced equivalence on *Kperm* is associativity of $\bigsqcap\!\!\bigsqcup$:

$$(a \bigsqcap\!\!\bigsqcup b) \bigsqcap\!\!\bigsqcup c = a \bigsqcap\!\!\bigsqcup (b \bigsqcap\!\!\bigsqcup c) \tag{6.4}$$

(since both have the same elements, read from left to right, and both denote the string $c+\!\!+b+\!\!+a$). This is the point where van Leijenhorst's analysis [vL90] is non-optimal: in the grammar he uses, associativity of $+\!\!+$ is (implicitly) used, whereas associativity of $\bigsqcap\!\!\bigsqcup$ is not.

Note that the definitions of O and I are sound for *Kperm* with the two associativity laws 6.1 and 6.4, since $+\!\!+$ is associative; if equation 6.2 were added as an equivalence, soundness of O would imply that $l+\!\!+m = m+\!\!+l$ for all l and m (since $O(l \bigsqcap\!\!\bigsqcup m) = l+\!\!+m$, $O(m+\!\!+l) = m+\!\!+l$). This has the undesired effect of reducing the result type of O from lists to bags.

[2]The generalization to different numbers of functions is obvious.

6.4 All K-permutations of a list

Now that functions O and I have been defined, the problem posed by Koster can be formally specified. The set of all K-permutations of a list l is

$$\{m \mid \exists n : (O\ n) = l \wedge (I\ n) = m\},$$

or, more concisely

$$I * (O^{-1} l).$$

Then the problem of determining whether there exist a sizable number of permutations that cannot be generated using $\sqcap\!\sqcup$ can be specified as follows:
Investigate $f[1..n]$ where

$$f = (\# \cdot perms)\hat{-}(\# \cdot I * \cdot O^{-1}).$$

One way to continue the analysis would be by finding a more efficient program for f. In this case, however, a further analysis on the *data structure* side will prove to be more useful.

6.5 Determining the number of K-permutation patterns

Using the formal specification given above, we can now give an upper bound for the number of K-permutations by considering K-permutation patterns.

Consider a string l of length n. Between each two successive elements of that string one can imagine a concatenation operator (so, $n-1$ in total). The O operation above consists of two steps: first, all $\sqcap\!\sqcup$ operators are replaced by $+\!\!+$. The second step is more implicit: because $+\!\!+$ is associative, all bracketing in the result is irrelevant and can be eliminated. So, arbitrary elements from $O^{-1}l$ can be constructed by reverting this process: first, an arbitrary complete bracketing of l is chosen, and then some $+\!\!+$ operators are replaced by $\sqcap\!\sqcup$.

Now it is easy to count the number of K-permutation *patterns* (i.e., the ways of placing brackets and $+\!\!+$ or $\sqcap\!\sqcup$ operators, without considering equivalences among those):

- As mentioned by van Leijenhorst [vL90], the number of complete binary bracketings of a string of length n is given by the Catalan number [Cat38]

$$C_n = \frac{1}{n}\left(\begin{array}{c} 2n-2 \\ n-1 \end{array}\right).$$

- Of $(n-1)$ $+\!\!+$-operators, an arbitrary number is replaced by $\sqcap\!\sqcup$; this can be done in 2^{n-1} ways.

- Thus, the number of K-permutation patterns of a string of length n is given by

$$K_n = 2^{n-1} C_n = \frac{2^{n-1}}{n}\left(\begin{array}{c} 2n-2 \\ n-1 \end{array}\right).$$

Since the Catalan numbers are known to be bounded by $C_n < 4^n$, this gives an upper bound of 8^n on the number of K-permutation patterns, and thus also on the number of K-permutations.

6.6 Lists and rose trees with append and reverse

An upper bound for the number of K-permutations has been given, by considering an abstract data type, disregarding equivalences on that type. In order to investigate more precisely the exact number of K-permutations, the data type involved must be as simple as possible. The description using *Kperm* is highly symmetric: there are two "concatenation" operators, both associative. This can be corrected by a translation from *Kperm* to a data type with one binary operator (concatenation) and one unary operator (reverse). This is the type *Revlist*, with introduction rules:

$$\frac{x : \alpha}{[x] : \mathit{Revlist}(\alpha)} \qquad \frac{a, b : \mathit{Revlist}(\alpha)}{a \# b : \mathit{Revlist}(\alpha)} \qquad \frac{a : \mathit{Revlist}(\alpha)}{\overline{a} : \mathit{Revlist}(\alpha)}$$

The transition from *Kperm* to *Revlist* is given by the translation function T, defined by:

$$\begin{aligned} T[a] &= [a] \\ T(l \# m) &= (T\ l) \# (T\ m) \\ T(l \sqcap\!\sqcup m) &= \overline{\overline{(T\ l)} \# \overline{(T\ m)}}. \end{aligned}$$

For *Revlist*, the "interpretation" and "original" homomorphisms I' and O' are given by:

$$\begin{aligned} I'\ [a] &= [a] \\ I'(l \# m) &= (I'\ l) \# (I'\ m) \\ I'\ \overline{m} &= \mathit{rev}(I'\ m) \\ O'[a] &= [a] \\ O'(l \# m) &= (O'\ l) \# (O'\ m) \\ O'\ \overline{m} &= O'\ m \end{aligned}$$

where *rev* is a suitably defined reverse-function on *List*.

The following laws are assumed:

$$\begin{aligned} \overline{\overline{m}} &= m \\ \overline{[a]} &= [a] \\ (l \# m) \# n &= l \# (m \# n), \end{aligned}$$

where the first two laws can be used as directed equivalences, for normalization of *Revlist* terms. I.e., except for the associativity of append, we can assume a unique normal form for *Revlist*. Note that the equivalence induced by these laws is exactly (I', O')-induced equivalence.

Theorem 6.5
If $x = y$ according to the laws of *Kperm*, then $(T\ x) = (T\ y)$ according to the laws of *Revlist*.

Proof 6.5
Trivially, using associativity of $\#$, one can prove that $T((l\#m)\#n) = T(l\#(m\#n))$. We tacitly use associativity in the other half of the proof:

$$\begin{array}{rll}
T(l \sqcap\!\sqcup (m \sqcap\!\sqcup n)) & =_{\{\text{definition } T\}} & \overline{\overline{(T\ l)}\#\overline{T(m \sqcap\!\sqcup n)}} \\
& =_{\{\text{definition } T\}} & \overline{\overline{(T\ l)}\#\overline{\overline{(T\ m)}\#\overline{(T\ n)}}} \\
& =_{\{\overline{\overline{m}} = m\}} & \overline{\overline{(T\ l)}\#\overline{(T\ m)}\#\overline{(T\ n)}} \\
& =_{\{\overline{\overline{m}} = m\}} & \overline{\overline{(T\ l)}\#\overline{(T\ m)}\#\overline{(T\ n)}} \\
& =_{\{\text{definition } T\}} & \overline{\overline{T(l \sqcap\!\sqcup m)}\#\overline{(T\ n)}} \\
& =_{\{\text{definition } T\}} & T((l \sqcap\!\sqcup m) \sqcap\!\sqcup n).\Box
\end{array}$$

Theorem 6.6
The interpretation and original homomorphisms I' and O' on *Revlist* are equivalent to I and O on *Kperm*, respectively:

1. $I = I' \cdot T$
2. $O = O' \cdot T$

Proof 6.6

1. By structural induction on *Kperm*, using the property of *rev*:

$$rev((rev\ x)\#(rev\ y)) = y\#x.$$

$$\begin{array}{rll}
I'(T[a]) & =_{\{\text{def. } T\}} & I'[a] \\
& =_{\{\text{def. } I'\}} & [a] \\
& =_{\{\text{def. } I\}} & I[a] \\
I'(T(l\#m)) & =_{\{\text{def. } T\}} & I'(T\ l)\#I'(T\ m) \\
& =_{\{\text{induction}\}} & (I\ l)\#(I\ m) \\
& =_{\{\text{def. } I\}} & I(l\#m) \\
I'(T(l \sqcap\!\sqcup m)) & =_{\{\text{def. } T\}} & I'\overline{\overline{(T\ l)}\#\overline{(T\ m)}} \\
& =_{\{\text{def. } I'\}} & rev(I'(\overline{(T\ l)}\#\overline{(T\ m)})) \\
& =_{\{\text{def. } I'\}} & rev(rev(I'(T\ l))\#rev(I'(T\ m))) \\
& =_{\{\text{property } rev\}} & (I'(T\ m))\#(I'(T\ l)) \\
& =_{\{\text{induction}\}} & (I\ m)\#(I\ l) \\
& =_{\{\text{def. } I\}} & I(l \sqcap\!\sqcup m)\Box
\end{array}$$

2. Analogously.

Theorem 6.7
T is total, injective and surjective, and thus an isomorphism between *Kperm* and *Revlist* (with laws).

Proof 6.7

- Obviously, T is total, since it is defined inductively over all the constructors of *Kperm*.
- Injectivity of T follows from the fact that the equivalences induced by the introduced laws are the (I, O)-induced and the (I', O')-induced equivalence, respectively.
- Surjectivity of T can be proved by induction on the length (i.e., the number of basic elements) of the *Revlist*, considering normalized terms only.

  Hypothesis $\forall Revlist\ x : length(x) \leq n \Rightarrow (\exists Kperm\ y : (T\ y) = x)$. This will be verbalized as "for every x a T-original exists".

  Base case For $n = 1$, obviously $[a] = T[a]$.

  Induction Any *Revlist* with *length* > 1 is of one of two forms: $l \# m$ or $\overline{p \# q}$.

  l and m have T-originals l' and m', by induction hypothesis, and thus $l \# m$ has a T-original, viz. $l' \# m'$.

  We may assume that $p = \overline{p'}$ and $q = \overline{q'}$ because of the law $\overline{\overline{m}} = m$. By induction, p' and q' have T-originals p'' and q'', and thus $\overline{p \# q}$ has a T-original, viz. $p'' \sqcap\!\sqcup q''$. □

In order to arrive at a unique representation of K-permutations, we use yet another data type, which may be called a rose tree [Mee89b]. In this type the associativity of $\#$ is factored out. We assume the existence of a type *Plist* of lists with ≥ 2 elements. The types RT and $\overline{RT}$ are defined by:

$$\frac{a : \alpha}{\begin{array}{c}[a] : RT(\alpha) \\ [a] : \overline{RT}(\alpha)\end{array}} \qquad \frac{l : Plist(RT(\alpha))}{\textcircled{\#} l : \overline{RT}(\alpha)} \qquad \frac{l : Plist(\overline{RT}(\alpha))}{\textcircled{\#} l : RT(\alpha)}$$

The notations $\textcircled{\#}$ and $\textcircled{\#}$ have no formal meaning in this context. They do, however, serve the intuition. A node of type $\textcircled{\#}$ represents all its sons from left to right, a node of type $\textcircled{\#}$ represents all its sons from right to left.

The transition from *Revlist* to RT can best be specified by its inverse T^I:

$$\begin{array}{rcl} T^I[a] & = & [a] \\ T^I(\textcircled{\#} l) & = & \#/T^I * l \\ T^I(\textcircled{\#} l) & = & \overline{\#/(\overline{} \cdot T^I) * L} \end{array}$$

which is correct since T^I is surjective and injective.

Returning to combinatorics, we can now see that the problem of counting all K-permutations is closely related to Schröder's generalized bracketing problem [Sch70] as presented in [Com74]. There, the problem is to determine the number c_n of different rose trees with ≥ 2 branches at each inner node and n leaves. A recursive formula is given[3] for c_n:

$$\begin{aligned} (n+1)c_{n+1} &= 3(2n-1)c_n - (n-2)c_{n-1} \quad \text{for } n \geq 2 \\ , c_1 = c_2 &= 1. \end{aligned}$$

Since the types (viz.  or) of the subtrees at all levels of a $RT/\overline{RT}$ tree are completely determined by the type of the root, for each rose tree with a given number of leaves there are exactly two $RT/\overline{RT}$ trees with that number of leaves, viz. one with a root and one with a root. Thus, we can conclude that the number of K-permutations of a string of length n equals $2c_n$, for $n \geq 2$. This improves the results as given in [vL90, vL91].

6.7 Concluding remarks

It has been shown how techniques from the area of formal specification may be profitably used in the analysis of combinatorial problems. The formal "game" we played may be relevant for other problems as well. In short, it may be described as follows:

- given an abstract data type A without laws, and two functions I_A and O_A, add laws L_A that construct the equivalence classes w.r.t. I_A and O_A, i.e. for terms x and y, $x =_{L_A} y$ should hold iff $I_A(x) = I_A(y) \wedge O_A(x) = O_A(y)$;

- while the type A has laws, do the following:
  define a new data type A', and a function T from the (terms of the) previous data type A to A', such that $T(x) = T(y) \Rightarrow x =_{L_A} y$. Define functions $I_{A'}$ and $O_{A'}$, such that $I_A(x) = I_{A'}(T(x))$ and $O_A(x) = O_{A'}(T(x))$. Then add laws for A' that construct equivalence classes w.r.t. $I_{A'}$ and $O_{A'}$.

  A suitable choice for such a function T is one such that maps two or more laws from L_A to one and the same law in the new type (like associativity of $+\!\!+$ and of $\sqcap\!\!\sqcup$ in *Kperm* were both mapped to associativity of $+\!\!+$ in *Revlist*). This may happen if T is not injective on *terms of* A.

- if a data type without laws is obtained, one has a unique normal form for the original data type A (w.r.t. I_A and O_A).

On a more abstract level, this means that for an abstract data type A and functions I and O, we construct the quotient type of A w.r.t. (I, O)-induced equality, by adding laws to that effect, and eventually construct a "free" data type for the quotient.

[3] The formula in [Com74] actually is incorrect: it has $(n-1)c_{n+1}$ as its left hand side. The formula above does conform with the table of c_n values presented in [Com74]. This error has been discovered by Hans Zantema, who refereed this chapter for CSN '91.

If this process could be carried out in reverse, this would have interesting applications in the area of implementation of abstract data types, since then abstract data types with laws could be implemented by free types with an explicit equality function.

If one were to consider a data type with $\sqcap\!\sqcup$, $+\!\!+$ and $\overline{}$, one would find that some nice equivalences hold on this data type, resembling the well-known De Morgan-laws in Boolean algebras. D. Turner describes this in [Tur90].

A remaining interesting problem, for which no better than a trivial exponential algorithm has been given, is the *correspondence* problem for K-permutations, i.e. given strings x and y, does a K-permutation z exist, such that $(O\ z) = x \wedge (I\ z) = y$? This may be the subject of further studies.

Bibliography

[AHU75] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Programs*. Addison-Wesley, Reading, Mass., 1975.

[AK82] J. Arsac and Y. Kodratoff. Some techniques for recursion removal from recursive functions. *ACM Transactions on Programming Languages and Systems*, 4(2):295–322, April 1982.

[BB84] M. Broy and F.L. Bauer. A systematic approach to language constructs for concurrent programs. *Science of Computer Programming*, 4:103–139, 1984.

[BBB+85] F.L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, and H. Wössner. *The Munich Project CIP. Volume I: The Wide Spectrum Language CIP-L*, volume 183 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin/Heidelberg/New York, 1985.

[BD77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[BdBM+91] R.C. Backhouse, P.J. de Bruin, G. Malcolm, E. Voermans, and J. van der Woude. Relational catamorphisms. In Möller [Möl91], pages 287–318.

[BEH+87] F.L. Bauer, H. Ehler, A. Horsch, B. Möller, H. Partsch, O. Paukner, and P. Pepper. *The Munich Project CIP. Volume II: The Transformation System CIP-S*, volume 292 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1987.

[BG85] V.J. Bush and J.R. Gurd. Transforming recursive programs for execution on parallel machines. In J.P. Jouannaud, editor, *Functional Programming and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 350–367, Berlin, 1985. Springer-Verlag.

[BGJ89] R.S. Bird, J. Gibbons, and G. Jones. Formal derivation of a pattern matching algorithm. *Science of Computer Programming*, 12:93–104, 1989.

[Bir80] R.S. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys*, 12(4):403–417, December 1980.

[Bir84] R.S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, 1984.

[Bir87] R.S. Bird. An introduction to the theory of lists. In Broy [Bro87], pages 5–42.

[BM77] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[Boi89] E.A. Boiten. Inverting the flow of computation. Technical Report 89-10, Dept. of Informatics, K.U. Nijmegen, 1989.

[Boi91a] E.A. Boiten. Can bag comprehension be used at all? Technical Report 91-21, Dept. of Informatics, K.U. Nijmegen, September 1991.

[Boi91b] E.A. Boiten. The many disguises of accumulation. Technical Report 91-26, Dept. of Informatics, K.U. Nijmegen, December 1991.

[Boi92] E.A. Boiten. Remember and recall: language constructs for abstract description of tabulation. In preparation, 1992.

[Bor85] P.B. Borwein. On the complexity of calculating factorials. *Journal of Algorithms*, 6:376–380, 1985.

[Bro87] M. Broy, editor. *Logic of Programming and Calculi of Discrete Design. NATO ASI Series Vol. F36*, Berlin, 1987. Springer-Verlag.

[Bur68] R.M. Burstall. Semantics of assignment. In E. Dale and D. Michie, editors, *Machine Intelligence, Vol. 2*, pages 3–20, Edinburgh, 1968. Oliver and Boyd.

[BvdBvD+90] E.A. Boiten, M.G.J. van den Brand, N.W.P. van Diepen, C.H.A. Koster, H.A. Partsch, and N. Völker. USTOPIA requirements – Thoughts on a User-friendly System for Transformation Of Programs In Abstracto. Technical Report 90-12, University of Nijmegen, Department of Informatics, 1990.

[BW82] F.L. Bauer and H. Wössner. *Algorithmic Language and Program Development*. Springer-Verlag, Berlin, 1982.

[Cat38] Catalan. Note sur une équation aux différences finies. *J.M. pures appl.*, 3:508–516, 1838.

[Coh79] N.H. Cohen. Characterization and elimination of redundancy in recursive programs. In *ACM Principles of Programming Languages*, pages 143–157, 1979.

[Coh83] N.H. Cohen. Eliminating redundant recursive calls. *ACM Transactions on Programming Languages and Systems*, 5(3):265–299, July 1983.

[Com74] L. Comtet. *Advanced Combinatorics - The art of finite and infinite expansions*. Reidel, Dordrecht, 1974.

[Coo66] D.C. Cooper. The equivalence of certain computations. *Computer Journal*, 9:45–52, 1966.

[CR89] W. Clinger and J. Rees, editors. *Revised Report on the Algorithmic Language Scheme*, August 1989.

[DFH+91] J. Darlington, A.J. Field, P.G. Harrison, D. Harper, G.K. Jouret, P.J. Kelly, K.M. Sephton, and D.W. Sharp. Structured parallel functional programming. In H. Glaser and P. Hartel, editors, *Proceedings of the Third International Workshop on the Implementation of Functional Languages on Parallel Architectures*, pages 31–51, 1991.

[Dij76a] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1976.

[Dij76b] E.W. Dijkstra. *An Exercise for Dr. R.M. Burstall, EWD 570*, pages 215–216. Springer-Verlag, Berlin, May 1976. In [Dij82].

[Dij76c] E.W. Dijkstra. *More About the Function "fusc" (A Sequel to EWD570), EWD578*, pages 230–232. Springer-Verlag, Berlin, August 1976. Published in [Dij82].

[Dij82] E.W. Dijkstra. *Selected Writings - A personal Perspective on Computer Science*. Springer-Verlag, Berlin, 1982.

[dR47] G. de Rham. Un peu de mathématiques à propos d'une courbe plane. *Elemente der Mathematik*, 2:95, 1947.

[Ear76] J. Earley. High-level iterators and a method for automatically designing data structure representation. *Computer Languages*, 1(4):321–342, 1976.

[Fea87] M.S. Feather. A survey and classification of some program transformation approaches and techniques. In Meertens [Mee87], pages 165–196.

[Flo67] R.W. Floyd. Assigning meaning to programs. In J.T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume XIX of *Proc. Symposia in Applied Mathematics*, pages 19–32. AMS, 1967.

[FvGGM90] W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J. Misra, editors. *Beauty is our business*. Springer Verlag, Berlin/Heidelberg/New York, 1990.

[Har88] P.G. Harrison. Linearisation: an optimisation for non-linear functional programs. *Science of Computer Programming*, 10(3):281–319, June 1988.

[HL78] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[JM80] N.D. Jones and C.M. Madsen. Attribute influenced LR parsing. In N.D. Jones, editor, *Proceedings of the Aarhus Workshop on Semantics-Directed Compiler Generation*, pages 393–407, Berlin, 1980. Springer-Verlag.

[Kho90] H. Khoshnevisian. Efficient memo-table management strategies. *Acta Informatica*, 28:43–81, 1990.

[KMP77] D.E. Knuth, J.H. Morris, Jr, and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1977.

[Kos65] C.H.A. Koster. On the construction of ALGOL-procedures for generating, analysing and translating sentences in natural languages. Technical Report MR 72, Mathematisch Centrum, Amsterdam, February 1965.

[Kos90] C.H.A. Koster, November 1990. Personal communication.

[Mal90] G.R. Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, Rijksuniversiteit Groningen, September 1990.

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[Mee86] L.G.L.T. Meertens. Algorithmics — towards programming as a mathematical activity. In J.W. de Bakker, M. Hazewinkel, and J.K. Lenstra, editors, *Proc. CWI Symposium on Mathematics and Computer Science*, volume 1 of *CWI Monographs*, pages 289–334, 1986.

[Mee87] L.G.L.T. Meertens, editor. *Program Specification and Transformation. Proceedings of the IFIP TC2/WG2.1 Working Conference on Program Specification and Transformation*, Amsterdam, 1987. North-Holland Publishing Company.

[Mee89a] L.G.L.T. Meertens. Lecture notes on the generic theory of binary structures. In *STOP International Summer School on Constructive Algorithmics, Ameland* [STO89]. Lecture notes.

[Mee89b] L.G.L.T. Meertens. Variations on trees. In *STOP International Summer School on Constructive Algorithmics, Ameland* [STO89]. Lecture notes.

[Mei86] H. Meijer. *Programmar: A Translator Generator*. PhD thesis, Katholieke Universiteit Nijmegen, 1986.

[MFP91] E. Meijer, M.M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Functional Programming and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[Mic67] D. Michie. Memo functions: A language feature with rote learning properties. DMIP Memo. MIP-R-29, Edinburgh, 1967.

[Möl91] B. Möller, editor. *Proceedings of the IFIP TC2 Working Conference on Constructing Programs from Specifications*, Amsterdam, 1991. North-Holland Publishing Company.

[Mor90] J.M. Morris. Programming by expression refinement: the KMP algorithm. In Feijen et al. [FvGGM90], chapter 37, pages 327–338.

[Nau60] P. Naur (ed.). Report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3:299–314, 1960.

[odAMT90] R. op den Akker, B. Melichar, and J. Tarhio. The hierarchy of LR-attributed grammars. In P. Deransart and M. Jourdan, editors, *Attribute Grammars and their Applications*, volume 461 of *Lecture Notes in Computer Science*, pages 13–28, Berlin, 1990. Springer-Verlag.

[Par90] H. Partsch. *Specification and Transformation of Programs - a Formal Approach to Software Development*. Springer-Verlag, Berlin, 1990.

[PB85] C.A. Purdon, Jr. and C.A. Brown. *The Analysis of Algorithms*. Holt, Rinehart and Winston, New York, 1985.

[PB91] H.A. Partsch and E.A. Boiten. A note on similarity of specifications and reusability of transformational developments. In Möller [Möl91], pages 71–89.

[Pep91] P. Pepper. Literate program derivation: A case study. In M. Broy and M. Wirsing, editors, *Methods of Programming*, volume 544 of *Lecture Notes in Computer Science*, Berlin, 1991. Springer-Verlag.

[Pet77] A. Pettorossi. Transformation of programs and use of the tupling strategy. In *Proc. Infor. 77, Bled, Yugoslavia*, pages 1–6, 1977.

[Pet84a] A. Pettorossi. *Methodologies for Transformations and Memoing in Applicative Languages*. PhD thesis, University of Edinburgh, October 1984.

[Pet84b] A. Pettorossi. A powerful strategy for deriving efficient programs by transformation. In *ACM Symposium on LISP and Functional Programming*, pages 273–281, 1984.

[PH70] M.S. Paterson and C.E. Hewitt. Comparative schematology. In *Record of the Project MAC Conf. on Conc.Syst. and Par.Comp., Woods Hole, Mass.*, pages 119–127, New York, 1970. ACM.

[PK82] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, July 1982.

[PP76] H. Partsch and P. Pepper. A family of rules for recursion removal related to the Towers of Hanoi problem. *Information Processing Letters*, 5(6):174–177, December 1976.

[PP86] P. Pepper and H. Partsch. Program transformations expressed by algebraic type manipulations. *Technology and Science of Informatics*, 5(3):197–212, 1986.

[PS90] H.A. Partsch and F.A. Stomp. A fast pattern matching algorithm derived by transformational and assertional reasoning. *Formal Aspects of Computing*, 2:109–122, 1990.

[PV91] H.A. Partsch and N. Völker. Another case study on reusability of transformational developments - pattern matching according to Knuth, Morris, and Pratt. In M. Broy and M. Wirsing, editors, *Methods of Programming*, volume 544 of *Lecture Notes in Computer Science*, pages 35–48, Berlin, 1991. Springer-Verlag.

[Sch70] Schröder. Vier combinatorische Probleme. *Z. für M. Phys.*, 15:361–376, 1870.

[Sin87] M. Sintzoff. Expressing program developments in a design calculus. In Broy [Bro87], pages 343–366.

[SKW85] D. R. Smith, G. B. Kotik, and S. J. Westfold. Research on knowledge-based software environments at Kestrel Institute. *IEEE Transactions on Software Engineering*, SE-11(11):1278–1295, November 1985.

[SL90] D.R. Smith and M.R. Lowry. Algorithm theories and design tactics. *Science of Computer Programming*, 14:305–321, 1990.

[Smi90] Douglas R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, September 1990.

[Spi90] Mike Spivey. A functional theory of exceptions. *Science of Computer Programming*, 14:25–42, 1990.

[STO89] STOP. *STOP International Summer School on Constructive Algorithmics, Ameland*, September 1989. Lecture notes.

[Tur90] D.A. Turner. Duality and De Morgan principles for lists. In Feijen et al. [FvGGM90], chapter 47, pages 390–398.

[vdW89] J. van der Woude. Playing with patterns, searching for strings. *Science of Computer Programming*, 12:177–190, 1989.

[vH76] F.W. von Henke. An algebraic approach to data types, program verification, and program synthesis. In *Proc. Math. Foundations of Comp. Science '76*, volume 45 of *Lecture Notes in Computer Science*, pages 330–336, Berlin, 1976. Springer-Verlag.

[vL90] D.C. van Leijenhorst. On a ternary bracketing problem from the theory of formal languages. Technical Report 90-24, KUN, December 1990.

[vL91] D.C. van Leijenhorst, January 1991. Addendum to [vL90].

[Völ92] N. Völker. Deriving string matching algorithms. Technical Report 92-02, Dept. of Informatics, University of Nijmegen, January 1992.

[Wan80] M. Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, January 1980.

[WPP+83] M. Wirsing, P. Pepper, H. Partsch, W. Dosch, and M. Broy. On hierarchies of abstract data types. *Acta Informatica*, 20:1–33, 1983.

Samenvatting

In dit proefschrift worden aspecten onderzocht van de *formele afleiding* van *correcte* en *efficiënte* computerprogramma's vanuit *formele specificaties*.

Een formele specificatie beschrijft een op te lossen probleem op een hoog niveau – idealiter door te beschrijven *wat* er moet gebeuren, zonder vast te leggen *hoe* dat moet. Zulke *descriptieve* specificaties hebben het voordeel dat alle mogelijke oplossingen er uit afgeleid kunnen worden, terwijl *operationele* specificaties vaak alleen een specifieke oplossing suggereren.

In deze context bestaan formele afleidingen uit *semantiek behoudende transformaties*, dat wil zeggen stappen die programma's omzetten in equivalente (efficiëntere, meer operationele) programma's, dan wel in zogenaamde *descendant* (meer gedefinieerde, specifiekere) programma's. De resulterende programma's zijn zodoende *correct* met betrekking tot de oorspronkelijke specificatie *vanwege de constructie*.

Dit proefschrift doet verslag van een aantal onderzoekingen in nieuwe terreinen op het gebied van de specificatie en transformatie van programma's.

Hoofdstuk 2, "*Improving recursive functions by inverting the order of evaluation*" geeft een uitgebreid overzicht van een bijzondere transformatie-*strategie* (een grotere conceptuele stap in een transformationele ontwikkeling die op een meer abstract niveau beschreven kan worden). Deze strategie voor recursieve functies houdt in dat equivalente functies afgeleid worden die dezelfde argumenten gebruiken, maar dan in omgekeerde volgorde. Deze strategie speelt een belangrijke rol bij optimalisatie van recursieve functies, in het bijzonder van functies met een boomvormige recursiestructuur. Vaak zijn functies zodanig gedefinieerd dat bepaalde recursieve aanroepen meer dan eens geëvalueerd moeten worden; door de evaluatievolgorde om te keren worden zulke meervoudige evaluaties voorkomen.

Hoofdstuk 3, "*Factorisation of the factorial*", geeft naast toepassingen van transformatieregels uit hoofdstuk 2 een indicatie van de "*state of the art*" op het gebied van transformatieregels voor vereenvoudiging van recursie. Eenvoudige heuristieken sturen de afleiding van een nieuw algoritme voor het berekenen van de faculteitsfunctie. Een variant van dit algoritme voor een *pipeline*-architectuur wordt op een soortgelijke manier afgeleid.

Hoofdstuk 4, "*A note on similarity of specifications and reusability of transformational developments*", behandelt de mogelijkheden voor *hergebruik* van *transformationele afleidingen*. Vaak is beweerd dat dit volledig automatisch zou kunnen gebeuren, maar de ervaringen met een aantal afleidingen in dit hoofdstuk geven de indruk dat deze veronderstelling nogal optimistisch is. Het blijkt dat afleidingen vooral hergebruikt kunnen worden wanneer de transformatiestappen zeer abstract worden beschreven (in natuurlijke taal), en wanneer zeer

algemene specificaties worden bekeken. Het centrale begrip hierin is "*similarity*". Verschillende definities van dit informele begrip blijken te leiden tot verschillende soorten hergebruik. Varianten van een afleiding van een lineair zoekalgoritme leveren verschillende interessante zoekalgoritmen op, uiteindelijk leidend tot afleidingen van twee ingewikkelde *string matching* algoritmen door hergebruik.

Hoofdstuk 5, "*Intersections of sets and bags of extended substructures – a class of problems*", geeft een generalisatie van de *specificatie* van patroonherkenning. Er wordt een klasse van problemen beschreven die gezien kunnen worden als gegeneraliseerde patroonherkenningsproblemen. De essentie van patroonherkenning wordt gezien in het bepalen van de doorsnede van een verzameling en een *bag* van uitgebreide deelstructuren van een gestructureerd object. De verzameling bevat de patronen, en de uitgebreide deelstructuren zijn de mogelijke voorkomens, voorzien van een karakterisatie van hun positie in het oorspronkelijke object. Hieruit volgen de eerste ideeën over een theorie van (uitgebreide) deelstructuren. Het blijkt dat de abstracte manier van beschrijven van deze klasse van problemen mogelijkheden geeft voor "calculatie" in de stijl van BMF. Verder biedt deze beschrijving van de essentiële structuur van zulke problemen verschillende aangrijpingspunten voor bijbehorende oplossingsstrategieën.

Hoofdstuk 6, "*Solving a combinatorial problem by transformation of abstract data types*", beschrijft een toepassing van technieken uit de formele programmaontwikkeling op een heel ander gebied, namelijk de combinatoriek. Een gegeven combinatorisch probleem wordt beschreven in termen van abstracte datatypen met equivalenties. Door transformatie van deze datatypen wordt het probleem gereduceerd tot een bekend probleem. Abstracte datatypen blijken, vanwege de mogelijkheid om willekeurige equivalenties te introduceren, in dit geval een krachtig specificatieformalisme te zijn.

Summary

This thesis investigates aspects of the *formal derivation* of *correct* and *efficient* computer *programs* from *formal specifications*.

A formal specification describes a problem to be solved on a high level – ideally, it specifies *what* has to be done, but not *how*. Such *descriptive* specifications facilitate the derivation of any of the possible solutions, whereas operational specifications suggest only particular ones.

Formal derivations in this framework consist of *semantics preserving transformations*, i.e. steps that proceed from solutions to the initial specification to other, more defined, more operational, or more efficient ones. Thus, the resulting programs are *correct by construction* with respect to their initial specifications.

This thesis contains a number of case studies aiming at the exploration of new territories in the area of program specification and transformation.

Chapter 2, "*Improving recursive functions by inverting the order of evaluation*" gives a comprehensive survey of one particular transformation *strategy* (a larger conceptual step in a transformational development that can be described at a more abstract level). This strategy for recursive functions entails the derivation of equivalent functions that use in their recursive evaluations the same arguments in an inverted order. This is an important optimization strategy, in particular for tree-like recursive functions, that are often defined in such a way that several function calls need to be evaluated more than once. By evaluating the function in an inverted order, such multiple evaluations are eliminated.

Chapter 3, "*Factorization of the factorial*", illustrates a number of the transformations in chapter 2, and also demonstrates the state of the art in recursion simplification transformations. Directed by a small set of simple and well-known heuristics, a previously unknown algorithm for computing factorials is derived. Also, a similar development is shown leading to a corresponding program for a simple pipeline architecture.

Chapter 4, "*A note on similarity of specifications and reusability of transformational developments*", explores the possibilities of *reuse* of *transformational developments*. Although it has often been claimed that this could be done fully mechanically, the experience with a number of derivations in this chapter indicates that this claim is somewhat preposterous. Only by describing the transformation steps in a very abstract way (using just natural language) and by considering very general specifications, can the developments be reused. The central concept is *similarity*, and several definitions of this informal notion are given, each leading to a particular kind of reuse of derivations. Variants of a derivation of linear search lead to several interesting search algorithms, culminating in derivations by reuse of

two complicated string matching algorithms.

Chapter 5, "*Intersections of sets and bags of extended substructures – a class of problems*" generalizes the *specification* of pattern matching. It describes a class of problems that can be viewed as a generalization of pattern matching problems. The essence of pattern matching is considered to be the intersection of a particular set with a bag (multiset) of extended substructures of a structured object. The set contains the patterns, the extended substructures are possible occurrences, extended with labels that mark their positions in the original object. This leads to the first ideas on an (interesting) theory on (extended) substructures. It is shown how the abstract description of this class of problems lends itself to calculation in a BMF style. Also, clearly exhibiting the basic structure of such problems facilitates connecting them with various solution strategies.

Chapter 6, "*Solving a combinatorial problem by transformation of abstract data types*" gives an application of techniques from the area of formal program development in a different area, viz. combinatorics. By describing a given combinatorial problem in terms of abstract data types with equivalences, and transforming those data types, a reduction to a known problem is obtained. Abstract data types proved to be a more suitable specification mechanism in this case than context free grammars, since arbitrary equivalences could be introduced on the data types.

Curriculum Vitae

7 mei 1966
Geboren te Warns (Hemelumer Oldeferd, thans Nijefurd).

3 juni 1983
V.W.O.-diploma, C.S.G. Oostergo, Dokkum.

29 januari 1988
Doctoraaldiploma (ir.) Informatica, Universiteit Twente.

15 februari 1988
In dienst bij de Nederlandse organisatie voor Wetenschappelijk Onderzoek (N.W.O.) als onderzoeker in opleiding in het N.F.I.-project Transformationeel Programmeren (STOP – *Specification and Transformation of Programs*), projectnr. 63/62-518.

ISBN 90-9004747-6