# Logic and Proof Method of Recursion

*Birgit Schieder*

Institut für Informatik
der Technischen Universität München

# Logic and Proof Method of Recursion

*Birgit Schieder*

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. T. Nipkow Ph.D.

Prüfer der Dissertation:

1. Univ.-Prof. Dr. M. Broy
2. Univ.-Prof. Dr. M. Paul

Die Dissertation wurde am 5. Juli 1994 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 10. Oktober 1994 angenommen.

# Kurzfassung

Ziel der Arbeit sind ein Kalkül für Rekursion und eine Methode zu dessen Anwendung, die für die Programmentwicklung geeignet sind.

Zuerst werden Kriterien für die Brauchbarkeit von Kalkülen in der Programmentwicklung aufgestellt. Diese Kriterien betreffen sowohl Term- und Formelsprache, als auch Schlußregeln und Ableitungsbegriff. Anschließend werden eine Sprache und ein Kalkül für rekursive Definitionen angegeben, die den gefundenen Kriterien genügen. Der Kalkül dient sowohl zum Beweisen von Eigenschaften rekursiver Definitionen, als auch zur Entwicklung rekursiver Definitionen aus Spezifikationen. Um eine systematische, zielgerichtete Anwendung des Rekursionskalküls zu unterstützen, wird eine Methode angegeben, nach der Beweise und Entwicklungen geführt werden können. Der Einsatz des Kalküls nach der vorgestellten Methode wird an einer Reihe nicht-trivialer Beispiele von Entwicklungen vorgeführt. Die Methode wird zuerst anhand der Entwicklung einer repetitiv-rekursiven Funktion aus einer geschachtelt-rekursiven, sowie der Behandlung von Hoares Problem der zwei while-Schleifen vorgestellt. Daran schließen sich zwei umfangreichere Beispiele an: die Entwicklung eines Übersetzers für eine Sprache mit Rekursion und die Entwicklung einer operationellen Semantik aus einer denotationellen.

# Danksagung

Herrn Prof. Dr. M. Broy danke ich für die Anregung zu diesem Thema sowie für zahlreiche Diskussionen und Ratschläge. Herrn Prof. Dr. M. Paul gebührt mein Dank für Diskussionen über Programmentwicklung und die Übernahme des zweiten Gutachtens. Meinen Kollegen Herrn Dr. H. Hußmann und Herrn B. Schätz danke ich herzlich für Diskussionen bzw. das sorgfältige Lesen einer Vorversion.

# Contents

# Chapter 0

# Introduction

Today there is general agreement that program development is important and difficult. Much research has been done and still is done, in order to make program development reliable and manageable. The main ingredients of program development are a formal calculus, which ensures correctness of development, and a method for its disciplined, goal-directed application.

The essence of formal program development is that a program is formally specified, and a program in executable notation is proved to meet that specification. It is generally agreed that the executable program should not be guessed and afterwards be proved to meet the specification. Instead, the program and its proof should be developed hand in hand from the specification. Of course, thought and insight into the problem domain are needed in the development of a program. As is well-known, program development cannot completely be mechanized. Therefore human guidance is indispensable to program development.

The next step towards systematic program development consists in basing decisions rather on the shape of formulae than on insights into the problem domain. Certainly, knowledge of the problem domain cannot completely be replaced by consideration of the shape of formulae. But methods for program development should take the shape of formulae into account as much as possible in order to guide the development process. There is evidence that syntactic considerations can avoid the blind search for ideas to a large extent.

Thus, a calculus for program development should be accompanied by a method for its application. The method should divide the development as clearly as possible into routine steps, and steps that require thought. The latter should be guided as much as possible by the shape of formulae. So, more systematic program development could be achieved.

Today many calculi exist for program development and proof. But they are rarely accompanied by a method for their application. Undoubtedly, it is difficult to give a method that covers program development in general. But one could concentrate attention on particular development tasks.

In this work we will concentrate on a special form of specifications, and find a development method for them. We have chosen formulae of predicate calculus that contain inequations

$$t \sqsubseteq u$$

or equations

$$t \equiv u$$

between terms. Those terms may contain recursive definitions. The task is to develop recursive programs for the unknowns in the terms.

Into this class of specifications fall, for instance, compiler specifications. They can be stated as follows: the value of the source program is less than or equal to the value obtained by executing the compiled program on the target machine. This example already indicates the practical importance of the mentioned specifications.

A method for developing programs from specifications essentially depends on the underlying calculus. It can only be as good as the underlying calculus. Therefore any methodological consideration must start with the choice of a suitable calculus.

When studying the literature, one is confronted with a lot of different rules for recursion. They are of quite different nature: the spectrum of rules ranges from very simple ones to very technical, elaborate ones. Some of those rules are intended for development, most of them, however, for proof.

But hardly any rules are equipped with methodological directions for their use. Sometimes strategies are given, but they are merely shown in a number of examples; they are not generalized into methodological rules. This lack of methodological rules concerns proof, but to a much larger extent it concerns program development.

Proofs of properties of recursively defined objects are not easy, in general. In view of the great amount of rules in the literature, it is difficult to find a suitable rule in a particular proof situation, and to combine rules in order to reach a certain goal. A typical example of such a difficult proof is compiler correctness. It has turned out that being merely provided with the set of rules known from the literature, with knowledge of the problem domain, and with operational intuition, one can hardly find a compiler correctness proof.

The situation is no better, if developments are to be found instead of proofs: in proofs, the solution of a problem is given, and it must be proved that it is indeed a solution; that is, it must be proved that the solution meets its specification. In developments, however, the solution and its proof must be found.

Our aim is a method for the development of programs from specifications of the form given above: as mentioned before, the inequations and equations may contain recursive definitions and unknowns, for which programs must be developed. Of course, these programs themselves may contain recursion.

We want the development to be as systematic as possible. Ideally, developments should not be found by striking ideas and deep insights into the problem domain; instead they should be found by systematic analysis and design. Even more rigorously, development should largely be based on the shape of formulae, as we have stated before for program development in general. The whole development process must be as calculational as possible.

This idea of calculational development based on syntactic considerations is similar to solving differential equations: when solving a differential equation, one proceeds according to its syntactic structure; methods exist that describe how to proceed. Although thought is still needed, the methods help to derive solutions systematically, and with a minimum of own ideas.

The work is organized as follows:

**Chapter 1.** As stated above, the literature provides many rules for proving about recursion. They differ much in their shape and complexity. It is not clear which rules are most suitable for proof and development. Hence we cannot start from a given set of rules. We must first find out which of them are most convenient for practical use. This question cannot be answered purely by experiment since the number of experiments we can do is too restricted. Therefore we first search for criteria of the practical usefulness of calculi. We do so for program development calculi in general; hence chapter 1 is not restricted to calculi for recursion.

**Chapter 2.** We first give a language with recursion that respects the criteria of chapter 1. Thereafter we define formulae and rules for recursion. They are chosen according to the usefulness criteria. Since we use essentially an enriched predicate calculus, we will not present the whole calculus, but only those rules that refer to recursion or that are particularly important to our development method. We will also discuss the reasons for our selection of rules.

**Chapter 3.** We apply our development method to a number of non-trivial examples. Among them are a compiler development, and the development of an operational semantics from a denotational one. They have been chosen for two reasons: firstly, they are difficult enough to be a touchstone for our development method. Secondly, they are not known from the literature, and thus are hopefully interesting in their own right. Other examples are the transformation of an intricate nested recursion into tail-recursive form, and the simplification of the sequential composition of two while-loops. All solutions are developed by the same method.

**Chapter 4.** After having shown the development method in various examples in chapter 3, we describe the method in chapter 4. We first describe the general proceeding we have chosen for program development. Then we explain the method for development of recursive definitions, which we have used throughout our examples. Finally, we discuss methodological reasons for which we have excluded certain rules from the calculus.

**(In)dependencies.** This work has been written in a way to ease independent reading of its parts. Chapter 1 is primarily intended for the reader interested in program design calculi as an object of study, and in the decisions underlying the present work. Chapter 2 gives the foundations that are used throughout all our examples; readers, who are merely interested in application, can confine themselves to chapter 3 and chapter 4, and read chapter 2 only by need. Chapter 3 and chapter 4 can be read by arbitrary interleaving.

# Chapter 1

# What makes a useful calculus?

As stated in the introduction, our ultimate goal is to give a calculus of recursion and a method for its application suited to program development.

In this chapter and the next we shall address the calculus. We are heading for a recursion calculus that is tailored to practical use. Unfortunately, the literature does not present us with criteria for the usefulness of calculi. Of course, there are a lot of calculi for program development, but they are primarily intended to put program development on a formal basis. The question, what properties of a calculus support proof finding and program design, is of secondary interest in those calculi. Consequently, we cannot expect to be provided with a list of such properties by the literature. The pragmatic side of proofs has even almost been denied in the past. An exception to this tradition is van Gasteren's work [28]. But whereas she explores the general nature of presentation and design of mathematical proofs, we want to find out in what way the formal calculus can contribute to proof design.

Therefore, we must first establish criteria for the usefulness of calculi in program development. Toward this end, we study the rôle of formal methods in program development. Then we derive criteria that we consider important for calculi to meet their rôle in program development. In doing so, we shall not concentrate on calculi of recursion, but extend our considerations to program development and verification in general. Hence this whole chapter is not restricted to the field of recursion.

One could also imagine to find useful calculi purely by experiment: one could test different calculi by experiment, and choose those that turn out to be most useful. But we take the analytical approach, which has been described above, for two reasons: firstly, the number of experiments we can do is strongly restricted, because we are our only guinea-pigs; secondly, we hope that it will be helpful in the design of other calculi to have criteria for their usefulness. In spite of taking the analytical approach, the criteria found by theoretical investigations are supported by experiments.

Let us now explain how we will use some central notions of this chapter. By *program development* we mean a formal, stepwise development of programs from specifications. By *program verification* we mean a formal proof that a property of a program holds. Our notion of "program development" includes "program verification". This viewpoint is possible since we adhere to a notion of flexible program development: performing a step and verifying it afterwards is a legal design step. By the term *proof* we refer both to a whole program development and to a single step in a development. Throughout this work we understand proofs as completely formal objects as opposed to the often non-formal notion of proof in mathematics. Moreover, we always use the term *programmer* in the sense of "a person who develops a program from a specification".

We do not presuppose a special kind of calculus. Whenever we say *calculus*, we include verification calculi, refinement calculi, and transformation calculi. Therefore, besides formulae, terms and programs can be the objects of manipulation, as they are in transformation calculi. For convenience we will always refer to the objects of manipulation as formulae, because we shall work with such a calculus afterwards. But one can equally imagine terms as objects manipulated by the calculus.

We begin our investigations by studying the rôle of proofs and calculi in program development as far as they are independent of the proof environment. Thereafter we determine the proof environment on which we will base our considerations throughout this work. Then we derive criteria for the usefulness of calculi from their rôles in program development. Finally, we examine in how far these criteria are influenced by our choice of the proof environment.

In chapter 2 we will apply our results of this chapter in order to build a calculus of recursion.

## 1.0 The rôle of proofs and calculi in program development

First we study the rôle of proofs in program development. We then derive requirements on proofs, such that proofs meet their rôle in program development. Finally we examine the rôle of calculi in program development, and similarly derive requirements for them.

### Rôle of proofs

The rôle of proofs in program development is at least threefold:

1. Proofs guarantee that a property of interest holds.
2. By proving, programmers should understand *why* their solutions (i.e. final programs and intermediate steps of developments) are correct.
3. Proofs serve as formal documentation.

Now we briefly comment on these rôles, and discuss how far they deviate from those of proofs in other areas.

1. Of course the first rôle of proofs is not specific to program development; it is shared by all notions of formal proof.

   In mathematics proofs often are non-formal and thus their correctness cannot be stated formally. Therefore, in [47] mathematical proof is regarded as a "social process": only after a proof has been studied and accepted by other mathematicians, it can be considered correct. In our opinion such an informal notion of proof is unacceptable in program development for two reasons: firstly, proofs in program development are not studied by (enough) other programmers so that they are not subjected to a process of correction. Secondly, proofs in program development typically contain too much detail to be reliably checked by other programmers. (At this point mechanical proof support becomes important.)

2. Aid of comprehension is an essential task of proofs in program development for several reasons:

   By understanding why a certain step in the development is correct, the programmer will get a deeper insight into the problem at hand. This reinforcement of the programmer's understanding may help in further development of the program.

   Attempts to prove false conjectures must contradict the programmer's wrong understanding as clearly as possible. Comprehensible proof attempts facilitate the detection of why a wrong conjecture does not hold, and hence help to correct it.

   Moreover proofs that reinforce the programmer's understanding of a special problem will also increase the programmer's overall experience in program development.

   Proving will become a rewarding activity for the programmer, if it increases understanding and experience. The benefit of improving their programming ability should motivate programmers to do proofs.

   The rôle of proof as a vehicle for understanding is of minor importance in mathematical logic and mathematics. In pure logic proofs are mere formal objects. (This general judgement does not deny that there are works in mathematical logic on pragmatic aspects of proofs, first and foremost Gentzen's work on natural deduction [29].) When developing a theory, mathematicians usually are solely interested in correct proofs, and do not strive to reveal the real argument behind the proved theorem. It is beyond the scope of this work to discuss this common practice in mathematics; instead we refer to van Gasteren's work [28].

3. In its characteristic as formal documentation, a proof provides a guarantee of the correctness of the derived program. As such, the proof may sometime later be consulted by the original programmer as well as by others.

Sometimes an already existing program development is changed in order to replace a design decision by a new one. Then the proof as a record of the program development must be examined.

Consequently, the proof is also a means to acquaint other people with the development, that is, as an aid to their understanding of the development. Their understanding must even reach so far that they can actively work with the development.

The rôle of (formal) documentation is also indispensable to proofs in mathematics. Proofs of mathematical theorems are primarily studied by other interested mathematicians. The engineer, who uses a theorem, usually relies on its correctness without reading its proof. Moreover, proofs in mathematics are typically not redesigned or changed in order to derive new theorems. By this passive rôle of documentation mathematical proofs differ from proofs in program development.

**(End ”Rôle of proofs”.)**

### Requirements on proofs

In order to formulate requirements that should be fulfilled by proofs in program development, we make explicit our expectations of the programmer.

Programmers are assumed to have good capabilities for working formally. In addition, equipped with a suitable formal framework and method for program derivation, they need to be able to make design decisions that lead to a correct and efficient program. This characterization matches exactly the profile of engineers. In contrast to mathematicians, programmers are not supposed to come up with solutions found by ”eureka”.

Now we are ready to derive requirements that are imposed on proofs in program development by their rôles that have been identified above.

- First of all, proofs have to be correct in the following sense: only properties that hold may have proofs.

- Proofs should be done on a level of abstraction that the programmer's understanding can be built on. In particular programmers should not be confronted with the whole theory (in our case fixed point theory) that underlies the language they use. That is, semantic details are to be kept away from programmers. Instead programmers should be provided with properties that they are inclined to accept as first principles.

- The profile of programmers implies that proofs should be done as systematically as possible. Proofs should be developed by thorough analysis and design, and not by searching for a striking idea. Ideally one is heading for completely systematic, almost computational proofs.

  Systematic proof design is, in general, not considered important in mathematics. On the contrary, mathematicians often proudly present proofs that contain "ingenious tricks". Discussing this attitude is beyond the scope of this work; again the reader is referred to van Gasteren's book [28]. But it should be obvious from the above stated capabilities of programmers that tricks are to be kept out of proofs in program development.

- For the purposes of comprehensibility and documentation, proofs should be well-designed. Well-designedness includes that proofs should be as explicit as possible about their arguments and decisions, and arrange them in a structured, accessible way. Only such arguments should enter a proof that are actually needed for the proved theorem to hold. Moreover design decisions should be made as clear as possible.

- Proof should be an interesting, pleasant task for programmers. It should even be fun.

Do we stand a chance that there exist proofs that satisfy our requirements? We do certainly, since our requirements are not contradictory as shown by experiment. The requirements for systematic proof design and for well-designed proofs even seem to be two sides of the same coin.

**(End "Requirements on proofs".)**

### Rôle of calculi

In order to be able to do proofs that meet our requirements, we need a suitable calculus as well as a method for its application.

Finding criteria for a suitable calculus needs clarification of the rôle of calculi in program development:

- The calculus is a tool to "produce" proofs that fulfil the requirements that we stated above.
- The calculus also plays a rôle in its own right: It is a tool to be used by programmers. As such it considerably influences the process of finding proofs.

In particular, in our investigation the calculus is not an object of study in the same sense as calculi in mathematical logic. Whereas logicians are interested in the abstract properties of calculi, we are mostly concerned with their properties relating to application.

**(End "Rôle of calculi".)**

### Requirements on calculi

As we have done for proofs, we now state requirements for calculi in their rôles in program development:

- Of course the calculus has to be correct in the usual sense that only valid formulae are derivable.

  Besides this formal notion, correctness has also a pragmatic aspect: users may make mistakes when applying the (formally correct) calculus. Hence application of the calculus should be as few error-prone as possible.

- The rules of the calculus must provide a basis for the programmer's thought. In arithmetic, for example, thought and understanding are based on laws such as $x + y = y + x$. Similarly, the rules of a calculus for program development must serve as the basis for the programmer's thought and understanding. Hence understanding ought to be based on formula manipulation rather than formula interpretation. We are well-acquainted with this distinction in arithmetic, where proofs are guided by the laws, and not by the knowledge of a model of the natural numbers.

- The calculus must support systematic proof design.
- Moreover, as a tool the calculus has to be handy. Often the formal calculus is felt to be a burden to the programmer. Instead the calculus should free the programmer for those parts of the development that require ingenuity.

These are only very general requirements. What they technically mean for the calculus will be investigated in section 1.2. First we determine the proof environment.

Undoubtedly, a mere formal calculus cannot guarantee that proof development always ends up with a proof that has our desired properties. Likewise, good proof design cannot solely be assured by the calculus. For both purposes the calculus also needs to be applied in a disciplined manner. Hence the calculus ought to be accompanied by a method for its application. We shall consider methodological aspects in chapter 4. They will be customized to proofs about recursive definitions since this is our primary interest.

In the current chapter, however, we stick to the calculus.

**(End "Requirements on calculi".)**

## 1.1 Choice of the proof environment

As the calculus is a tool, its handiness seems to depend upon the environment in which it is used. Therefore we now determine the proof environment.

The main existing proof environments are pen and paper on the one hand and mechanical systems on the other hand. Throughout this work we base our considerations on pen and paper as proof environment for the following reason.

It is well-known that in calculi that comprise predicate logic proofs cannot completely be automatized. Hence proofs have at least partly to be done by humans. Therefore theorem proving ought to be tailored to their needs. Moreover, proofs and calculi should meet our requirements of the previous section. In order to get rid of peculiarities of mechanical support, we choose pen and paper as the proof environment that allows focusing on human needs most genuinely.

We do not doubt the usefulness of mechanical proof support. But we are of the opinion that mechanical systems should be adapted to the way in which people do proofs. Whenever human interaction is needed in a mechanical proof, the user should find a proof status that could stem from a proof done by hand. The users must not be obliged to familiarize themselves with the details of the mechanical proof procedure. An easily understandable proof status becomes even more important when a proof attempt fails. In that case an understandable proof status can help the user to get the theorem right. As we saw in the previous section, not only the existence of a proof but also the proof itself is of interest in program development. Hence proofs done with mechanical assistance also have to fulfil the requirements on proofs that we stated in section 1.0.

Consequently, any methodological consideration of proof should start with pen and paper as proof environment.

## 1.2 Criteria for the usefulness of calculi

Now we establish criteria for the usefulness of calculi based on the requirements stated in section 1.0. We again consider program development calculi in general, and not only calculi of recursion. In preparation of the next chapter, our main examples, however, will be taken from the field of recursion.

We briefly repeat our requirements on calculi. A calculus for program development must be

- correct,
- a good foundation of thought and understanding,

- a good support in systematic proof design, and
- a handy tool.

Having fixed the proof environment, we are ready to concretize these requirements. We seek for criteria that make calculi fulfil the requirements, provided the calculi are used with pen and paper. In section 1.3 we shall discuss in how far the proof environment takes effect on the usefulness criteria.

As correctness in the usual sense of mathematical logic is well-known, it needs no further consideration here. Of course, when designing a calculus, one has to ensure that all rules are correct.

Let us begin with the requirement for handiness. Since we have chosen pen and paper as proof environment, the calculus must be easily memorizable. It is important for the programmer to have a good survey of the calculus. In every proof situation the syntactically applicable rules should suggest themselves. The programmer must not be obliged to start an annoying process of remembrance and enumeration of the rules in the calculus. If the programmer had to study a catalogue of rules in order to find the ones that are applicable in the current situation, all motivation to do proofs would go down very soon. Whether a rule is applicable to a certain proof state must easily be seen, and must not require complicated syntactic comparisons. In addition, the application of a rule must be a simple syntactic manipulation.

This syntactic familiarity with the calculus is necessary not only for handiness, but also for systematic proof design. When relieved of the search for syntactically applicable rules, the programmer can fully concentrate on the choice of rules in favour of the intended aim.

Systematic proof design, however, requires an even more intimate familiarity with the calculus. Programmers must be so familiar with the calculus that they can assess the profit of rules in order to reach a certain goal. The calculus must enable the programmer to select an appropriate rule at any stage of a proof, and even to foresee and plan the whole structure of a proof beforehand. (Of course, we do not claim that all proof attempts will be successful, even if the programmer is familiar with the calculus. Familiarity, however, helps to understand why a proof attempt fails, and then to correct it.)

Familiarity with the calculus, supported by a proof method, is an essential prerequisite of systematic proof design. Therefore, we will now investigate what properties of a calculus let the programmer become intimately familiar with it.

Now we give a number of criteria that we have found out to be important for the usefulness of calculi. We arrange these criteria according to the affected constituents of the calculus: starting with formulae, we continue with rules, and then with the calculus as a collection of rules, until we reach the structure of derivations.

### 1.2.0 Proof-oriented syntax

Parsing and manipulating formulae are the main activities in proofs. Firstly, these activities have a purely technical aspect: formulae should be easy to parse by the eye, and comfortably manipulable by a human user. These properties contribute to the handiness of the syntax. Secondly, finding proof steps and taking design decisions are also aspects of manipulation. Apart from supporting the technical side, syntax has to assist in proof design. Of course, technical manipulability is a necessary prerequisite for design. But, in addition, a customized syntax can aid to find a proof for example by the mere shape of a formula.

Since programming languages are primarily designed to program *in* them, instead of *into* them, we will first look for criteria for a proof-oriented syntax. This search will lead us to the following criteria:

- combinatorial freedom
- homogeneous syntax
- explicitness of formulae
- economy of syntactic categories
- syntactic sugar

### Combinatorial freedom

Combinatorial freedom was identified as essential for the convenience of formula manipulation by van Gasteren [28]. *Combinatorial freedom* means that one is not obliged to use certain distinct concepts or operators always together, but instead is free to use them separately or to combine them as needed.

Let us illustrate combinatorial freedom by an example from the field of recursion.

**Example 0** When considering how recursion appears in most programming languages, we find combinatorial freedom violated.

In programming languages occurrence of recursion is typically confined to recursive declarations. Apart from syntactic variations, recursive function declarations have the form

$$\textbf{funct}\ f = (x) : t\ ,$$

where identifier $f$ may occur in term $t$, and $x$ is a formal parameter.

The semantics of such a function declaration is the binding of a certain function to identifier $f$. That is to say, the operator **funct** joins two distinct operators together:

- selection of a certain semantic function,
- binding of a function to an identifier.

Combination of these two operators may be appropriate for coding problems in a programming language, but it is improper to proof. Declarations as above compel one to refer to the recursively defined function exclusively by the identifier $f$ within terms, and to remember the binding as the context one is working in.

On the contrary, direct reference to the recursively defined function would be rendered possible by separation of the two operators. In this way the defining expression of the function could immediately be written in terms without introduction of a binding. Moreover, when we speak of the equivalence of two recursive function declarations, we actually mean the equality of the defined functions, and are not interested in the bindings to identifiers. Hence being able to refer to the recursively defined function directly turns out to be valuable in proofs.

As regards binding we do not even need a new operator. Since we will have a logic with equality, binding can be expressed by explicit use of the equality sign. Therefore, in this example combinatorial freedom is for free: the number of operators does not increase.

**(End of example.)**

This example shows how a flexible syntax can be helpful in proofs by allowing to express things as directly as needed.

**(End "Combinatorial freedom".)**

### Homogeneous syntax

By *homogeneous syntax* we mean that syntactic similarities should be mirrored by the semantics. To put it the other way, we do not want semantically different concepts to be expressed by a similar syntax.

The reason for requiring a homogeneous syntax again lies in our striving for conveniently manipulable formulae. Proving is rendered difficult by syntactic constructs that look similar but obey quite different laws. In this way formula manipulation becomes a less mechanical activity than it could be. Even worse, the syntactic similarities can be misleading and provoke errors.

**Example 1** As discussed in example 0, the semantics of a declaration

$$\textbf{funct } f = (x) : t$$

is the binding of identifier $f$ to a function. Thus the binding is valid outside the declaration, too.

Let us compare the operator **funct** to standard binding operators of functional languages and of predicate logic. All of the operators in

$\lambda x.t$ (function abstraction),

$\forall x.A$ (universal quantification), and

$\exists x.A$ (existential quantification)

bind the identifier $x$ only *within* these terms and formulae. Outside these terms and formulae, the bindings are invisible.

The different binding scopes lead to different rôles of the bound identifiers: The identifier bound by one of the operators $\lambda$, $\forall$ and $\exists$ merely is a means to define a semantic object. It may be exchanged by any other identifier (provided no name clashes are introduced) without effect to the semantics. On the contrary the identifier bound by the operator **funct** is itself part of the semantics, and consequently cannot be substituted by another identifier.

Hence, despite the syntactic similarity in their occurrence, the operator **funct** on the one hand, and the operators $\lambda$, $\forall$ and $\exists$ on the other hand behave quite differently.

This inhomogeneity can be avoided by replacing the operator **funct** by a new operator, say **rec**, such that the semantics of

**rec** $f.(x) : t$

merely is a semantic function, instead of a binding to the identifier $f$.

**(End of example.)**

Syntactic inhomogeneities can be eluded by restricting the semantics of similar syntactic constructs to the "common semantics", and by introducing additional operators for the differing parts of the semantics. This is what was done in the above example.

**(End "Homogeneous syntax".)**

### Explicitness of formulae

We call a formula *explicit*, if it is independent of context.

**Example 2** Let $f$ be recursively declared by

$$\textbf{funct } f = (x) : t \; .$$

Then the formula

$$\forall x : f(x) = u \; ,$$

where $u$ is a term, is *not* explicit about $f$, because it refers to the $f$ declared in the context, and, in general, does not hold for all $f$.

Having used the operator **funct**, we are unable to make the formula explicit. As discussed above, the information about $f$ cannot be included in the formula for syntactic reasons (not even as a premise). Since the operator **funct** prevents us from writing explicit formulae, we have another reason for refusing it.

**(End of example.)**

Explicit formulae are desirable, because they make available all information in a compact form. In order to find the next step in a proof, one must only look at the current formula; one need not keep in mind a context or switch to a context that stands elsewhere in the proof. Proofs are hampered especially, if the context changes from time to time.

Therefore we consider it important for a proof-oriented syntax that it allows explicit formulae.

Although demanding a syntax, in which explicit formulae can be written, we do not say that one should *always* use explicit formulae. Sometimes it is for example more convenient to take apart a premise from a formula, and to make it a general assumption. The point is that explicitness should be enabled by the syntax. Then one can decide freely to make a formula explicit or not. It is unacceptable to be obliged to use a non-explicit formula only by syntactic reasons. As we have seen in the above example, the operator **funct** prevents explicitness, and therefore must be refused.

When using contexts, one should make sure that they do not change too often. But if contexts are forced by the syntax, they are uncontrollable, and thus may change continuously. Frequent context changes are error-prone and unhandy.

**(End "Explicitness of formulae".)**

### Economy of syntactic categories

By *economy of syntactic categories* we mean that in the object language the number of syntactic categories should be kept small.

**Example 3** Besides a syntactic category of terms, programming languages usually contain a syntactic category of declarations. We have seen in example 0 that one can avoid the syntactic category of function declarations by introducing a recursion operator in terms.

**(End of example.)**

Parsing and manipulating formulae become easier, when the number of syntactic categories decreases.

**(End "Economy of syntactic categories".)**

### Syntactic sugar

By *syntactic sugar* we mean that syntactic patterns that occur frequently in applications are abbreviated by a new syntactic construct. Introduction of syntactic sugar is particularly attractive, if it makes another syntactic construct surplus.

Let us consider an example from the field of recursion.

**Example 4** Let **fix** be an operator that allows to define functions recursively by writing

$$\mathbf{fix}\ (\lambda f.\, \tau)\ .$$

Thus, application of **fix** to a $\lambda$-abstraction is a frequently occurring syntactic pattern. If we introduce a special syntax for this pattern, say

**rec** , then we can rewrite the original function definition by

**rec** $f.\ \tau$ .

Introduction of **rec** makes **fix** surplus as we shall see in the next chapter. Thus **fix** can completely be eliminated from the language.

**(End of example.)**

Syntactic sugar makes formula parsing and manipulation more convenient, because formulae can be kept concise. In addition, proof rules for the original syntactic constructs can also be combined into new rules for the syntactic abbreviation. This adaptation of rules to frequently occurring syntactic patterns leads to shorter proofs.

At first sight, syntactic sugar seems to contradict our requirement for combinatorial freedom. But it does not for the following reason: Combinatorial freedom says that two (or more) semantically independent operators should not be glued together by the syntax. Contrastingly, when syntactic sugar is introduced, one has a single operator in mind, and wants a convenient syntax for it.

A typical effect of introducing syntactic sugar and eliminating other constructs is that non-standard cases become more complicated than with the original construct. Therefore, syntactic sugar must be chosen very carefully, that is, the standard cases must carefully be discerned from the non-standard ones.

**(End ”Syntactic sugar”.)**

These criteria help the programmer to become syntactically familiar with the language. In this way the programmer can more intensely concentrate on those tasks of program development that require ingenuity. Thus, the above criteria contribute also to systematic proof design.

### 1.2.1 Simple rules

As usual, rules consist of premises, a conclusion, and possibly some applicability conditions. We tacitly include transformation calculi by allowing terms, declarations etc. as premises and conclusions of rules.

In this section we consider rules purely syntactically. No semantic knowledge is needed to study the examples.

The following two examples give a first visual impression of the difference between complicated and simple rules.

**Example 5** The following complicated rule for recursive declarations is taken from [41]. It is only intended for purpose of illustration here, and will not be studied further.

Let $\Sigma$, $\Sigma'$, $s_1$, $s_2$, $s_3$, $s_4$ and $F_1$ be meta-variables. Let $\Omega$ and *dom* be a value and a function respectively. Let $S$, $V$, $A$ and $\Phi$ be meta-variables bound outside the rule. (It does not concern us here what all these variables stand for.)

$$\frac{}{\Sigma \equiv_{S \cup \{(s_4, \Omega)\}} \Sigma'}$$

```
where $\langle s_1, s_2, s_3, F_1 \rangle$ is a WUF($S$)-transformation from $\Sigma$ to
    $\Sigma'$, and
    $V'$ is the set of variables occuring in $\Sigma$, and
    $s_4$ is the term constructed by the algorithm:
    $dom(s_4) \subseteq dom(s_1) \cap dom(s_3)$
    $\forall w \in dom(s_4)$
      either $s_3(w) \in V'$ then $s_4(w) = s_3(w)$
      or $s_3(w) \in A$, if $s_1(w) = s_3(w)$ then $s_4(w) = s_3(w)$
                         if $s_1(w) \in V'$ then $s_4(w) = s_1(w)$
                         if $s_1(w) \in \Phi$ then $s_4(w) = \Omega$
      or $s_3(w) \in \Phi$, if $s_1(w) \neq s_3(w)$ then $s_4(w) = \Omega$
                                         else $s_4(w) = y$
                         where $y$ is a variable of $V - V'$ with
                         no occurence in $s_4$
```

The description of "WUF-transformation", and of the syntactic correlation between $\Sigma$, $\Sigma'$, $s_1$, $s_2$, $s_3$, $s_4$ and $F_1$ needs another half page, and a graphical explanation.

**(End of example.)**

We contrast this complicated rule with two simple ones.

**Example 6** The first of the following rules is well-known from predicate logic, the second one is a so-called "fixed point induction rule".

$$\frac{A \Rightarrow B}{A \Rightarrow \forall x\, B} \quad \text{where } x \text{ is not free in } A$$

$$\frac{x \sqsubseteq u \Rightarrow t \sqsubseteq u}{\mathbf{rec}\ x.\, t \sqsubseteq u} \quad \text{where } x \text{ is not free in } u$$

**(End of example.)**

Now we will examine the simplicity of rules more systematically. Our search for factors that make a rule simple will lead to the following criteria:

- obvious syntactic applicability
- decidable applicability conditions
- economy of concepts
- clear presentation
- separation of elementary rules
- small rules

**Obvious syntactic applicability**

In [19] Courcelle writes about transformations of recursive program schemes:

> Courcelle and Kott have given syntactical conditions [...]. Since these conditions are quite technical we do not even state them.

The mentioned "syntactical conditions" are intended for use in program transformations. We doubt that rules that are too technical to be included in a book on theoretical computer science can be helpful in practice. Let us make this point more precise.

We first define what we mean by *syntactic applicability.* Assume we use the calculus in a forward manner, that is, we move from the premises of a rule to its conclusion. Then *syntactic applicability* of a rule to a proof situation means that some derived formulae syntactically match the premises of the rule, and that all decidable applicability conditions of the rule hold. The definition is analogous, if we work backwards, that is, from the conclusion of a rule to its premises. In that case *syntactic applicability* means that the current formula matches the conclusion of the rule, and that all decidable applicability conditions hold.

We say that the syntactic applicability of a rule is obvious, if in every proof situation the syntactic applicability can be perceived by the eye. If the rule turns out to be applicable, the resulting formula must also be perceivable by the eye. In particular, neither the applicability check, nor the application itself involve non-trivial operations. One only has to compare the actual proof situation to the rule one has in mind.

**Example 7** The syntactic applicability of the complicated rule of example 5 is not obvious. The computation of the term $s_4$ is too complicated to be performed by the eye.

Contrastingly, the application of the two simple rules given in example 6 is obvious. As one knows from experience, the free variables of a formula can be determined by the eye.

**(End of example.)**

A typical technique that leads to non-obvious syntactic applicability is the use of labels: often for applicability of a rule, certain occurrences of identifiers in a formula must be labelled, others must not. Application of the rule then erases certain labels and introduces others. Such syntactic applicability conditions are usually too complicated to be perceived at a glance.

Non-obvious applicability is not only caused by complicated applicability conditions. Of course, complicated syntactic conditions can also directly be coded into the premises or the conclusion of a rule.

**(End "Obvious syntactic applicability".)**

**Decidable applicability conditions**

Often the applicability conditions of a rule are even undecidable.

**Example 8** The complicated rule in example 5 contains an undecidable applicability condition:

$$\langle s_1, s_2, s_3, F_1 \rangle \text{ is a WUF}(S)\text{-transformation from } \Sigma \text{ to } \Sigma'$$

An equality proof is needed in order to show the existence of a WUF($S$)-transformation from $\Sigma$ to $\Sigma'$. In fact, a separate transformation calculus hides behind the applicability conditions.

**(End of example.)**

As the example shows, undecidable applicability conditions are a means to take difficult parts out of the calculus. But then a second calculus for the applicability conditions must be added. Hence one obtains a calculus consisting of two levels. But a calculus with two levels will usually be too complicated for practical use.

**(End "Decidable applicability conditions".)**

**Economy of concepts**

By *economy of concepts* we mean that a rule should include as few different concepts as possible.

**Example 9** Let us compare two rules from the field of recursion in regard to the concepts they use.

As usual, the notation [./.] stands for substitution of variables by terms.

The so-called "computational induction rule" relies on natural numbers:

$$\frac{\begin{array}{c}A[t_0/x]\\ \forall n \in I\!N(A[t_n/x] \Rightarrow A[t_{n+1}/x])\end{array}}{A[\mathbf{rec}\ x.\ t/x]}$$

where $A$ is syntactically admissible in $x$,

and the sequence $(t_n)_{n \in I\!N}$of terms is recursively

defined by $t_0 \equiv \bot$,

$t_{n+1} \equiv t[t_n/x]$

The following rule (known as "fixed point induction") can be used to prove the same properties as the previous rule, but does not make use of natural numbers:

$$\frac{\begin{array}{c}A[\bot/x]\\ \forall x(A \Rightarrow A[t/x])\end{array}}{A[\mathbf{rec}\ x.\ t/x]} \qquad \text{where } A \text{ is syntactically admissible in } x$$

Experience shows that these rules often lead to quite different proofs of the same theorem. Proofs that use the fixed point induction rule are more abstract than proofs that use computational induction.

**(End of example.)**

Obviously, the more concepts a rule contains, the more difficult becomes its use, and the more concepts are introduced in proofs. Therefore we claim that economy of concepts generally leads to more abstract proofs without unnecessary detail.

**(End "Economy of concepts".)**

### Clear presentation

First we give an example of how a rule can be stated clearly or not.

**Example 10** We again do not study the meaning of the rules, but merely their syntactic shapes.

The so-called ”unfold-rule” for recursively defined functions can be formulated as follows:

$$\frac{}{\begin{array}{l} \left[\begin{array}{l} \mathbf{funct} f_0 = (x_{0,1}, \ldots, x_{0,m_0}) : t_0 \\ \vdots \\ \mathbf{funct} f_n = (x_{n,1}, \ldots, x_{n,m_n}) : t_n \end{array}\right] \sqsubseteq \\ \left[\begin{array}{l} \mathbf{funct} f_0 = (x_{0,1}, \ldots, x_{0,m_0}) : u_0[t_0/f'_0, \ldots, t_n/f'_n] \\ \vdots \\ \mathbf{funct} f_n = (x_{n,1}, \ldots, x_{n,m_n}) : u_n[t_0/f'_0, \ldots, t_n/f'_n] \end{array}\right] \end{array}}$$

where $n \geq 0$,
$\forall i : 0 \leq i \leq n : m_i \geq 0$
and $t_i$ stands for $u_i[f_0/f'_0, \ldots, f_n/f'_n]$

This rule can more simply (and, as we shall see in the next chapter, even more generally) be formulated as follows:

$$\frac{}{\mathbf{rec}\ f.\, T \sqsubseteq \mathbf{rec}\ f.\, U[T/\varphi]} \qquad \text{where } T \text{ stands for } U[f/\varphi]$$

If we had a nondeterministic substitution operator [**some** ./.] to our disposal, we could write even more succinctly:

$$\frac{}{\mathbf{rec}\ f.\, T \sqsubseteq \mathbf{rec}\ f.\, T[\mathbf{some}\ T/f]}$$

**(End of example.)**

*Clear presentation* means that technicalities are avoided as far as possible. One may only make sparing use of indices, especially of multiple indices, primes, labels etc. Their use should be restricted to cases where it is unavoidable.

The example shows how much a clear presentation contributes to readability and memorization of a rule.

**(End ”Clear presentation”.)**

### Separation of elementary rules

By *separation of elementary rules* we mean that elementary rules are not repeated in other rules.

**Example 11** A typical elementary rule is that bound variables may be substituted by other variables (provided no name clashes are introduced). Using that the names of bound variables are irrelevant, one could write

$$\frac{\mathbf{rec}\ x.\, t \equiv \mathbf{rec}\ y.\, u[y/z]}{\mathbf{rec}\ x.\, t \sqsupseteq \mathbf{rec}\ y.\, u[t[y/x]/z]}$$

instead of

$$\frac{\mathbf{rec}\ x.\, t \equiv \mathbf{rec}\ x.\, u[x/z]}{\mathbf{rec}\ x.\, t \sqsupseteq \mathbf{rec}\ x.\, u[t/z]} \quad .$$

It could be argued that the first rule is applicable in situations where the bound variables of **rec** are named differently, whereas the second rule is not. But renaming of bound variables is such a routine matter that the greater complexity of the first rule does not pay.

**(End of example.)**

Sometimes it is tempting to repeat an elementary rule, such as substitution of bound variables, in other rules. This is seemingly justified by the greater generality of the rule, because certain variables need not be the same. This shortens proofs, because e.g. renaming is unnecessary in a proof. But as elementary rules are usually applied without thinking and not written down, in fact proofs do not become shorter. But the rules become more complicated. Therefore elementary rules should not be repeated in other rules.

**(End "Separation of elementary rules".)**

### Small rules

Even if rules contain only simple formulae and obvious applicability conditions, they may still be inappropriate for practical use. Users will not become familiar with rules that contain a lot of premises and applicability conditions. Therefore rules should be small.

**(End "Small rules".)**

### 1.2.2 Concise calculus

By *concise calculi* we mean calculi that consist only of few rules, and nevertheless are powerful enough for proofs of interest.

Conciseness of a calculus helps the programmer to become familiar with it. Calculi that consist of few rules are easy to memorize. Thus, it becomes easier to find all rules that are syntactically applicable in a proof situation. Conciseness makes a calculus a more handy tool.

Not being overwhelmed by a vast amount of rules, users of a concise calculus will more easily plan and design their proofs. Hence conciseness of calculi facilitates their goal-directed application.

In addition, programmers will more readily base their thought and understanding on a concise calculus than on a huge one. A calculus that cannot easily be surveyed seems to be unsuitable as a foundation of understanding.

We provide four techniques that we consider important in order to obtain concise calculi:

- economy of syntactic categories
- parametrized rules
- compact rules
- careful addition of rules

#### Economy of syntactic categories

In section 1.2.0 we mentioned economy of syntactic categories as a criterion for convenient formula parsing and manipulation. In addition, economy of syntactic categories contributes to a concise calculus by avoiding that essentially the same rules must be stated for several syntactic categories.

**Example 12** Programming languages usually contain the syntactic categories of declarations and terms. Thus, for reasoning about equality of function declarations and about equality of terms, the typical rules of equality must be duplicated, that is, they must be stated for each of these two categories. This duplication of essentially the same rules can be avoided, if terms are the only syntactic category as suggested in example 0.

**(End of example.)**

Therefore, if objects of different syntactic categories behave essentially in the same way, one should try to unify these syntactic categories into one.

**(End "Economy of syntactic categories".)**

### Parametrized rules

By *parametrization of a rule* we mean the replacement of some rules by a single rule, such that each of the old rules is an instance of the new one. The new rule is called a *parametrized rule.*

**Example 13** We take the commutativity rules of conjunction and disjunction as a simple example:

$$\frac{A \vee B}{B \vee A} \qquad \frac{A \wedge B}{B \wedge A}$$

They can be replaced by a single parametrized rule:

$$\frac{A\;op\;B}{B\;op\;A} \qquad \text{where } op \in \{\vee, \wedge\}$$

**(End of example.)**

The benefit of parametrized rules is twofold: Firstly, the calculus becomes smaller. Secondly, the user of the calculus learns and memorizes the rules in a structured way: similarities and differences in the properties of operators are made explicit.

(Of course, a formal language for the applicability conditions of rules is needed. But it is beyond the scope of this work to define such a language.)

**(End "Parametrized rules".)**

### Compact rules

By a *compact rule* we mean a rule that includes one or several other rules as instances or combinations, and makes them surplus. In addition, we require for compact rules that they lead to shorter and more easily findable proofs than the original rules do.

**Example 14** Logic with equality provides a typical example of the replacement of a rule by a compact one.

Let $s$, $t$ and $u$ be meta-variables denoting terms. Then the well-known rules of reflexivity, symmetry, and transitivity read as follows:

$$\frac{}{t = t} \qquad \frac{t = u}{u = t} \qquad \frac{t = u \quad u = s}{t = s}$$

Leibniz's law can be added in form of the rule

$$\frac{t_0 = u_0 \quad \ldots \quad t_n = u_n}{f(t_0, \ldots, t_n) = f(u_0, \ldots, u_n)} \quad ,$$

where $f$ is a meta-variable for function identifiers, and $t_0, \ldots, t_n$, and $u_0, \ldots, u_n$ are meta-variables for terms.

Instead of the last rule we could add the rule

$$\frac{t_0 = u_0 \quad \ldots \quad t_n = u_n}{s[t_0\sigma/x_0, \ldots, t_n\sigma/x_n] = s[u_0\sigma/x_0, \ldots, u_n\sigma/x_n]} \quad ,$$

where $\sigma$ ranges over substitutions of variables by terms. In application to terms, substitutions are written as postfixes. As usual, $[./.]$ denotes substitutions.

The last rule obviously comprises Leibniz's rule as an instance.

The advantage of the replacement of the Leibniz rule by the more general one is well-known: Equality of the application of a composed function to equal terms can be proved in a single step instead of in a number of steps. Even more important, the introduction of a substitution in the new rule further facilitates and shortens proofs in special theories with equality, that is, in presence of non-logical axioms.

(From a theoretical point of view we could also eliminate the reflexivity, symmetry, and transitivity rules, because they are also included in the generalized Leibniz rule. For practical application of the calculus, however, it is more convenient to keep the three rules.)

**(End of example.)**

As indicated by our last example, compact rules tend to be more complicated than the original ones. Therefore, one must weigh the merit of a compact rule against its complexity.

**(End "Compact rules".)**

### Careful addition of rules

In calculi for practical use, careful consideration whether to add a new rule to a calculus or not, is even more important than in theoretical investigations of calculi: if a calculus intended for theoretical investigation contains unnecessarily many rules, then only skilled logicians are concerned; but if a calculus intended for practical use contains too many or too few rules, then the programmer must cope with the problems. In any case, the criteria for the addition of new rules differ considerably from calculi for theoretical investigations to calculi for practical application, as we shall see.

Two different kinds of rules can be added to a calculus:

- The new rule is such that all theorems that can be proved with the new rule could also be proved exclusively with the original rules. Derived rules are contained in this group of rules: The application of a derived rule can always be replaced by a combination of old rules, without any change in the rest of the proof.

- The new rule enlarges the set of provable valid formulae. Of course, such rules only exist for incomplete calculi. They make the calculus less incomplete.

When should a new rule be added to a calculus, if we aim at concise calculi? Of course, the new rule must meet our requirements of the remaining sections. In addition, we distinguish between the two kinds of new rules:

- A rule of the first kind should only be added to a calculus, if it is very often applicable. Moreover, we require that the new rule facilitates proofs. That is, an appropriate combination of the old rules would be much harder to find than a proof using the new rule. A further reason to add a rule is that it provides a good basis for understanding. These are the main reasons to include a new rule. The reduction of the length of proofs should only play a rôle if the reduction is considerable.

- A rule of the second kind should only be added to a calculus, if the formulae that thus become provable are of practical importance. That is, the new theorems must actually emerge in practical applications. With new rules of the second kind there is a particular danger of obtaining complicated rules. We claim that instead of addition of a complicated rule, one should rather accept a more incomplete calculus. A too complicated rule will not be used and in addition hampers familiarity with the calculus.

Whenever a rule is added to a calculus, the application of which is not obvious, the designer of the calculus should provide the user with a good method of the application of the rule. At least, examples of evidence should be given.

**(End "Careful addition of rules".)**

### 1.2.3 Local proof steps

We call a proof step *local*, if it depends only on the formulae that have already been derived. In particular, a local proof step must not depend on the proof history, that is, on the shape and structure of the preceding proof.

Let us consider an example of local proof steps.

**Example 15** In predicate calculi proof steps are local: one may derive a formula during a proof, if it is an axiom or if it can be obtained from already derived formulae by application of a rule. Particularly, the proof step does not depend on *how* these formulae have been derived.

**(End of example.)**

This notion of locality of proof steps can immediately be adapted to transformation calculi, where terms are manipulated instead of formulae: a transformation step is local, if it depends only on the terms that have already been derived.

In existing calculi, transformation steps often are non-local, because they depend on the structure of the transformation history. We give an example of this kind of non-locality.

**Example 16** Let us consider the transformation calculus that hides behind the applicability condition in example 5. This transformation calculus contains three transformation rules, say, $r_0$, $r_1$ and $r_2$. In every transformation step, rule $r_0$ may only be applied to a term $t$, if none of the rules $r_1$ and $r_2$ have been applied in the transformation that lead to $t$. Likewise, rule $r_1$ may only be applied in a transformation step, if only $r_0$ and $r_1$ have been applied in the preceding transformation. Therefore, these transformation steps are non-local.

**(End of example.)**

In some transformation calculi, the proof history restricts not only the set of rules applicable in a transformation step; the proof history restricts also the positions in terms, at which rules may be applied. Let us give an example of this kind of non-locality, too.

**Example 17** In [19] Courcelle presents a transformation technique, which he calls ”restricted folding-unfolding”. The objects of transformation are systems of function declarations. The single function declarations of a system form the positions at which transformation rules may be applied. In every transformation step, a function declaration $\Delta$ may only be transformed by use of a declaration $\Gamma$, if no transformation rule has been applied at position $\Gamma$ throughout the preceding transformation. Hence the transformation steps are not local.

**(End of example.)**

Non-locality is often avoided by introducing labels into formulae or terms. The labels are used to code the proof history or the transformation history into the formulae and terms respectively. Although this technique makes proof steps local, it does not make the calculus better: as we have seen in section 1.2.1 the use of labels violates the requirement for obvious syntactic applicability.

Our discussion suggests that we can restate locality of proof steps as *extensionality of derivation*: whether a derivation is continuable by a certain rule depends only on the derived formulae, and does not depend on the structure of the derivation.

Although being of theoretical interest, calculi with non-local proof steps are inappropriate for practical use. Non-locality offends against a number of requirements that we have stated for program development calculi:

Since a proof step may depend on a wide range of the preceding derivation, and on its internal structure as well, the legality of proof steps cannot be perceived easily. Thus the user of the calculus is hindered from becoming syntactically familiar with it. By forcing its users to take many properties into account at one time, a calculus with non-local proof steps becomes an unhandy tool. Moreover, non-local proof steps make the application of a calculus more error-prone, at least, if it is performed by hand.

When planning a proof in a calculus with an extensional notion of derivation, one must only decide which intermediate formulae to prove. In a calculus with non-local proof steps, however, the structure of the subderivations must be planned, too. Hence non-local proof steps make proof design more difficult.

In addition, non-extensional notions of derivation are too complex to provide a good basis for understanding. They will hardly be accepted by programmers as first principles of their thought.

### 1.2.4 Combination of the criteria

Unfortunately, not all stated criteria go well together.

When trying to keep a calculus concise, we may be faced with rules that are not as simple as desired. We encountered this effect when we replaced Leibniz's rule by a generalization in example 14. A slightly increased complexity of a rule is compensated if the rule is often applicable since this helps the user to become familiar with it.

Another conflict arises, when a number of simple rules are to be added to a calculus. Such a modification contradicts the conciseness criterion.

Moreover we have mentioned that careless observance of local proof steps can lead to complicated rules.

Hence the design of a calculus must be a trade-off among all criteria. We consider proof-oriented syntax, simple rules, and local proof steps as most important. Conciseness of calculi seems to be slightly less important.

Therefore a calculus for practical use will never be designed in a single step. Design of a calculus will rather be a process of construction, experiment, and adjustment. The experiments should become more and more intricate as the number of cycles increases. At least the final calculus (better some calculi before) should be accompanied by an application method.

## 1.3 Influence of the proof environment on the criteria

In section 1.1 we determined pen and paper as proof environment. All criteria for the usefulness of calculi were based on this proof environment. To what extent would a change of the proof environment influence the identified criteria?

The requirement for systematic proof design was a essential reason of each of the stated criteria. Systematic proof design is indispensable even if we change to a mechanical system as proof environment. Therefore we claim that all criteria remain relevant in a machine-assisted proof environment.

Not being trivial, the transition from pen and paper as proof environment to a mechanically supported one opens an interesting field of investigation: Having found a method for doing proofs of a certain area (e.g. proofs about recursive definitions), one could investigate how this method could best be supported by a mechanical system. That is, one should refine the method in order to exploit the mechanical support for routine steps as far as possible. These questions, however, go beyond the scope of this work.

# Chapter 2

# A calculus of recursion

Now we apply the criteria of chapter 1 in order to find a useful calculus for recursion. There exist many different languages with recursion, and many different rules for reasoning about programs written in those languages. It is far from obvious, which of them are better suited to program development than others. Therefore we use the criteria to judge the usefulness of calculi of recursion.

We will first present a language, then formulae and rules; we will discuss our choices and compare them to languages and rules in the literature. Finally we define some notation that we need in the examples of chapter 3.

We will not list an entire calculus for program development, but concentrate on the rules relating to recursion. Likewise we prove only such properties at the meta-level, which relate to recursion.

## 2.0 Object language

In this section we define a class of languages, in which we will write all our examples throughout this work. All these languages contain a recursion operator. In order to concentrate on recursion, we leave certain parts of the language unspecified, such as basic sorts and predefined functions. Thus we obtain a whole class of languages, which we can instantiate as it is needed in each of our examples. Those parts of our examples, however, which refer to recursion, are handled uniformly.

### 2.0.0 Syntax

In this section we give the syntax of the class of languages that we will use throughout this work.

We start with the type system.

Let $S$ be a set of symbols, called *sorts*, and $Bool \in S$ the sort of *boolean values.*

**Definition (Types)** The set of *types* is inductively defined as follows:

- Each sort $s \in S$ is a type.
- If $\tau_0$ and $\tau_1$ are types, then so are $\tau_0 \times \tau_1$ and $\tau_0 \to \tau_1$.

**(End of definition.)**

Types of the form $\tau_0 \times \tau_1$ stand for product types; types of the form $\tau_0 \to \tau_1$ stand for function types. (The precise definitions follow in section 2.0.1.) The set of sorts can be instantiated differently from application to application.

**Notation** The operator $\times$ binds tighter than $\to$. As usual, the operator $\to$ associates to the left such that $\rho \to \sigma \to \tau$ is parsed as $(\rho \to \sigma) \to \tau$.

**(End of notation.)**

Let $F$ be a set of symbols, called *function symbols*, such that a type is associated with each function symbol $f \in F$. For each type $\tau$ we assume a distinguished function symbol $\perp_\tau$ to belong to $F$. Whenever its type can be inferred from the context, we simply write $\perp$ for $\perp_\tau$. Function symbols of non-functional types can be considered as nullary function symbols or as constants. Moreover, let $tt$ and $ff$ be function symbols of type $Bool$.

The set $S$ of sorts, and the set $F$ of function symbols together form a *signature* $\Sigma = (S, F)$.

Let $X$ be a set of variables such that a type is associated with each variable $x \in X$.

**Definition (Terms)** The set $T$ of *terms* is inductively defined as follows:

- If $f \in F$ is a function symbol with type $\tau$, then $f$ is a term of type $\tau$.

- If $x \in X$ is a variable with type $\tau$, then $x$ is a term of type $\tau$.

- If $t_0$ is a term of type $Bool$, and $t_1$ and $t_2$ are terms of type $\tau$, then **if** $t_0$ **then** $t_1$ **else** $t_2$ **fi** is a term of type $\tau$.

- If $t$, $t_0$ and $t_1$ are terms of types $\tau_0 \times \tau_1$, $\tau_0$ and $\tau_1$ respectively, then $(t_0, t_1)$ is a term of type $\tau_0 \times \tau_1$, and $fst\ t$ and $snd\ t$ are terms of types $\tau_0$ and $\tau_1$ respectively.

- If $x \in X$ is a variable of type $\sigma$, and $t$ is a term of type $\tau$, then $\lambda x.\, t$ is a term of type $\sigma \rightarrow \tau$.

- If $t$ is a term of type $\sigma \rightarrow \tau$, and $s$ is a term of type $\sigma$, then $t\, s$ is a term of type $\tau$.

- If $x \in X$ is a variable of type $\tau$, and $t$ is a term of type $\tau$, then **rec** $x.\, t$ is a term of type $\tau$.

**(End of definition.)**

One can understand these terms informally as follows: Function symbols denote predefined functions (inclusive of constants). Variables may also occur as terms. By **if** . **then** . **else** . **fi** we denote conditional expressions. Terms of the kind $(.,.)$ denote pairs, the functions $fst$ and $snd$ denote projection to the first and to the second component of pairs, respectively. As usual, $\lambda$ denotes function abstraction, and juxtaposition denotes function application. Finally, **rec** denotes recursive definition. (The precise definitions follow.)

**Notation** We write $\doteq$ for syntactic equality on terms, and on symbol sets (such as $F$ and $X$). The negation of $\doteq$ is denoted by $\not\doteq$.

**(End of notation.)**

*Free* and *bound variables* are defined as usual, where $\lambda$ and **rec** are the only binding operators in terms.

**Definition (Substitution in terms)** The function .[./.] substitutes terms for variables in terms. It is defined by structural induction on terms as usual:

$$f[t/x] \doteq f$$

$$y[t/x] \doteq \begin{cases} t & \text{if } x \doteq y \\ y & \text{if } x \not\doteq y \end{cases}$$

$$\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \textbf{ fi}[t/x] \doteq$$
$$\qquad \textbf{if } t_0[t/x] \textbf{ then } t_1[t/x] \textbf{ else } t_2[t/x] \textbf{ fi}$$

$$(t_0, t_1)[t/x] \doteq (t_0[t/x], t_1[t/x])$$

$$(fst\ u)[t/x] \doteq fst(u[t/x])$$

$$(snd\, u)[t/x] \doteq snd(u[t/x])$$

$$(\lambda y.\ u)[t/x] \doteq$$
$$\begin{cases} \lambda y.\ u & \text{if } x \doteq y \\ \lambda y.\ (u[t/x]) & \text{if } x \not\doteq y\text{, and } y \text{ is not free in } t \\ \lambda z.\ ((u[z/y])[t/x]) & \text{if } x \not\doteq y\text{, and } y \text{ is free in } t\text{,} \\ & \text{and } z \text{ is a fresh variable} \end{cases}$$

$$(u\ s)[t/x] \doteq (u[t/x])(s[t/x])$$

$$(\textbf{rec } y.\ u)[t/x] \doteq$$
$$\begin{cases} \textbf{rec } y.\ u & \text{if } x \doteq y \\ \textbf{rec } y.\ (u[t/x]) & \text{if } x \not\doteq y\text{, and } y \text{ is not free in } t \\ \textbf{rec } z.\ ((u[z/y])[t/x]) & \text{if } x \not\doteq y\text{, and } y \text{ is free in } t\text{,} \\ & \text{and } z \text{ is a fresh variable} \end{cases}$$

**(End of definition.)**

### 2.0.1 Semantics

Now we give denotational semantics to the languages defined in the previous section. For that purpose we need some standard definitions from order theory.

#### Order-theoretic preliminaries

We start with the well-known notion of partially ordered sets.

**Definition (Partially ordered sets)** A *partially ordered set* is a pair $(D, \sqsubseteq)$, where $D$ is a set, and $\sqsubseteq$ is a binary relation which is reflexive, antisymmetric and transitive.

**(End of definition.)**

Partially ordered sets may have least elements:

**Definition (Least elements)** Let $(D, \sqsubseteq)$ be a partially ordered set. An element $d \in D$ is called a *least element* of $D$, if $d \sqsubseteq x$ holds for all $x \in D$.

**(End of definition.)**

Each partially ordered set has at most one least element. This property is an immediate consequence of the definitions.

The concept of upper bounds, and least upper bounds is also well-known from order theory:

**Definition ((Least) upper bounds)** Let $(D, \sqsubseteq)$ be a partially ordered set, and $E \subseteq D$ a subset of $D$. An element $d \in D$ is called an *upper bound* of $E$, if $x \sqsubseteq d$ holds for all $x \in E$. An element $d \in D$ is called a *least upper bound* of $E$, if $d$ is a least element of the set of all upper bounds of $E$ in $D$.

**(End of definition.)**

**Definition (Directed sets)** Let $(D, \sqsubseteq)$ be a partially ordered set, and $E \subseteq D$ a subset of $D$. $E$ is called *directed*, if $E$ is not empty, and if for each two elements $x, y \in E$ there is a $z \in E$ such that $x \sqsubseteq z$ and $y \sqsubseteq z$ hold.

**(End of definition.)**

**Definition (Complete partial orders)** A *complete partial order*, or *cpo* for short, is a partially ordered set $(D, \sqsubseteq)$ with a least element such that every directed subset of $D$ has a least upper bound.

**(End of definition.)**

**Notation** We will use the following notations:

- We write $\equiv$ for the identity relation on a set $D$, that is, for equality of the elements of set $D$.
- When the relation $\sqsubseteq$ is clear from the context, we briefly write $D$ for partially ordered sets $(D, \sqsubseteq)$.
- We write $\perp$ for the least element of a partially ordered set, if it exists.
- We write $\bigsqcup D$ for the least upper bound of a partially ordered set, if it exists.

**(End of notation.)**

**Continuous functions.** A function is called continuous, if it preserves least upper bounds. The precise definitions follow:

**Definition (Monotonicity)** Let $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$ be partially ordered sets. A function $f : D \rightarrow E$ is *monotonic*, if $f(x) \sqsubseteq_E f(y)$ holds for all $x, y \in D$ with $x \sqsubseteq_D y$.

**(End of definition.)**

**Definition (Continuity)** Let $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$ be cpo's. A function $f : D \rightarrow E$ is *continuous*, if it is monotonic, and $f(\bigsqcup M) \equiv \bigsqcup(\{f(m) | m \in M\})$ holds for every directed subset $M \subseteq D$.

**(End of definition.)**

Let $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$ be cpo's. Let $D \rightarrow E$ be the set of all continuous functions from $D$ to $E$. Let $\sqsubseteq$ be the pointwise ordering on $D \rightarrow E$ (that is, let $f \sqsubseteq g$ if and only if $f(x) \sqsubseteq_E g(x)$ holds for all $x \in D$).

Then $(D \rightarrow E, \sqsubseteq)$ again is a cpo.

For every cpo $(D \rightarrow D, \sqsubseteq)$ the continuous function $\mathsf{FIX} : (D \rightarrow D) \rightarrow D$ yields the least fixed point of functions. Formally, for every continuous function $f$ the following properties hold:

$$\mathsf{FIX}(f) \equiv f(\mathsf{FIX}(f)) \quad \text{(fixed point)}$$
$$\forall d \in D : f(d) \equiv d \Rightarrow \mathsf{FIX}(f) \sqsubseteq d \quad \text{(least fixed point)}$$

The least fixed point of a function can be represented as a least upper bound:

$$\mathsf{FIX}(f) \equiv \bigsqcup_n f^n \bot \ ,$$

where $f^n$ denotes $n$-fold iteration of $f$.

**(End "Continuous functions".)**

**Products.** Let $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$ be partially ordered sets. Let $D \times E$ be the Cartesian product of sets $D$ and $E$. Let $\sqsubseteq$ be the coordinatewise ordering on $D \times E$ (that is, let $(d, e) \sqsubseteq (d', e')$ if and only if $d \sqsubseteq_D d'$ and $e \sqsubseteq_E e'$).

Then $(D \times E, \sqsubseteq)$ again is a partially ordered set. If $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$ are cpo's, then $(D \times E, \sqsubseteq)$ is a cpo.

**(End "Products".)**

**Booleans.** The cpo $\mathbb{B}$ of boolean values consists of the three elements $\{tt, ff, \bot\}$, where $\bot$ is the least element, and $tt$ and $ff$ are incomparable elements.

**(End "Booleans".)**

**(End "Order-theoretic preliminaries".)**

**Notation** We use the symbols

$$\times, (.,.), \rightarrow, \lambda, \bot,$$

and juxtaposition for function application both in the object language (cf. section 2.0) and at the meta-level of semantics.

We write $[.,.]$ for the update of functions: if $f : M \rightarrow N$ is a function, and $m \in M, n \in N$, then the function $f[n/m] : M \rightarrow N$ is defined as follows:

$$f[n/m] \equiv \lambda y. \begin{cases} n & \text{if } y \equiv m \\ f(y) & \text{otherwise} \end{cases}$$

**(End of notation.)**

Now we are ready to define a denotational semantics for our language. We will give a non-strict (also known as call-by-name or lazy) denotational semantics. The definition we will give is standard in denotational semantics (for instance cf. [32]).

We begin with the semantics of types. For sorts we only assume that they are interpreted as cpo's. Product types are interpreted as non-strict products. Function types are interpreted as sets of continuous functions.

**Definition (Semantics of types)** Let $D$ be a function, which associates a cpo with each sort $s \in S$. Then $D$ is extended to types by the following structural induction:

- $D(\tau \times \sigma) = D(\tau) \times D(\sigma)$
- $D(\tau \rightarrow \sigma) = D(\tau) \rightarrow D(\sigma)$

**(End of definition.)**

**Definition (Continuous algebras)** Let $\Sigma$ be a signature. A *continuous $\Sigma$-algebra* $C$ consists of a function $D$, which associates a cpo $s^C$ with each sort $s$ of the signature, and of an element $f^C \in D(\tau)$ for each function symbol $f$ of type $\tau$ of the signature, where sort $Bool$ and the function symbols $\perp$, $tt$ and $ff$ are interpreted in the standard way. For $D(\tau)$ we write $\tau^C$.

**(End of definition.)**

Let $C$ be a continuous algebra. Then $Env$ is the set of all functions from variables $X$ into the cpo's of $C$ such that each variable $x$ of type $\tau$

is associated with a data value of $\tau^C$. The functions in $Env$ are called *environments*.

**Definition (Semantics of terms)** Let $C$ be a continuous algebra, and $Env$ the set of environments. The function

$$[\![.]\!]$$

associates a value $[\![t]\!]\eta$ of $\tau^C$ with each term $t$ of type $\tau$, and each environment $\eta$. Function $[\![.]\!]$ is defined as follows:

$$\begin{aligned}
&[\![f]\!]\eta \equiv f^C \\
&[\![x]\!]\eta \equiv \eta\ x \\
&[\![\mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2\ \mathbf{fi}]\!]\eta \equiv \begin{cases} [\![t_1]\!]\eta & \text{if } [\![t_0]\!]\eta \equiv tt \\ [\![t_2]\!]\eta & \text{if } [\![t_0]\!]\eta \equiv ff \\ \bot & \text{if } [\![t_0]\!]\eta \equiv \bot \end{cases} \\
&[\![(t_0, t_1)]\!]\eta \equiv ([\![t_0]\!]\eta, [\![t_1]\!]\eta) \\
&[\![fst\ t]\!]\eta \equiv d_0\ , \quad \text{if } [\![t]\!]\eta \equiv (d_0, d_1) \\
&[\![snd\ t]\!]\eta \equiv d_1\ , \quad \text{if } [\![t]\!]\eta \equiv (d_0, d_1) \\
&[\![\lambda x.\ t]\!]\eta \equiv \lambda d.\ [\![t]\!](\eta[d/x]) \\
&[\![t\ s]\!]\eta \equiv ([\![t]\!]\eta)([\![s]\!]\eta) \\
&[\![\mathbf{rec}\ x.\ t]\!]\eta \equiv \mathsf{FIX}(\lambda d.\ [\![t]\!](\eta[d/x]))
\end{aligned}$$

**(End of definition.)**

(The reader might wonder why we have not used lifted function spaces. We have given the above semantics in order to keep things simple, which are not of primary interest in this work. The choice does not influence the sequel.)

Note that mutual recursion can be expressed in the language by **rec** $p.\ t$, where $p$ is of product type. So, if we write $(x, y)$ for $p$, the recursion term can written as **rec** $(x, y).\ (t_0, t_1)$, where $x$ and $y$ may occur both in $t_0$ and $t_1$. Since $p$ can be of any product type, this notation is more general than mutually recursive function declarations. This is what we meant in example 10 of chapter 1.

**Lemma (Substitution lemma for terms)** Substitution in terms is compatible with update of environments: Let $t$ and $u$ be terms, $x$ a variable, and $\eta$ an environment. The following equivalence holds:

$$[\![t[u/x]]\!]\,\eta \equiv [\![t]\!]\,\eta[[\![u]\!]\,\eta/x]$$

**Proof** By structural induction on $t$.

**(End of proof.)**

### 2.0.2 Discussion

According to our requirements of chapter 1 we have chosen the recursion operator so that it can be written into terms. No new syntactic category, such as function declarations, is introduced. Thus the syntax becomes simple and easily manipulable. We have already discussed the **rec**-notation as an example in chapter 1.

Moreover, we have chosen the operator **rec** instead of **fix** for the reasons, which have been discussed in example 4 of chapter 1: Since in practice the least-fixed-point-operator is in most cases applied to a $\lambda$-abstraction, it is worth to introduce the abbreviation **rec** for that combination.

## 2.1 Formulae

Next we define formulae on the class of terms that have been defined in the previous section. Again we get a whole class of formula languages since we do not presuppose a concrete signature, and in addition allow user-defined predicate symbols.

### 2.1.0 Syntax

Let $P$ be a set of symbols, called *predicate symbols*, such that a type is associated with each predicate symbol $p \in P$.

**Definition (Atomic formulae)** If $t$ and $u$ are terms of $T$, which have the same type $\tau$, and $p$ is a predicate symbol of $P$ with associated type $\tau$, then

- $t \sqsubseteq u$ ,

- $t \equiv u$ , and
- $p(t)$

are *atomic formulae.*

**(End of definition.)**

The symbols $\sqsubseteq$ and $\equiv$ are built-in predicate symbols that denote inequality and equality respectively. The predicate symbols of $P$ may be specified by the user.

**Notation** We use the symbols $\sqsubseteq$ and $\equiv$ both on the syntactic level and on the semantic level (as in section 2.0.1).

**(End of notation.)**

**Definition (Formulae)** The set of *formulae* is inductively defined as follows:

- Every atomic formula is a formula.
- If $A$ and $B$ are formulae, and $x \in X$ is a variable, then

  $$\neg A, A \Rightarrow B, A \Leftrightarrow B, A \vee B, A \wedge B, \forall x\ A \text{ and } \exists x\ A$$

  are formulae.

**(End of definition.)**

**Notation** In order to allow omission of brackets, we give binding powers to the logical connectives. We list them from highest binding power to lowest:

$\neg$

$\vee, \wedge$

$\Rightarrow, \Leftrightarrow$

$\forall, \exists$

Connectives of the same line are given equal binding powers.

**(End of notation.)**

**Definition (Substitution in formulae)** The function .[./.] substitutes terms for variables in formulae. It is defined by structural induction on formulae as usual:

$$(s \sqsubseteq u)[t/x] \doteq (s[t/x]) \sqsubseteq (u[t/x])$$

$$(s \equiv u)[t/x] \doteq (s[t/x]) \equiv (u[t/x])$$

$$p(u)[t/x] \doteq p(u[t/x])$$

$$(\neg A)[t/x] \doteq \neg(A[t/x])$$

$$(A\ op\ B)[t/x] \doteq (A[t/x]) op (B[t/x])$$

where *op* ranges over $\Rightarrow$, $\Leftrightarrow$, $\vee$ and $\wedge$

$$(op\, y\ A)[t/x]$$

$$\begin{cases} op\, y\ A & \text{if } x \doteq y \\ op\, y(A[t/x]) & \text{if } x \not\doteq y\text{, and } y \text{ is not free in } t \\ op\, z((A[z/y])[t/x]) & \text{if } x \not\doteq y\text{, and } y \text{ is free in } t\text{,} \\ & \text{and } z \text{ is a fresh variable} \end{cases}$$

where *op* ranges over the binding operators $\forall$ and $\exists$

**(End of definition.)**

### 2.1.1 Semantics

**Definition (Truth values)** We take $\{true, false\}$ as the set of *truth values*. We presuppose the usual functions $\neg$ (negation), $\Rightarrow$ (implication), $\Leftrightarrow$ (bi-implication or equivalence), $\vee$ (disjunction), and $\wedge$ (conjunction) on the truth values.

**(End of definition.)**

**Notation** We use the following notations:

- The identity relation on the truth values is denoted by $=$.
- We use the symbols $\neg$, $\Rightarrow$, $\Leftrightarrow$, $\vee$ and $\wedge$ both at the syntactic level and at the semantic level.
- We use the symbols $\forall$ and $\exists$ both at the object level and at the meta-level.

**(End of notation.)**

**Definition (Continuous interpretations)** A *continuous interpretation* $I$ consists of a continuous algebra $C$, and of a function $p^I : \tau^C \rightarrow \{true, false\}$ for each predicate symbol $p \in P$ of type $\tau$.

**(End of definition.)**

**Definition (Interpretation of formulae)** For every continuous interpretation $I$, and every environment $\eta$ a truth value is assigned to each formula in the following way by the function $[\![.]\!]$, which is defined recursively on the structure of formulae:

$$\begin{aligned}
&[\![t \sqsubseteq u]\!]\eta = ([\![t]\!]\eta \sqsubseteq [\![u]\!]\eta)\\
&[\![t \equiv u]\!]\eta = ([\![t]\!]\eta \equiv [\![u]\!]\eta)\\
&[\![p(t)]\!]\eta = p^I([\![t]\!]\eta)\\
&[\![\neg A]\!]\eta = \neg([\![A]\!]\eta)\\
&[\![A \Rightarrow B]\!]\eta = ([\![A]\!]\eta \Rightarrow [\![B]\!]\eta)\\
&[\![A \Leftrightarrow B]\!]\eta = ([\![A]\!]\eta \Leftrightarrow [\![B]\!]\eta)\\
&[\![A \vee B]\!]\eta = ([\![A]\!]\eta \vee [\![B]\!]\eta)\\
&[\![A \wedge B]\!]\eta = ([\![A]\!]\eta \wedge [\![B]\!]\eta)\\
&[\![\forall x\ A]\!]\eta = \forall d : [\![A]\!](\eta[d/x])\\
&[\![\exists x\ A]\!]\eta = \exists d : [\![A]\!](\eta[d/x])
\end{aligned}$$

**(End of definition.)**

**Definition (Validity)** A formula $A$ is called *valid* under a continuous interpretation $I$, if for all environments $\eta$ of $I$ the following holds:

$$[\![A]\!]\eta = true$$

**(End of definition.)**

**Lemma (Substitution lemma for formulae)** Substitution in formulae is compatible with update of environments: Let $A$ be a formula,

$t$ a term, $x$ a variable, and $\eta$ an environment. The following equivalence holds:

$$[\![A[t/x]]\!]\,\eta \equiv [\![A]\!]\,\eta[[\![t]\!]\,\eta/x]$$

**Proof** By structural induction on $A$.

**(End of proof.)**

## 2.2 Rules

Now we give the rules relating to recursion. They are rules of a predicate calculus. But as already mentioned, we do not list the entire calculus; we confine ourselves to those rules that are important in our development method.

### 2.2.0 Syntactic admissibility

We have seen that least fixed points can be approximated by function iterations. This leads to the idea to prove formulae for all elements of the approximation, and to conclude that the formula holds for the least fixed point. Unfortunately, as is well-known, this reasoning is not possible for all formulae. Therefore we first characterize the formulae for which this pattern of reasoning is allowed.

**Definition (Admissibility)** A formula $A$ is called *admissible* in a variable $x \in X$ of type $\tau$, if for all continuous interpretations $I$, for all environments $\eta$, and for all directed sets $D \subseteq \tau^I$ the following holds:

If $\forall d \in D : [\![A]\!](\eta[d/x]) = true$, then $[\![A]\!](\eta[\bigsqcup D/x]) = true$.

**(End of definition.)**

This semantic notion of admissibility is undecidable. Since we have required decidable applicability conditions for rules in chapter 1, we must define a decidable property of formulae, which implies admissibility. In [53] such a characterization is given, which is sufficient for most practical applications:

**Definition (Syntactic admissibility)** Let $x \in X$ be a variable. The set of formulae that are *syntactically admissible* in $x$ is inductively defined as follows:

- If $x$ is not free in formula $A$, then $A$ is syntactically admissible in $x$.
- If $t$ and $u$ are terms of the same type, then $t \sqsubseteq u$ and $t \equiv u$ are syntactically admissible in $x$.
- If $t$ and $u$ are terms of the same type, and $x$ is not free in $u$, then $\neg(t \sqsubseteq u)$ is syntactically admissible in $x$.
- If formula $A$ is syntactically admissible in $x$, and $y$ is any variable, then $\forall y\, A$ is syntactically admissible in $x$.
- If formulae $A$ and $B$ are syntactically admissible in $x$, then so are $A \vee B$ and $A \wedge B$.
- If formulae $\neg A$ and $B$ are syntactically admissible in $x$, then so is $A \Rightarrow B$.
- If $t$ is a term, then $\neg(t \equiv \bot)$ is syntactically admissible in $x$.

**(End of definition.)**

Since most of the proof is omitted in [53], we now show that syntactic admissibility indeed implies admissibility.

**Proposition (Syntactic admissibility)** Syntactic admissibility implies admissibility.

**Proof** Let formula $A$ be syntactically admissible in variable $x$ of type $\tau$. Let $I$ be a continuous interpretation, $\eta$ an environment, and $D \sqsubseteq \tau^I$ a directed set.

We prove the admissibility of $A$ by induction on the generation rules for syntactically admissible formulae:

- Let $x$ be not free in $A$. Then the admissibility of $A$ follows from the coincidence lemma for formulae.

- If $A$ is of the form $t \sqsubseteq u$, then admissibility follows from monotonicity and continuity of all functions of the object language. If $A$ is of the form $t \equiv u$, then admissibility follows from the admissibility of $t \sqsubseteq u$ and $u \sqsubseteq t$.

- If $A$ is of the form $\neg(t \sqsubseteq u)$, and $x$ is not free in $u$, then admissibility follows from the coincidence lemma for terms, and from monotonicity of all functions of the object language.

- Let $A$ be of the form $\forall y\ B$, where $B$ is syntactically admissible in $x$. If $y \doteq x$, then the admissibility of $A$ follows from the first case above, because $x$ is not free in $\forall x\ B$.

  If $y \not\doteq x$, then the following holds:

$$\begin{array}{ll} & \forall d \in D : [\![\forall y\ B]\!](\eta[d/x]) \\ \Leftrightarrow & \quad \{\text{semantics of formulae; update of environments}\} \\ & \forall d \in D, \forall e : [\![B]\!]((\eta[e/y])[d/x]) \\ \Rightarrow & \quad \{B \text{ is syntactically admissible in } x; \\ & \quad \text{induction hypothesis}\} \\ & \forall e : [\![B]\!]((\eta[e/y])[\sqcup D/x]) \\ \Leftrightarrow & \quad \{\text{semantics of formulae; update of environments}\} \\ & [\![\forall y\ B]\!](\eta[\sqcup D/x]) \end{array}$$

- Let $A$ be of the form $B \vee C$, where $B$ and $C$ are syntactically admissible in $x$.

$$\begin{array}{ll} & \forall d \in D : [\![B \vee C]\!](\eta[d/x]) \\ \Leftrightarrow & \quad \{\text{semantics of formulae}\} \\ & \forall d \in D : ([\![B]\!](\eta[d/x]) \vee [\![C]\!](\eta[d/x])) \\ \Rightarrow & \quad \{\text{let } D_B \subseteq D \text{ be the set of all } d \text{ with} \\ & \quad [\![B]\!](\eta[d/x]) = true\text{; let } D_C \text{ be defined} \\ & \quad \text{analogously for } C\text{; distinguish the cases } \sqcup D \in D \\ & \quad \text{and } \sqcup D \notin D\text{; for the latter distinguish further:} \\ & \quad D_B \text{ or } D_C \text{ is finite or both are infinite; for the} \\ & \quad \text{latter: both } D_B \text{ and } D_C \text{ are directed or not} \end{array}$$

both;

induction hypothesis; semantics of formulae}

$[\![B \vee C]\!](\eta[\sqcup D/x])$

Let $A$ be of the form $B \wedge C$, where $B$ and $C$ are syntactically admissible in $x$.

$$\begin{array}{ll}
 & \forall d : [\![B \wedge C]\!](\eta[d/x]) \\
\Leftrightarrow & \quad \{\text{semantics of formulae}\} \\
 & \forall d : ([\![B]\!](\eta[d/x]) \wedge [\![C]\!](\eta[d/x])) \\
\Rightarrow & \quad \{\text{induction hypothesis on } B \text{ and } C\} \\
 & [\![B]\!](\eta[\sqcup D/x]) \wedge [\![C]\!](\eta[\sqcup D/x]) \\
\Leftrightarrow & \quad \{\text{semantics of formulae}\} \\
 & [\![B \wedge C]\!](\eta[\sqcup D/x])
\end{array}$$

- If $A$ is of the form $B \Rightarrow C$, where $\neg B$ and $C$ are syntactically admissible in $x$, then $A$ is admissible, because $B \Rightarrow C$ is equivalent to $\neg B \vee C$, and because of the preceeding case.

- The formula $\neg(t \equiv \perp)$ is equivalent to $\neg(t \sqsubseteq \perp)$, and thus admissible according to the third case above.

**(End of proof.)**

### 2.2.1 Fixed point induction rule

The fixed point induction rule states that the validity of a formula for a least fixed point may be inferred from its validity of the approximations, provided the formula is syntactically admissible:

$$\frac{\begin{array}{c} A[\perp/x] \\ \forall x(A \Rightarrow A[t/x]) \end{array}}{A[\mathbf{rec}\ x.\ t/x]} \qquad \text{where } A \text{ is syntactically admissible in } x$$

We prove the soundness of the fixed point induction rule.

**Proof** Let $A$ be syntactically admissible in $x$, and $\eta$ an arbitrary environment in a continuous interpretation.

$$
\begin{array}{ll}
 & \llbracket A[\mathbf{rec}\ x.\ t/x] \rrbracket\ \eta \\
\Leftrightarrow & \quad \{\text{substitution lemma for formulae}\} \\
 & \llbracket A \rrbracket\ \eta[\llbracket \mathbf{rec}\ x.\ t \rrbracket\ \eta/x] \\
\Leftrightarrow & \quad \{\text{semantics of terms}\} \\
 & \llbracket A \rrbracket\ \eta[\mathsf{FIX}(\lambda d.\ \llbracket t \rrbracket\ \eta[d/x])/x] \\
\Leftrightarrow & \quad \{\text{approximation of least fixed point}\} \\
 & \llbracket A \rrbracket\ \eta[\bigsqcup_{n \in \mathbb{N}} d_n/x] \\
 & \quad \text{with } \ d_0 \equiv \bot \\
 & \qquad\qquad d_{n+1} \equiv \llbracket t \rrbracket\ \eta[d_n/x] \\
\Leftarrow & \quad \{\text{admissibility of } A \text{ in } x\} \\
 & \forall n \in \mathbb{N} : \llbracket A \rrbracket \eta[d_n/x] \\
\Leftarrow & \quad \{\text{mathematical induction}\} \\
 & \llbracket A \rrbracket\ \eta[d_0/x] \wedge \\
 & (\forall n \in \mathbb{N} : \llbracket A \rrbracket\ \eta[d_n/x] \Rightarrow \llbracket A \rrbracket\ \eta[d_{n+1}/x]) \\
\Leftrightarrow & \quad \{\text{definition of } d_0 \text{ and } d_{n+1}\} \\
 & \llbracket A \rrbracket\ \eta[\bot/x] \wedge \\
 & (\forall n \in \mathbb{N} : \llbracket A \rrbracket\ \eta[d_n/x] \Rightarrow \llbracket A \rrbracket\ \eta[\llbracket t \rrbracket\ \eta[d_n/x]/x]) \\
\Leftrightarrow & \quad \{\text{semantics of terms; property of update}\} \\
 & \llbracket A \rrbracket\ \eta[\bot\ \eta/x] \wedge \\
 & (\forall n \in \mathbb{N} : \llbracket A \rrbracket\ \eta[d_n/x] \Rightarrow \llbracket A \rrbracket\ (\eta[d_n/x])[\llbracket t \rrbracket\ \eta[d_n/x]/x]) \\
\Leftrightarrow & \quad \{\text{substitution lemma for formulae}\} \\
 & \llbracket A[\bot/x] \rrbracket\ \eta \wedge \\
 & (\forall n \in \mathbb{N} : \llbracket A \rrbracket\ \eta[d_n/x] \Rightarrow \llbracket A[t/x] \rrbracket\ \eta[d_n/x]) \\
\Leftrightarrow & \quad \{\text{semantics of formulae}\} \\
 & \llbracket A[\bot/x] \rrbracket\ \eta \wedge \\
 & (\forall n \in \mathbb{N} : \llbracket A \Rightarrow A[t/x] \rrbracket\ \eta[d_n/x]) \\
\Leftarrow & \quad \{\text{universal quantification on more elements}\} \\
 & \llbracket A[\bot/x] \rrbracket\ \eta \wedge \\
 & (\forall d : \llbracket A \Rightarrow A[t/x] \rrbracket\ \eta[d/x]) \\
\Leftrightarrow & \quad \{\text{semantics of formulae}\}
\end{array}
$$

$$
\begin{aligned}
&[\![A[\bot/x]]\!]\,\eta \wedge \\
&[\![\forall x(A \Rightarrow A[t/x])]\!]\,\eta
\end{aligned}
$$

**(End of proof.)**

In the inductive step

$$\forall x(A \Rightarrow A[t/x])$$

we call the induction hypothesis $A$ *fully applicable* to the conclusion $A[t/x]$, if its application removes *all* occurrences of induction variable $x$.

### 2.2.2 Fixed point rule

The fixed point rule is our second rule relating to recursion.

The fixed point rule states that a recursively defined object is a fixed point of the defining function:

$$\frac{}{\mathbf{rec}\ x.\ t \equiv t[\mathbf{rec}\ x.\ t/x]}$$

We prove the soundness of the fixed point rule.

**Proof** Let an arbitrary continuous interpretation be given, and $\eta$ be an arbitrary environment in that interpretation.

$$
\begin{aligned}
& [\![t[\mathbf{rec}\ x.\ t/x]]\!]\,\eta \\
= &\quad \{\text{substitution lemma for terms}\} \\
& [\![t]\!](\eta[[\![\mathbf{rec}\ x.\ t]\!]\eta/x]) \\
= &\quad \{\text{semantics of terms}\} \\
& [\![t]\!](\eta[\mathsf{FIX}(\lambda d.\ [\![t]\!](\eta[d/x]))/x]) \\
= &\quad \{\text{fixed point approximation}\} \\
& [\![t]\!](\eta[\bigsqcup_n(\Theta^n\bot)/x])\ , \quad \text{where } \Theta \equiv \lambda d.\ [\![t]\!](\eta[d/x]) \\
= &\quad \{\text{continuity of semantics}\} \\
& \bigsqcup_n [\![t]\!](\eta[\Theta^n\bot/x])
\end{aligned}
$$

$$\begin{array}{ll}
= & \quad \{\text{definition of } \Theta\} \\
 & \bigsqcup_n \Theta^{n+1} \bot \\
= & \quad \{\Theta^0 \equiv \bot \text{ least element}\} \\
 & \bigsqcup_n \Theta^n \bot \\
= & \quad \{\text{fixed point approximation; definition of } \Theta\} \\
 & \mathsf{FIX}(\lambda d.\ [\![t]\!](\eta[d/x])) \\
= & \quad \{\text{semantics of terms}\} \\
 & [\![\mathbf{rec}\ x.\ t]\!]\eta
\end{array}$$

**(End of proof.)**

### 2.2.3 Generalization rule

The generalization rule is well-known from predicate calculus. We repeat it here, because it plays an important rôle in our development method.

The generalization rule states that all variables (free and bound ones) of a formula may be universally quantified:

$$\frac{A}{\forall x\ A}$$

We prove the soundness of the generalization rule.

**Proof** Let an arbitrary continuous interpretation be given.

$$\begin{array}{ll}
 & \forall \eta \in Env : [\![A]\!]\eta \\
= & \quad \{\text{environments}\} \\
 & \forall \eta \in Env : \forall d : [\![A]\!](\eta[d/x]) \\
= & \quad \{\text{semantics of formulae}\} \\
 & \forall \eta \in Env : [\![\forall x\ A]\!]\eta = true
\end{array}$$

**(End of proof.)**

### 2.2.4 Further rules

Fixed point induction, and the fixed point rule are the only rules concerning recursion in our calculus. In addition, we use axioms and rules of predicate logic. But we do not list them. It is well-known that minimal predicate calculi, which are used in mathematical logic as objects of study, are inappropriate to practical proof. Hence we should have to list many more rules. Since the laws of predicate logic are common knowledge, we leave them out.

The calculus can also be enriched by other rules, in particular by structural induction rules. We leave them out, too, because they do not play a special rôle in our method.

It would go beyond the scope of this work to include a whole program development calculus. Hence we have decided to give up full formalization in order not to get bogged down into detail.

### 2.2.5 Discussion

Now we discuss why we have included the above rules, and omitted other rules for recursion, which are known from the literature. In this section we confine ourselves to reasons concerning the criteria of chapter 1. Methodological reasons that led to exclusion of rules will be discussed in chapter 4.

A pair of rules, which is well-known from the literature, is the unfold-rule together with the fold-rule ([10]). The unfold-rule substitutes inside the body of a recursive definition the recursive definition itself for the variable bound by the recursion operator. The fold-rule does the converse.

We can express the unfold-rule in our notation as follows:

$$\mathbf{rec}\ x.\ t[x/y] \equiv \mathbf{rec}\ x.\ (t[t[x/y]/y])$$

The fold-rule can be expressed as follows:

$$\frac{\mathbf{rec}\ x.\ t[u/y] \equiv \mathbf{rec}\ x.\ u}{\mathbf{rec}\ x.\ t[u/y] \sqsupseteq \mathbf{rec}\ x.\ t[x/y]}$$

It is well-known that $\sqsupseteq$ in the conclusion of the fold-rule cannot be replaced by $\equiv$. Application of the fold-rule may lead to smaller values.

Speaking operationally, the fold-rule preserves only partial correctness, but termination can get lost.

Therefore the fold-rule has extensively been studied in the literature, in order to find additional conditions that allow equivalence in the conclusion. Those investigations led to a great amount rules based on the unfold/fold-technique. But those extended rules contradict the criteria found in chapter 1 in two respects:

Firstly, those rules tend to be rather complicated. We have seen such a rule in example 5 in chapter 1. Because of our requirement for simple rules, we did not include such modified unfold/fold-rules into our calculus.

Secondly, the transformation calculi, in which those elaborate rules are used, do not meet the requirement for local proof steps: typically, the applicability of a rule depends on the structure of the transformation history. We have mentioned such a non-local transformation calculus in example 17 of chapter 1. The loss of locality was another reason for which we excluded such rules from the calculus.

Although fulfilling the criteria of chapter 1, the pure unfold-rule and fold-rule given above do not belong to our calculus. They have been excluded for methodological reasons, which we will discuss in chapter 4.

## 2.3 Notations for the examples

In this section we introduce notations that we need in the examples of chapter 3.

**Lists.** We introduce

$$List$$

as the sort of finite lists (also called sequences). Since we need lists on different element sorts, we use the notation $List(\tau)$ for lists on elements of type $\tau$.

Sort *List* has three strict constructors:

$$\varepsilon : List$$
$$\langle . \rangle : \tau \to List$$
$$. \circ . : List, List \to List$$

By $\varepsilon$ we denote the empty list; the constructor $\langle . \rangle$ generates lists with one element; list concatenation is denoted by $. \circ .$.

In addition, the functions

$$hd : List \to \tau$$
$$tl : List \to List$$

are given.

The functions *hd* and *tl* stand for selection of the head and tail of a list, respectively. They are assumed to obey the usual laws.

These notations are understood to enrich the set $S$ of sorts, and the set $F$ of function symbols.

**(End "Lists".)**

**Pattern matching.** For convenience, we use the notation of pattern matching in our examples. Since it is known from many programming languages, we do not define it precisely. Instead we give an example:

A function $f$ on lists can be defined by pattern matching:

$$f(\varepsilon) \equiv t$$
$$f(\langle x \rangle \circ l) \equiv u$$

This abbreviates the following definition:

$$f \equiv \lambda s.\ \textbf{if } s = \varepsilon \textbf{ then } t \textbf{ else } u[hd\ s/x, tl\ s/l] \textbf{ fi}$$

**(End "Pattern matching".)**

# Chapter 3

# Examples of proofs about recursion

In this chapter we develop recursive programs from specifications for a number of examples. Because of their length, we cannot present the examples in full detail. But we explain the main steps of the developments, and how they relate to our method.

## 3.0 List reversal

In order to acquaint the reader with our development method for recursion, we start with the small example of list reversal.

The task is to develop a tail-recursive function $F$ from a recursively defined reverse function $rev$. In the literature (e.g. [43], [3]) the solution $F$ is usually presented, and then proved equal to $rev$. But it is not said how $F$ can be found systematically, and how the lemma used in the proof can be found.

We use the list notation introduced in section 2.3 in this example.

Assume the following recursive definition of the reverse function $rev$ to be given:

$$rev \equiv \textbf{rec}\ r.\ \lambda s.\ \textbf{if}\ \ s = \varepsilon\ \ \textbf{then}\ \ \varepsilon\ \ \textbf{else}\ \ r(tl\ s) \circ \langle hd\ s\rangle\ \textbf{fi}$$

Let $\tau_{rev}$ denote the functional associated with this recursive definition.

Our task is to develop a tail-recursive function $F$ with the property

$$\textbf{(Rev)} \qquad rev \sqsubseteq F \ .$$

Since the left hand side of specification **(Rev)** is recursively defined, we will refine this specification by fixed point induction. Therefore we start with an analysis, whether fixed point induction on $rev$ is possible and promising.

**Fixed point analysis.** Let us analyse fixed point induction on $rev$ in specification **(Rev)**. The base case holds trivially. The inductive step is

$$\forall r : r \sqsubseteq F \Rightarrow (\tau_{rev}\ r) \sqsubseteq F \ .$$

By definition of $\tau_{rev}$, the inductive step is equivalent to

$$\begin{array}{l} \forall r : \\ r \sqsubseteq F \Rightarrow \\ \lambda s.\ \textbf{if}\ \ s = \varepsilon\ \ \textbf{then}\ \ \varepsilon\ \ \textbf{else}\ \ r(tl\ s) \circ \langle hd\ s \rangle\ \textbf{fi} \sqsubseteq F \ . \end{array}$$

The induction hypothesis $r \sqsubseteq F$ is applicable to $r(tls)$ in the conclusion of the inductive step. But since the function $\circ$ is applied to $r(tl\ s)$, and since we are interested in a tail-recursive definition of $F$, nothing would be gained, if we applied the induction hypothesis.

**(End of fixed point analysis.)**

Thus we have found a fixed point induction in which the induction hypothesis is fully applicable, but application of the induction hypothesis would lead to a term of an inappropriate shape. Therefore we introduce an auxiliary function.

**Auxiliary function.** Now we abstract the term that prevented us from applying the induction hypothesis to an auxiliary function $h$:

$$h \equiv \lambda r.\ \lambda s, t.\ (r\ s) \circ t$$

The first argument abstracts the induction variable.

From now on we are looking for a tail-recursive function $G$ that fulfils the following specification:

$$\textbf{(Aux)} \qquad h\, rev \sqsubseteq G$$

How can function $F$ be defined in terms of $G$ so that the original specification **(Rev)** is fulfilled? We calculate:

$$\begin{array}{ll} & rev \sqsubseteq F \\ \Leftrightarrow & \quad \{\text{definition of } h\} \\ & \lambda s.\ h\, rev(s, \varepsilon) \sqsubseteq F \\ \Leftarrow & \quad \{\text{specification } \textbf{(Aux)} \text{ of } G\} \\ & \lambda s.\ G(s, \varepsilon) \sqsubseteq F \end{array}$$

The last inequation can immediately be fulfilled by defining $F$ as follows:

$$\textbf{(F)} \qquad F \equiv \lambda s.\ G(s, \varepsilon)$$

**(End of auxiliary function.)**

Now we recursively apply our development method in order to derive a function $G$. The start is fixed point analysis.

**Fixed point analysis, and fixed point induction.** We analyse fixed point induction on $rev$ in specification **(Aux)**.

The base case is trivially fulfilled, that is, it imposes no restriction on $G$:

$$\begin{array}{ll} & h \perp \sqsubseteq G \\ \Leftrightarrow & \quad \{\text{definition of } h\text{; strictness of } \circ\} \\ & \perp \sqsubseteq G \\ \Leftrightarrow & \quad \{\perp \text{ least element}\} \\ & true \end{array}$$

Now we turn to the inductive step. Let $r$ be such that the induction hypothesis $h\, r \sqsubseteq G$ holds.

$$h(\tau_{rev}\, r) \sqsubseteq G$$

$$
\begin{array}{rl}
\Leftrightarrow & \{\text{definition of } h\} \\
& \lambda s,t.\ ((\tau_{rev}\, r)\ s) \circ t \sqsubseteq G \\
\Leftrightarrow & \{\text{definition of } \tau_{rev}\} \\
& \lambda s,t.\ (\textbf{if}\ \ s=\varepsilon\ \ \textbf{then}\ \ \varepsilon\ \ \textbf{else}\ \ r(tl\ s)\circ\langle hd\ s\rangle\ \textbf{fi})\circ t \sqsubseteq G \\
\Leftrightarrow & \{\text{strictness of } \circ\} \\
& \lambda s,t.\ \textbf{if}\ \ s=\varepsilon\ \ \textbf{then}\ \ t\ \ \textbf{else}\ \ r(tl\ s)\circ\langle hd\ s\rangle\circ t\ \textbf{fi} \sqsubseteq G \\
\Leftrightarrow & \{\text{definition of } h\} \\
& \lambda s,t.\ \textbf{if}\ \ s=\varepsilon\ \ \textbf{then}\ \ t\ \ \textbf{else}\ \ h\, r(tl\ s,\langle hd\ s\rangle\circ t)\ \textbf{fi} \sqsubseteq G \\
\Leftarrow & \{\text{induction hypothesis}\} \\
\textbf{(I)} & \lambda s,t.\ \textbf{if}\ \ s=\varepsilon\ \ \textbf{then}\ \ t\ \ \textbf{else}\ \ G(tl\ s,\langle hd\ s\rangle\circ t)\ \textbf{fi} \sqsubseteq G
\end{array}
$$

This time application of the induction hypothesis was reasonable, because it put $G$ at a tail-recursive position.

**(End of fixed point analysis, and fixed point induction.)**

In order to fulfil specification **(I)**, we turn it into a recursive definition of $G$:

$$\textbf{(G)}\quad G \equiv \textbf{rec}\ g.\ \lambda s,t.\ \textbf{if}\ \ s=\varepsilon\ \ \textbf{then}\ \ t\ \ \textbf{else}\ \ g(tl\ s,\langle hd\ s\rangle\circ t)\ \textbf{fi}$$

By the fixed point rule it immediately follows that **(G)** implies **(I)**.

Specifications **(F)** and **(G)** together form a program for $F$.

## 3.1 Nested recursion

Now we come to a more difficult example. It is taken from [3]. There a function $F$ is defined with nested recursion, and a function, say $G$, with tail-recursion. It is stated that both functions are equivalent, but unfortunately, no proof is given. The example turns out to be difficult enough that ad hoc proofs are likely to fail.

Therefore, let us treat this example by our development method. We will not only prove the equivalence of $F$ and $G$, but develop $G$ from $F$.

We are given the following definition of $F$ with nested recursion:

$$F \equiv \textbf{rec } f.\ \lambda x.\ \textbf{if } \ p\,x \ \textbf{ then } \ g\,x \ \textbf{ else } \ f(f(h\,x))\ \textbf{fi}\ ,$$

where $p$, $g$ and $h$ are function symbols that are not specified further. Let $\tau_F$ denote the functional associated with the recursive definition of $F$.

Our task is to develop a function $G$ that is tail-recursive, and fulfils the specification

$$F \equiv G\ .$$

We develop $G$ from the inequation

$$\textbf{(GE)} \qquad F \sqsubseteq G\ ,$$

and prove the remaining inequation

$$\textbf{(LE)} \qquad G \sqsubseteq F$$

thereafter.

**Fixed point analysis.** We analyse fixed point induction on $F$ in the specification **(GE)**. The inductive step is

$$\forall f : f \sqsubseteq G \Rightarrow \lambda x.\ \textbf{if } \ p\,x \ \textbf{ then } \ g\,x \ \textbf{ else } \ f(f(h\,x))\ \textbf{fi} \ \sqsubseteq G\ .$$

The induction hypothesis is fully applicable in the inductive step, but it does not lead to a tail-recursive form, because of the nested occurrence $f(f(h\,x))$ of the induction variable $f$. Therefore we do not apply the induction hypothesis.

**(End of fixed point analysis.)**

Since fixed point induction would have lead to a term of an inappropriate shape, we define an auxiliary function.

**Auxiliary function.** We need an auxiliary function that abstracts the induction variable, and the term that prevented us from applying the induction hypothesis. Therefore we define a function *iter*, which iterates the induction variable $f$:

$$iter \equiv \textbf{rec}\ it.\ \lambda f.\ \lambda x, i.\ \textbf{if}\ \ i = 0\ \ \textbf{then}\ \ x\ \ \textbf{else}\ \ it\ f(f\ x, i-1)\ \textbf{fi}\ .$$

Now we must develop a tail-recursive function $Q$ with the property

$$\textbf{(Aux)}\qquad iter\ F \equiv Q\ .$$

We need a definition of function $G$, which we had to develop originally, in terms of function $Q$ so that specification **(GE)** is fulfilled. Therefore we calculate:

$$\begin{array}{ll} & F \sqsubseteq G \\ \Leftrightarrow & \quad \{\text{definition of } iter\} \\ & \lambda x.\ iter\ F(x,1) \sqsubseteq G \\ \Leftrightarrow & \quad \{\text{specification } \textbf{(Aux)} \text{ of } Q\} \\ & \lambda x.\ Q(x,1) \sqsubseteq G \end{array}$$

The last inequation can immediately be satisfied by defining

$$\textbf{(G)}\qquad G \equiv \lambda x.\ Q(x,1)\ .$$

By requiring $iter\ F \equiv Q$ in **(Aux)** instead of $iter\ F \sqsubseteq Q$, we have already fulfilled specification **(LE)**.

**(End of auxiliary function.)**

Now we recursively apply our development method to the new specification **(Aux)**. We again develop $Q$ from the inequation

$$\textbf{(AuxGE)}\qquad iter\ F \sqsubseteq Q\ ,$$

and prove the remaining inequation

$$\textbf{(AuxLE)}\qquad Q \sqsubseteq iter\ F\ ,$$

thereafter.

**Fixed point analysis, and fixed point induction.** There are two possibilities of fixed point induction in **(Aux)**: $F$ and *iter*.

Fixed point induction on $F$ fails, because the induction hypothesis is not applicable in the inductive step.

Now we analyse fixed point induction on $iter$. Let $\tau_{iter}$ denote the functional that is associated with the recursive definition of $iter$. Since the base case is trivial, we immediately come to the inductive step. Let $it$ be such that the induction hypothesis $it\ F \sqsubseteq Q$ holds. We calculate for the conclusion of the inductive step:

$$\begin{array}{ll}
 & \tau_{iter}\ it\ F \sqsubseteq Q \\
\Leftrightarrow & \quad \{\text{definition of } \tau_{iter}\} \\
 & \lambda x, i.\ \mathbf{if}\ \ i = 0\ \ \mathbf{then}\ \ x\ \ \mathbf{else}\ \ it\ F(F\ x, i-1)\ \ \mathbf{fi}\ \ \sqsubseteq Q \\
\Leftarrow & \quad \{\text{induction hypothesis}\} \\
 & \lambda x, i.\ \mathbf{if}\ \ i = 0\ \ \mathbf{then}\ \ x\ \ \mathbf{else}\ \ Q(F\ x, i-1)\ \ \mathbf{fi}\ \ \sqsubseteq Q
\end{array}$$

Here we did apply the induction hypothesis, because $Q$ became an outermost operation in the else-branch.

**(End of fixed point analysis, and fixed point induction.)**

The last line cannot yet be turned into a definition of $Q$, because it still contains $F$. Therefore we bring it into a form to which we can again apply our development method. The last line is implied by conjunction of the following two inequations:

$$\begin{array}{ll}
 & \forall x : x \equiv Q(x, 0) \\
\textbf{(Q)} & \forall x, i : Q(F\ x, i) \sqsubseteq Q(x, i+1)
\end{array}$$

We recursively apply our development method to **(Q)**.

**Fixed point analysis, and fixed point induction.** Since we do not know $Q$, we can only do fixed point induction on $F$ in specification **(Q)**.

By requiring strictness

$$\textbf{(S)} \qquad \forall i : Q(\bot, i) \equiv \bot\ ,$$

we strengthen the base case.

Let $f$ be such that the induction hypothesis $\forall x, i : Q(f\,x, i) \sqsubseteq Q(x, i{+}1)$ holds. We calculate for the conclusion of the inductive step:

$$\begin{array}{ll}
& Q(\tau_F\ f\ x, i) \sqsubseteq Q(x, i+1) \\
\Leftrightarrow & \quad \{\text{definition of } \tau_F\} \\
& Q(\textbf{if}\ p\,x\ \textbf{then}\ g\,x\ \textbf{else}\ f(f(h\,x))\ \textbf{fi}, i) \sqsubseteq Q(x, i+1) \\
\Leftrightarrow & \quad \{\textbf{(S)}\} \\
& \textbf{if}\ p\,x\ \textbf{then}\ Q(g\,x, i)\ \textbf{else}\ Q(f(f(h\,x)), i)\ \textbf{fi}\ \sqsubseteq \\
& Q(x, i+1) \\
\Leftarrow & \quad \{\text{induction hypothesis}\} \\
& \textbf{if}\ p\,x\ \textbf{then}\ Q(g\,x, i)\ \textbf{else}\ Q(f(h\,x), i+1)\ \textbf{fi}\ \sqsubseteq \\
& Q(x, i+1) \\
\Leftarrow & \quad \{\text{induction hypothesis}\} \\
& \textbf{if}\ p\,x\ \textbf{then}\ Q(g\,x, i)\ \textbf{else}\ Q(h\,x, i+2)\ \textbf{fi}\ \sqsubseteq Q(x, i+1)
\end{array}$$

**(End of fixed point analysis, and fixed point induction.)**

This last requirement on $Q$ together with the above requirement on $Q(x, 0)$ leads us to the following recursive definition:

$$\begin{array}{lll}
Q & \equiv & \textbf{rec}\ q.\ \lambda x, i.\ \textbf{if}\ \ i = 0\ \ \textbf{then}\ x \\
& & \qquad\qquad\qquad\quad \textbf{else if}\ p\,x\ \ \textbf{then}\ q(g\,x, i-1) \\
& & \qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ q(h\,x, i+1)\ \textbf{fi}\ \ \textbf{fi}
\end{array}$$

The fixed point rule immediately shows that this definition meets the last inequation on $Q$. In addition, the so defined $Q$ is strict in its first argument, and thus fulfils **(S)**.

We still must show **(AuxLE)**:

$$Q \sqsubseteq \mathit{iter}\ F$$

**Proof** We do fixed point induction on $Q$. As the base case holds trivially, we immediately turn to the inductive step. Let $q$ be such that the induction hypothesis $q \sqsubseteq \mathit{iter}\ F$ holds. We prove for the inductive step:

$$\lambda x, i.\ \textbf{if}\ \ i = 0\ \ \textbf{then}\ \ x$$

$$\qquad\qquad\textbf{else if } p\,x \textbf{ then } q(g\,x, i-1)$$

$$\qquad\qquad\qquad\qquad\textbf{else } q(h\,x, i+1) \textbf{ fi} \textbf{ fi}$$

$$\sqsubseteq \qquad \{\text{induction hypothesis}\}$$

$$\lambda x, i.\, \textbf{if } i = 0 \textbf{ then } x$$

$$\qquad\qquad\textbf{else if } p\,x \textbf{ then } \mathit{iter}\,F(g\,x, i-1)$$

$$\qquad\qquad\qquad\qquad\textbf{else } \mathit{iter}\,F(h\,x, i+1) \textbf{ fi} \textbf{ fi}$$

$$\equiv \qquad \{\text{definition of } \mathit{iter}\}$$

$$\lambda x, i.\, \textbf{if } i = 0 \textbf{ then } x$$

$$\qquad\qquad\textbf{else if } p\,x \textbf{ then } \mathit{iter}\,F(g\,x, i-1)$$

$$\qquad\qquad\qquad\textbf{else } \mathit{iter}\,F(F(F(h\,x)), i-1) \textbf{ fi} \textbf{ fi}$$

$$\equiv \qquad \{\text{strictness of } \mathit{iter}\,F\}$$

$$\lambda x, i.\, \textbf{if } i = 0 \textbf{ then } x$$

$$\qquad\textbf{else } \mathit{iter}\,F(\textbf{if } p\,x \textbf{ then } g\,x \textbf{ else } F(F(h\,x)) \textbf{ fi},$$

$$\qquad\qquad\qquad i-1) \textbf{ fi}$$

$$\equiv \qquad \{\text{definition of } F\}$$

$$\lambda x, i.\, \textbf{if } i = 0 \textbf{ then } x \textbf{ else } \mathit{iter}\,F(F\,x, i-1) \textbf{ fi}$$

$$\equiv \qquad \{\text{definition of } \mathit{iter}\}$$

$$\mathit{iter}\,F$$

**(End of proof.)**

This proof concludes the development.

## 3.2 Two while-loops

In this example we will consider the following property of while-loops:

$$\textbf{while } b \textbf{ do } P \textbf{ od } ; \textbf{ while } b \vee c \textbf{ do } P \textbf{ od } \equiv$$
$$\textbf{while } b \vee c \textbf{ do } P \textbf{ od}$$

This property has been proved in [38] by fixed point induction. In [27] the same property has been proved by an algebraic technique. Although leading to elegant proofs, this algebraic technique has two disadvantages: according to the authors, one disadvantage is that certain functions are needed that do not always exist. Secondly, it is not (yet) obvious, how proofs in this technique can systematically be constructed.

We will turn this property of while-loops into a development task: given the sequential composition of the two while-loops

$$\textbf{while } b \textbf{ do } P \textbf{ od } ; \textbf{ while } b \vee c \textbf{ do } P \textbf{ od } \ ,$$

how can we find an equivalent, single while-loop? We will construct such a while-loop by our development method for recursion. In particular, we do not use any operational knowledge or reasoning about while-loops.

First we give a suitable formalization of the problem in our language. As is well-known, while-loops can be given semantics by use of the recursion-operator. We choose a presentation that gives a nice property to our development: it could literally be understood as a development in a generalization of Dijkstra's guarded command language using algebraic laws of programming as given in [37].

Let us understand commands as functions on a state space $S$. Let $F$, $G$, $H$ and $P$ be function symbols with functionality $S \rightarrow S$. Let $f$, $g$ and $h$ be variables of type $S \rightarrow S$. Let $b$ and $c$ be functions of type $S \rightarrow Bool$ such $c\,s \equiv \perp$ implies $b\,s \equiv \perp$ that for all $s$. In addition, we will use the following four functions:

- The function $SKIP : S \rightarrow S$ denotes the identity on states.
- The function $\perp : S \rightarrow S$ denotes the everywhere undefined function. It represents the command $ABORT$.
- The function $.;. : (S \rightarrow S) \times (S \rightarrow S) \rightarrow (S \rightarrow S)$ denotes function composition: $f; g \equiv \lambda s.\ g(f(s))$. It represents sequential composition of commands.
- The function $.\triangleleft.\triangleright. : (S \rightarrow S) \times (S \rightarrow Bool) \times (S \rightarrow S) \rightarrow (S \rightarrow S)$. It represents the conditional.

We merely needed states $S$ in order to formalize the problem. In the sequel, we will write programs without explicit use of states. Thus they look like programs in the generalization of Dijkstra's guarded command language, as presented in [37]. Moreover, the programs here obey the laws of that language.

Now we can write the two while-loops in our notation as follows, and give them names $G$ and $H$:

$$\begin{aligned} G &\equiv \textbf{rec } g.\ (P; g) \triangleleft b \triangleright SKIP \\ H &\equiv \textbf{rec } h.\ (P; h) \triangleleft b \vee c \triangleright SKIP \end{aligned}$$

We are looking for a recursive definition of $F$ with only one "iteration", which is equivalent to the sequential composition of loops $G$ and $H$. Thus we write as a specification of $F$:

$$\textbf{(S)} \qquad G; H \equiv F$$

We will develop a program for $F$ from the inequation

$$\textbf{(GE)} \qquad G; H \sqsubseteq F\ ,$$

and prove that the other inequation

$$\textbf{(LE)} \qquad F \sqsubseteq G; H$$

holds for the constructed $F$.

Since $G$ and $H$ are recursively defined, we try fixed point induction in **(LE)**.

**Fixed point analysis.** It is easy to see that in fixed point induction on $H$, the induction hypothesis is not applicable in the inductive step: in the inductive step $h$ occurs in the subterm $G; P; h$, whereas the induction hypothesis has the form $G; h \sqsubseteq F$.

Now let us analyse fixed point induction on $G$. Let $\tau_G$ denote the functional associated with the recursive definition of $G$. We get the base case

$$\textbf{(B)} \qquad \bot; H \sqsubseteq F\ ,$$

and the inductive step

$$\textbf{(I)} \qquad \forall g : g; H \sqsubseteq F \Rightarrow (\tau_G\, g); H \sqsubseteq F\ .$$

By definition of $\tau_G$, the inductive step is equivalent to

$$\begin{array}{ll} & \forall g : g; H \sqsubseteq F \Rightarrow ((P; g) \lhd b \rhd SKIP); H \sqsubseteq F \\ \Leftrightarrow & \{\text{H is strict, and distributes backwards;} \\ & SKIP \text{ is the identity of sequential composition}\} \\ & \forall g : g; H \sqsubseteq F \Rightarrow (P; g; H) \lhd b \rhd H \sqsubseteq F \end{array}$$

The induction hypothesis is applicable to the conclusion of the inductive step. But its application would leave $H$ in the left hand side of the inequation. As $H$ is recursively defined, this would not directly lead to a while-loop for $F$.

Therefore, we next analyse where a fixed point induction on $H$ in the conclusion leads. After applying some laws, we obtain as inductive step

$$\begin{array}{l} \forall h : \\ (P; g; H) \lhd b \rhd h \sqsubseteq F \Rightarrow \\ (P; g; H) \lhd b \rhd (P; h \lhd c \rhd SKIP) \sqsubseteq F \end{array}$$

Since the induction variable $h$ occurs in the context $\ldots \lhd b \rhd h$ in the induction hypothesis, but in a different context in the conclusion, the induction hypothesis cannot be applied to the conclusion. Hence fixed point induction on $H$ fails.

**(End of fixed point analysis.)**

Fixed point analysis has taken us to a point where typically a design decision has to be made:

**Design decision.** We have found a fixed point induction, in which the induction hypothesis is fully applicable, but which is not promising for the development, because the recursively defined function $H$, which we would like to eliminate, remains in the specification. As we have seen, no other fixed point induction is possible. Therefore, we make a design decision based on the current specification

$$\forall g : g; H \sqsubseteq F \Rightarrow (P; g; H) \lhd b \rhd H \sqsubseteq F\ .$$

If $b$ evaluates to false, it is necessary that $H \sqsubseteq F$ holds. The restriction to this case caused the problem when we tried the inner fixed point induction on $H$. Therefore we try to strengthen the specification by requiring the inequality $H \sqsubseteq F$ independently of the value of $b$. Hence we can state our design decision as follows:

$$\begin{array}{lll} & & \forall g : g; H \sqsubseteq F \Rightarrow (P; g; H) \triangleleft b \triangleright H \sqsubseteq F \\ & \Leftarrow & \quad \{\text{strengthening by conjunction}\} \\ \mathbf{(D)} & & \forall g : (g; H \sqsubseteq F \Rightarrow (P; g; H) \triangleleft b \triangleright H \sqsubseteq F) \wedge H \sqsubseteq F \end{array}$$

The new conjunct immediately leads us to take the definition of $H$ as definition of $F$:

$$F \equiv \mathbf{rec}\ f.\ (P; f) \triangleleft b \vee c \triangleright SKIP$$

**(End of design decision.)**

We still must prove the remaining conjunct of **(D)** for the chosen definition of $F$.

**Proof**

$$\begin{array}{ll} & \forall g : g; H \sqsubseteq F \Rightarrow (P; g; H) \triangleleft b \triangleright H \sqsubseteq F \\ \Leftarrow & \quad \{\text{induction hypothesis; definition of } F\} \\ & \forall g : g; H \sqsubseteq F \Rightarrow (P; F) \triangleleft b \triangleright F \sqsubseteq F \\ \Leftarrow & \quad \{\text{law of conditional, and condition on } b \text{ and } c\} \\ & \forall g : g; H \sqsubseteq F \Rightarrow ((P; F) \triangleleft b \vee c \triangleright SKIP) \triangleleft b \triangleright F \sqsubseteq F \\ \Leftarrow & \quad \{\text{definition of } F\} \\ & \forall g : g; H \sqsubseteq F \Rightarrow F \triangleleft b \triangleright F \sqsubseteq F \\ \Leftarrow & \quad \{\text{law of conditional}\} \\ & \forall g : g; H \sqsubseteq F \Rightarrow F \sqsubseteq F \\ \Leftarrow & \quad \{\text{reflexivity of } \sqsubseteq\} \\ & true \end{array}$$

**(End of proof.)**

The base case **(B)** follows immediately from the definition of $F$.

Now only **(LE)** remains to be shown:

$$F \sqsubseteq G; H$$

**Proof** We again proceed by fixed point induction on $F$. The base case is trivial. The inductive step is proved as follows, where $\tau_F$ denotes the functional associated with the recursive definition of $F$: Let $f$ be such that the induction hypothesis $f \sqsubseteq G; H$ holds. We prove the conclusion of the inductive step:

$$\begin{array}{ll}
 & \tau_F\ f \\
\equiv & \quad \{\text{definition of } \tau_F\} \\
 & (P; f) \triangleleft b \vee c \triangleright SKIP \\
\sqsubseteq & \quad \{\text{induction hypothesis}\} \\
 & (P; G; H) \triangleleft b \vee c \triangleright SKIP \\
\sqsubseteq & \quad \{\text{case analysis}\} \\
 & ((P; G; H) \triangleleft b \vee c \triangleright SKIP) \triangleleft \neg b \wedge c \triangleright \\
 & ((P; G; H) \triangleleft b \vee c \triangleright SKIP) \\
\sqsubseteq & \quad \{\text{simplification}\} \\
 & (P; G; H) \triangleleft \neg b \wedge c \triangleright ((P; G; H) \triangleleft b \triangleright SKIP) \\
\sqsubseteq & \quad \{\text{definition of } G \text{ and } H\} \\
 & (P; G; H) \triangleleft \neg b \wedge c \triangleright ((G; H) \triangleleft b \triangleright (G; H)) \\
\sqsubseteq & \quad \{\text{law of conditional}\} \\
 & (P; G; H) \triangleleft \neg b \wedge c \triangleright (G; H) \\
\sqsubseteq & \quad \{\text{already proved inequation } \textbf{(GE)}\text{, and } F \equiv H\} \\
 & (P; H) \triangleleft \neg b \wedge c \triangleright (G; H) \\
\sqsubseteq & \quad \{\text{definition of } H\} \\
 & H \triangleleft \neg b \wedge c \triangleright (G; H) \\
\sqsubseteq & \quad \{\text{definition of } G\} \\
 & G; H \triangleleft \neg b \wedge c \triangleright (G; H) \\
\sqsubseteq & \quad \{\text{law of conditional}\} \\
 & G; H
\end{array}$$

**(End of proof.)**

Hence we have developed a single while-loop $F$ that meets the original specification **(S)**.

## 3.3 Compiler correctness

In this section we will develop a code generator for a small functional programming language. Code is generated for a stack machine. We are only interested in the code generator here, and not in other parts of the compiler (e.g. the parser). Therefore, whenever we say ”compiler”, we only refer to the code generator throughout this section.

We have chosen compiler development as an example for the application of our method, because it is a rather difficult development task, when the source language allows recursive function declarations, but the target language does not.

Although being heavily studied in the literature, compiler correctness proofs fall into two categories in the literature: in the first category, the source language of the compiler does not contain recursion (cf. e.g. [4], [11], [13], [15]). In the second category of compiler correctness proofs, the source language does contain recursion, but so does the target language (cf. e.g. [6], [14]).

We are interested in source languages with recursion, and target languages without recursion. A denotational semantics is given for the source language, and an operational semantics is given for the target stack machine. One could define a compiler, and then prove it correct with respect to these two semantics. But we will develop a compiler systematically from its specification, thereby establishing its correctness.

Compiler correctness proofs in the same semantic setting as here have been studied in [8] and [39]. There the difficulties have been shown, but an unnecessarily complicated solution was presented, which used a lot of insight into the problem domain. We base our language on those of these two works.

### 3.3.0 Common basis of source language, and target language

Both source language, and target language use the sort

$$Data\ ,$$

which we need not specify further. $Data$ contains the data elements that are manipulated by programs both of the source language, and of the target language. For convenience, we content ourselves with one sort of data elements.

In addition, both source language, and target language use the sort

$$Fsb\ ,$$

which we again need not specify further. $Fsb$ contains the predefined function symbols of the source language, and of the target language. For convenience, all function symbols are unary.

We further assume a function

$$cfct : Fsb \to (Data \to Data)$$

to be given. It assigns a function $cfct(g)$ to each predefined function symbol $g \in Fsb$.

### 3.3.1 Source language

The source language is kept as simple as possible, in order to concentrate on recursive function definition.

#### Syntax

The abstract syntax of the source language is given by the sorts

$$Exp$$

and

$$Fp\ ,$$

which represent the syntactic classes of expressions and functional programs respectively.

The elements of sort $Exp$ are inductively generated by the strict constructors

$$\begin{aligned}
&cst : Data \rightarrow Exp \ , \\
&x : \rightarrow Exp \ , \\
&if : Exp, Exp, Exp \rightarrow Exp \ , \text{ and} \\
&app : Fsb, Exp \rightarrow Exp \ .
\end{aligned}$$

The constructor $cst$ stands for constants of sort $Data$. We assume that $x$ is the only variable of the source language. The constructor $if$ introduces conditional expressions. The constructor $app$ denotes application of functions to expressions.

The sort $Fp$ of functional programs has the only strict constructor

$$\langle . \rangle : Exp, Exp \rightarrow Fp \ .$$

The constructor $\langle . \rangle$ builds a functional program from two expressions: the first expression is the body of a recursively defined function, the second expression represents the main program. For simplicity, we assume that in every program the name of the recursively defined function is $f$, where $f$ is a function symbol of $Fsb$. The recursive declaration overwrites the predefined meaning of $f$.

**(End ”Syntax”.)**

**Semantics**

We give a denotational semantics of the source language. As metalanguage we use our language of chapter 2.

The semantics of expressions is given by the higher order function

$$val : Exp \rightarrow (Data \rightarrow Data) \rightarrow (Data \rightarrow Data) \ .$$

The function $val$ assigns a function $val\,e\,r$ to each expression $e$ and each function $r$ in the following way: function $r$ is taken as interpretation of function symbol $f$; under this interpretation, expression $e$ is interpreted as a function of the value of variable $x$.

We assume a continuous function

$$test : Data \to Bool \ ,$$

which assigns a boolean value to each element of $Data$. We need not specify $test$ further.

The function $val$ is strict in its first argument. Now we give a recursive definition of $val$. (In order to make the definition more readable, we use pattern matching as described in chapter 2.)

$$\begin{aligned}
&val(cst(c))r\,d \equiv \textbf{if}\ \ d = d\ \ \textbf{then}\ \ c\ \textbf{else}\ \ c\ \textbf{fi}\\
&val\,x\,r\,d \equiv d\\
&val(if(e_0, e_1, e_2))r\,d \equiv \textbf{if}\ \ test(val\,e_0\,r\,d)\ \textbf{then}\ \ val\,e_1\,r\,d\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ val\,e_2\,r\,d\ \textbf{fi}\\
&val(app(g, e))r\,d \equiv \textbf{if}\ \ g = f\ \ \textbf{then}\ \ r(val\,e\,r\,d)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ cfct(g)(val\,e\,r\,d)\ \textbf{fi}
\end{aligned}$$

In the sequel we write $\tau_{val}$ for the functional associated with the recursive definition of $val$.

The semantics of functional programs is defined by the function

$$mean : Fp \to (Data \to Data) \ .$$

The semantics of a functional program is a function from the value of the variable $x$ into $Data$, where the function symbol $f$ is interpreted according to the recursive declaration of the program. As usual, the least fixed point of the functional associated with the recursive declaration is bound to $f$.

The function $mean$ is strict. For $b \not\equiv \perp$, and $m \not\equiv \perp$ we define it by the following equation (again using pattern matching):

$$mean\langle b, m\rangle \equiv val\,m(\textbf{rec}\ r.\ val\,b\,r)$$

**(End "Semantics".)**

### 3.3.2 Target language

Now we turn to the definition of the target language. The architecture we use is a simple stack machine.

#### Syntax

The target machine consists of four components:

- a (read-only) memory, which stores the assembler program to be executed,
- a stack of data values and labels,
- a program counter, which contains the postfix of the assembler program that remains to be executed, and
- an accumulator, which holds one data value.

Formally we define the sort $Conf$ of machine configurations as the smash product of the sorts of assembler programs, stacks, assembler programs, and data values:

$$Conf \equiv Asp \otimes Stack \otimes Asp \otimes Data$$

Sort $Stack$ contains all possibly empty, finite lists of data values and labels. For sake of readability, we let the stack be inhomogeneous, that is, it may contain both data values, and labels as stack elements

$$StackEl\,.$$

We use the following operations on stacks:

$$\begin{aligned}&\varepsilon :\rightarrow Stack\\&\langle . \rangle : StackEl \rightarrow Stack\\&. \circ . : Stack, Stack \rightarrow Stack\end{aligned}$$

We write $\varepsilon$ for the empty stack, and $\langle . \rangle$ for the stack with one element. The operator $\circ$ denotes concatenation of stacks. These three operators are the constructors of $Stack$.

The reader might object that these are not the stack operations, and thus we are defining lists instead of stacks. We have taken this implementation of stacks by lists, in order to keep our programs and proofs readable. But we shall operate on stacks only as we could do with stack operations, throughout.

We need four sorts to define the syntax of assembler programs:

$$Com\ ,$$
$$Asp\ ,$$
$$Label\ , \text{and}$$
$$Mark\ .$$

The sort $Com$ contains the assembler commands, $Asp$ the assembler programs. The sort $Label$ contains all labels; most labels are constructed of elements of sort $Mark$.

Now we list the commands of the target language, that is, the strict constructors of sort $Com$:

$$appcfct : Fsb \to Com$$
$$lab : Label \to Com$$
$$jump : Label \to Com$$
$$cjump : Label \to Com$$
$$return : \to Com$$
$$swap : \to Com$$
$$push : StackEl \to Com$$
$$pushA : \to Com$$

The command $appcfct(g)$ applies the predefined function $cfct(g)$, which is associated with $g$, to the topmost stack element. The command $lab(l)$ defines a label. When command $jump(l)$ has been executed, program execution continues at label $l$. When a conditional jump $cjump(l)$ is executed, program execution switches to label $l$ only, if the topmost stack element has a certain value. The command $return$ leaves the topmost element on the stack, but removes the next two elements. The first of these elements is the label, where program execution is to be continued; the second element is written into the accumulator. The command *swap* exchanges the content of the accumulator with the topmost stack element. The command *push* pushes a data value or a label onto the stack, whereas *pushA* pushes the accumulator content onto the stack.

Assembler programs are possibly empty, finite lists of commands. Therefore we again take the strict list constructors as constructors of sort *Asp*:

$$\varepsilon :\rightarrow Asp$$
$$\langle . \rangle : Com \rightarrow Asp$$
$$. \circ . : Asp, Asp \rightarrow Asp$$

These operators denote the empty assembler program, the assembler programs of only one command, and concatenation of assembler programs, in turn.

Labels are non-empty finite sequences of marks:

$$\langle . \rangle : Mark \rightarrow Label$$
$$. \circ . : Label, Label \rightarrow Label$$

Sort *Mark* is generated by the following constructors:

$$body, main, 0, 1, 2, 3 :\rightarrow Mark$$

**(End "Syntax".)**

**Semantics**

Now we define an operational semantics for assembler programs. We again use the language of chapter 2 as meta-language. For notational convenience, we again use pattern matching.

The strict function

$$step : Conf \rightarrow Conf$$

executes the next command, that is, the first command of the program counter.

We now define the function *step*, using the function *goto*, which we shall define afterwards.

$$\begin{aligned}
&\mathrm{step}\,\langle p, \langle d\rangle \circ s, \langle appcfct(g)\rangle \circ pc, a\rangle \equiv \\
&\qquad \langle p, \langle (cfct(g))d\rangle \circ s, pc, a\rangle \\
&\mathrm{step}\,\langle p, s, \langle lab(l)\rangle \circ pc, a\rangle \equiv \langle p, s, pc, a\rangle \\
&\mathrm{step}\,\langle p, s, \langle jump(l)\rangle \circ pc, a\rangle \equiv \langle p, s, goto(l, p), a\rangle \\
&\mathrm{step}\,\langle p, \langle d\rangle \circ s, \langle cjump(l)\rangle \circ pc, a\rangle \equiv \\
&\qquad \langle p, s, \mathbf{if}\ \ test(d)\ \ \mathbf{then}\ \ goto(l, p)\ \ \mathbf{else}\ \ pc\ \mathbf{fi}, a\rangle \\
&\mathrm{step}\,\langle p, \langle d_0\rangle \circ \langle l\rangle \circ \langle d_1\rangle \circ s, \langle return\rangle \circ pc, a\rangle \equiv \\
&\qquad \langle p, \langle d_0\rangle \circ s, goto(l, p), d_1\rangle \\
&\mathrm{step}\,\langle p, \langle d\rangle \circ s, \langle swap\rangle \circ pc, a\rangle \equiv \langle p, \langle a\rangle \circ s, pc, d\rangle \\
&\mathrm{step}\,\langle p, s, \langle push(se)\rangle \circ pc, a\rangle \equiv \langle p, \langle se\rangle \circ s, pc, a\rangle \\
&\mathrm{step}\,\langle p, s, \langle pushA\rangle \circ pc, a\rangle \equiv \langle p, \langle a\rangle \circ s, pc, a\rangle
\end{aligned}$$

The strict function

$$goto : Label, Asp \to Asp\ ,$$

which branches to a label in an assembler program, is recursively defined as follows:

$$\begin{aligned}
&goto(l, \varepsilon) \equiv \varepsilon \\
&goto(l, \langle c\rangle \circ p) \equiv \mathbf{if}\ \ c = lab(l)\ \ \mathbf{then}\ \ p\ \ \mathbf{else}\ \ goto(l, p)\ \mathbf{fi}
\end{aligned}$$

The operational semantics of assembler programs is given by the strict function

$$exec : Conf \to Conf\ ,$$

which executes an assembler program by applying function *step* until the program counter becomes empty.

The strict function *exec* is recursively defined as follows:

$$\begin{aligned}
&exec\langle p, s, \varepsilon, a\rangle \equiv \langle p, s, \varepsilon, a\rangle \\
&exec\langle p, s, \langle c\rangle \circ pc, a\rangle \equiv exec\langle step\langle p, s, \langle c\rangle \circ pc, a\rangle\rangle
\end{aligned}$$

Finally, we define the strict function

$$res : Conf \rightarrow Data \ ,$$

which extracts a data value as a result from a configuration.

The strict function *res* is defined as follows:

$$res\langle p, \langle d\rangle \circ s, pc, a\rangle \equiv d$$

**(End "Semantics".)**

### 3.3.3 A compiler development

In this section we will develop a compiler from its specification by stepwise refinement. We proceed according to our development method for recursion.

**Compiler specification.** A compiler from $Fp$ to $Asp$ is a function

$$comp : Fp \rightarrow Asp$$

with the following property: Execution of each compiled program yields a value that is greater than or equal to the semantics of the source program. This specification of the compiler can formally be stated as follows:

$$\textbf{(S)} \qquad \forall fp \in Fp, d \in Data : \\ mean\ fp\ d \sqsubseteq res(exec\langle comp\ fp, \varepsilon, comp\ fp, d\rangle)$$

**(End "Compiler specification".)**

Our task is to develop a program for *comp* that fulfils specification **(S)**.

First we substitute *mean* by its non-recursive definition. This is the only transformation we can do in the above specification without knowing anything about *comp*. Thus we obtain the following specification **(Comp)**, which is equivalent to **(S)**:

$$\textbf{(Comp)} \qquad \forall b, m \in Exp, d \in Data : b \not\equiv \bot \Rightarrow \\ val\ m(\mathbf{rec}\ r.\ val\ b\ r)d \sqsubseteq \\ res(exec\langle comp\langle b, m\rangle, \varepsilon, comp\langle b, m\rangle, d\rangle)$$

Specification **(Comp)** is an inequation with recursion (in $val$ and $\mathbf{rec}\ r$) on its left hand side, and the unknown $comp$ on its right hand side. According to our method, we start with fixed point analysis.

**Fixed point analysis.** Recursion occurs three times in the left hand side

$$val\, m(\mathbf{rec}\ r.\ val\, b\, r)d$$

of the inequation in **(Comp)**: besides $val$, which has been defined recursively, the subterm $\mathbf{rec}\ r.\ val\, b\, r$ contains recursion.

To which of the three occurences of the recursion operator shall we best apply the fixed point induction rule?

For better readability, we abbreviate the right hand side of the inequation by $rhs$.

As we have seen, there are three candidates for fixed point induction in the left hand side of the inequation. Let us consider them in turn.

For convenience, we repeat the fixed point induction rule:

$$\frac{\begin{array}{c} A[\bot/x] \\ \forall x(A \Rightarrow A[t/x]) \end{array}}{A[\mathbf{rec}\ x.\ t/x]} \qquad \text{where } A \text{ is syntactically admissible in } x$$

**Analysis 0.** Let us first analyse fixed point induction on the outer $val$ in the term $val\, m(\mathbf{rec}\ r.\ val\, b\, r)d$.

It turns out that the induction hypothesis is applicable in the inductive step, but leaves the recursion $\mathbf{rec}\ r.\ val\, b\, r$ in the term. Therefore we do not apply the induction hypothesis, and instead apply fixed point analysis recursively. But both fixed point induction on $\mathbf{rec}\ r$ and on $val$ fail, because the induction hypotheses are not applicable in the inductive steps for structural reasons that cannot be circumvented by generalizations. Hence fixed point induction on the outer $val$ fails.

**(End of analysis 0.)**

Hence we must try another candidate for fixed point induction.

**Analysis 1.** Let us now analyse fixed point induction on the inner $val$ in the term $val\, m(\mathbf{rec}\ r.\ val\, b\, r)d$.

It is easy to see that the induction hypothesis is not applicable in the inductive step for similar reasons as in analysis 0. Thus fixed point induction on the inner $val$ fails, too.

**(End of analysis 1.)**

As no fixed point induction has been successful so far, we now try the last candidate for fixed point induction.

**Analysis 2.** Now we analyse fixed point induction on **rec** $r$ in the term

$$val\, m(\mathbf{rec}\ r.\ val\, b\, r)d\ .$$

Variable $b$ is free in the subterm **rec** $r.\ val\, b\, r$, on which we will try fixed point induction. But $b$ is universally quantified in the formula **(Comp)**, which we will refine. Therefore we must take

$$\forall m, d : b \not\equiv \bot \Rightarrow val\, m\, r\, d \sqsubseteq rhs$$

as formula $A$ of the fixed point induction rule, where $r$ is the induction variable. In this formula, $b$ is free and not explicitly universally quantified.

Thus we get the base case

$$\forall m, d : b \not\equiv \bot \Rightarrow val\, m \perp d \sqsubseteq rhs\ ,$$

and the inductive step

$$\begin{array}{l}\forall r :\\ (\forall m, d : b \not\equiv \bot \Rightarrow val\, m\, r\, d \sqsubseteq rhs) \Rightarrow\\ (\forall m, d : b \not\equiv \bot \Rightarrow val\, m(val\, b\, r)d \sqsubseteq rhs)\ .\end{array}$$

Let us first turn to the base case. We recursively apply fixed point analysis to $val$. The base case is trivial. The inductive step is

$$\forall v :$$

$$\begin{aligned}&(\forall m, d : b \not\equiv \bot \Rightarrow v\, m \perp d \sqsubseteq rhs) \Rightarrow\\&(\forall m, d : b \not\equiv \bot \Rightarrow (\tau_{val}\, v) m \perp d \sqsubseteq rhs)\ .\end{aligned}$$

Evaluation of $\tau_{val}$ leads to case distinction on $m$. All cases of $m$ allow complete application of the induction hypothesis.

Now we turn to the inductive step. We recursively apply fixed point analysis to the conclusion. The conclusion contains two occurrences of $val$, to which we could apply fixed point induction.

Let us first analyse fixed point induction to the inner $val$. We get the inductive step

$$\begin{aligned}&\forall v :\\&(\forall m, d : b \not\equiv \bot \Rightarrow val\, m(v\, b\, r)d \sqsubseteq rhs) \Rightarrow\\&(\forall m, d : b \not\equiv \bot \Rightarrow val\, m((\tau_{val}\, v)\, b\, r)d \sqsubseteq rhs)\ .\end{aligned}$$

Hence, if we applied fixed point induction to the inner $val$, we would be faced with a problem that arose in analysis 0: since $b$ is bound outside the inductive step, but $v$ is applied to other expressions than $b$ when $\tau_{val}\, v$ is evaluated, the induction hypothesis is not applicable. Therefore, fixed point induction on the inner $val$ fails.

Let us now analyse fixed point induction on the outer $val$ in term $val\, m(val\, b\, r)d$.

The base case is trivial. The inductive step is

$$\begin{aligned}&\forall v :\\&(\forall m, d : b \not\equiv \bot \Rightarrow v\, m(val\, b\, r)d \sqsubseteq rhs) \Rightarrow\\&(\forall m, d : b \not\equiv \bot \Rightarrow (\tau_{val}\, v)m(val\, b\, r)d \sqsubseteq rhs)\ .\end{aligned}$$

If $m$ has one of the forms $\bot$, $cst(c)$, $x$, $if(e_0, e_1, e_2)$, and $app(g, e)$ with $g \not\equiv f$, then the induction hypothesis is fully applicable. The most difficult case is $m \equiv app(f, e)$. In this case, the conclusion of the inductive step evaluates to

$$\forall d : b \not\equiv \bot \Rightarrow (val\, b\, r)(v\, e(val\, b\, r)d) \sqsubseteq rhs\ .$$

The subterm $v\ e(val\ b\ r)d$ has the form of the hypothesis of the fixed point induction on $val$. The other subterm, $val\ b\ r$, has the form of the hypothesis of the fixed point induction on $\mathbf{rec}\ r.\ val\,b\,r$. Therefore, also in case $m \equiv app(f,e)$, the induction hypotheses are completely applicable.

Hence in this analysis we have found a proof strategy in which the induction hypotheses are completely applicable.

**(End of analysis 2.)**

**(End of fixed point analysis.)**

Having found a refinement strategy in analysis 2 so that the induction hypotheses are completely applicable, we will now try to reduce the context of the unknown *comp* in specification **(Comp)**.

**Context reduction.** If the left hand side of the inequation of **(Comp)** were surrounded by the same functions as *comp* on the right hand side, we could strengthen **(Comp)** due to monotonicity by removing these functions from both sides.

From the definition of *res* and *exec* we know that specification **(Comp)** is equivalent to

$$\begin{array}{l}\forall b,m,d : b \not\equiv \bot \Rightarrow\\ \exists p \not\equiv \bot :\\ res(exec(p,\langle val\,m(\mathbf{rec}\ r.\ val\,b\,r)d\rangle,\varepsilon,d)) \sqsubseteq\\ res(exec(comp\langle b,m\rangle,\varepsilon,comp\langle b,m\rangle,d))\ .\end{array}$$

Since *res* is monotonic, this formula can be strengthened by

$$\begin{array}{ll}\textbf{(Exec)} & \forall b,m,d : b \not\equiv \bot \wedge m \not\equiv \bot \Rightarrow\\ & comp\langle b,m\rangle \not\equiv \bot \wedge\\ & exec(comp\langle b,m\rangle,\langle val\,m(\mathbf{rec}\ r.\ val\,b\,r)d\rangle,\varepsilon,d) \sqsubseteq\\ & exec(comp\langle b,m\rangle,\varepsilon,comp\langle b,m\rangle,d)\ .\end{array}$$

(Note that $p \equiv comp(b,m)$ is the only choice we have for $p$ in strengthening the formula since the first component of the machine state does not change during execution.) Thus we have at least removed *res* from the context of *comp*. Obviously, the context cannot be reduced further.

**(End of context reduction.)**

**Fixed point induction.** If we try to apply fixed point induction to **(Exec)** in the way we found out to be successful for **(Comp)**, the induction hypothesis turns out to be too weak. In the base case of the induction on **rec** $r$, and the inductive step of the induction on $val$, we have to show (case $m \equiv if(e_0, e_1, e_2)$):

$$\begin{aligned}
&\forall b, m, e, d : \exists a \not\equiv \bot : \\
&v\; e_0 \perp d \equiv tt \Rightarrow \\
&exec(comp(b, m), \langle v\; e_1 \perp d\rangle, \varepsilon, a) \sqsubseteq \\
&exec(comp(b, m), \varepsilon, comp(b, m), d)\ .
\end{aligned}$$

The induction hypothesis is not applicable since $m$ and $e_1$ do not coincide as they do in the hypothesis. Therefore we must generalize the specification.

**(End of fixed point induction.)**

**Generalization.** In

$$exec(comp\langle b, m\rangle, \langle val\; m(\mathbf{rec}\ r.\ val\, b\, r)d\rangle, \varepsilon, d)$$

we must split variable $m$ into two variables, say $m$ and $e$. Thus we get

$$exec(comp\langle b, m\rangle, \langle val\; e(\mathbf{rec}\ r.\ val\, b\, r)d\rangle, \varepsilon, d)\ .$$

If we did not change anything else in the formula, we should get the strengthened specification

$$\begin{aligned}
&\forall b, m, e, d : b \not\equiv \bot \wedge m \not\equiv \bot \Rightarrow \\
&comp\langle b, m\rangle \not\equiv \bot \wedge \\
&exec(comp\langle b, m\rangle, \langle val\; e(\mathbf{rec}\ r.\ val\, b\, r)d\rangle, \varepsilon, d) \sqsubseteq \\
&exec(comp\langle b, m\rangle, \varepsilon, comp\langle b, m\rangle, d)\ .
\end{aligned}$$

When strengthening a specification, we must always be careful not to strengthen it too much. In our case, the strengthened specification must still be satisfiable by some function *comp*. The strengthened specification implies that the value of every expression $e$ is less than or equal to the result of executing program $comp\langle b, m\rangle$. Certainly, this condition is not satisfiable by any *comp*.

Hence we must make the right hand side of the inequation dependent on $e$. Can we simply replace $comp\langle b, m\rangle$ by $comp\langle b, e\rangle$ in the third argument of *exec*? We know about the machine architecture that the program counter (third component of the machine state) is a postfix of the entire program (first component of the machine state). Thus $comp\langle b, e\rangle$ would have to be a postfix of $comp\langle b, m\rangle$ for every expression $e$. Obviously, this requirement again is not satisfiable by any compiler *comp*.

Therefore it seems reasonable to restrict the specification to those $e$, for which $comp\langle b, e\rangle$ is a postfix of $comp\langle b, m\rangle$. Originally, the generalization was intended for those $e$ that are subterms of $m$. But we cannot expect that the code of each subterm $e$ of $m$ will be a postfix of the code of $m$. Therefore we only require that $comp\langle b, e\rangle$ is the beginning of a postfix. Let us formalize the postfix relation on assembler programs by the predicate $\preceq$ :

$$p \preceq q \Leftrightarrow \exists r : r \circ p \equiv q$$

Thus we get

$$\begin{aligned}
&\forall b, m, e, d : b \not\equiv \bot \wedge m \not\equiv \bot \Rightarrow \\
&comp\langle b, m\rangle \not\equiv \bot \wedge \\
&(\forall pc : comp\langle b, e\rangle \circ pc \preceq comp\langle b, m\rangle \Rightarrow \\
&exec(comp\langle b, m\rangle, \langle val\, e(\mathbf{rec}\ r.\ val\, b\, r)d\rangle, pc, d) \sqsubseteq \\
&exec(comp\langle b, m\rangle, \varepsilon, comp\langle b, e\rangle \circ pc, d))
\end{aligned}$$

as a new candidate for a generalized specification.

Is this specification still too strong? Let $e$ be a subterm of $m$. Code that is generated for $e$ will, in general, depend on the position at which $e$ occurs in $m$: if $e$ occurs several times in $m$, and its code contains some labelled statement, then the labels must be different for all occurrences of $e$. Since $comp\langle b, e\rangle$ does not depend on $m$, and thus does not depend on the position of $e$ in $m$, the code for $e$ cannot be generated by *comp*. Therefore we introduce a function

$$cexp : Exp, Label \to Asp\ ,$$

which compiles expressions, using only labels that are determined by its second argument.

We base the function *comp* on *cexp* in the following way:

$$\begin{array}{ll} \textbf{(Prog)} & \forall b, m : b \not\equiv \bot \wedge m \not\equiv \bot \Rightarrow \\ & comp\langle b, m\rangle \equiv \langle jump(main)\rangle \circ \langle lab(body)\rangle \circ \\ & \qquad cexp(b, \langle 0\rangle) \circ \langle return\rangle \circ \langle lab(main)\rangle \circ \\ & \qquad cexp(m, \langle 1\rangle) \wedge \\ & cexp(b, \langle 0\rangle) \not\equiv \bot \wedge cexp(m, \langle 1\rangle) \not\equiv \bot \end{array}$$

Now we can formulate the generalized specification by making use of *cexp*. Since it seems too restrictive to assume an empty stack whenever a compiled expression is executed, we further generalize the specification to arbitrary stacks $s$:

$$\begin{array}{ll} \textbf{(Cexp)} & \forall b, m, d, e, pc, l, s : b \not\equiv \bot \wedge m \not\equiv \bot \Rightarrow \\ & (cexp(e, l) \circ pc \preceq comp\langle b, m\rangle \Rightarrow \\ & exec(comp\langle b, m\rangle, \langle val\ e(\mathbf{rec}\ r.\ val\ b\ r)d\rangle \circ s, pc, d) \sqsubseteq \\ & exec(comp\langle b, m\rangle, s, cexp(e, l) \circ pc, d)) \end{array}$$

In order that specification **(Exec)** is implied, we require as a third property

$$\begin{array}{ll} \textbf{(JumpMain)} & \forall b, m, d : b \not\equiv \bot \wedge m \not\equiv \bot \Rightarrow \\ & exec(comp\langle b, m\rangle, \varepsilon, cexp(m, \langle 1\rangle), d) \sqsubseteq \\ & exec(comp\langle b, m\rangle, \varepsilon, comp\langle b, m\rangle, d) \end{array}$$

Conjunction of **(Prog)**, **(Cexp)** and **(JumpMain)** implies specification **(Exec)**.

**(End of generalization.)**

**Fixed point induction.** Now we carry out the fixed point induction of analysis 2 for the generalized specification **(Cexp)**.

According to analysis 2, we do fixed point induction on **rec** $r$. Let us first address the base case, and the inductive step thereafter.

**Base case.** Analogously to the base case in analysis 2, we get the following base case for the generalization **(Cexp)**:

$$\begin{aligned}
&\forall b, m, d, e, pc, l, s : b \not\equiv \perp \wedge m \not\equiv \perp \Rightarrow \\
&(cexp(e, l) \circ pc \preceq comp\langle b, m\rangle \Rightarrow \\
&exec(comp\langle b, m\rangle, \langle val\ e \perp d\rangle \circ s, pc, d) \sqsubseteq \\
&exec(comp\langle b, m\rangle, s, cexp(e, l) \circ pc, d))
\end{aligned}$$

As planned in analysis 2, this formula is refined by fixed point induction on $val$. Since the base case is trivial, we immediately turn to the inductive step.

**Inductive step.** The inductive step instantiates to

$$\begin{aligned}
&\forall v : \\
&(\forall b, m, d, e, pc, l, s : b \not\equiv \perp \wedge m \not\equiv \perp \Rightarrow \\
&(cexp(e, l) \circ pc \preceq comp\langle b, m\rangle \Rightarrow \\
&exec(comp\langle b, m\rangle, \langle v\ e \perp d\rangle \circ s, pc, d) \sqsubseteq \\
&exec(comp\langle b, m\rangle, s, cexp(e, l) \circ pc, d))) \\
&\Rightarrow \\
&(\forall b, m, d, e, pc, l, s : b \not\equiv \perp \wedge m \not\equiv \perp \Rightarrow \\
&(cexp(e, l) \circ pc \preceq comp\langle b, m\rangle \Rightarrow \\
&exec(comp\langle b, m\rangle, \langle (\tau_{val}\ v) e \perp d\rangle \circ s, pc, d) \sqsubseteq \\
&exec(comp\langle b, m\rangle, s, cexp(e, l) \circ pc, d)))\ .
\end{aligned}$$

Assume that the induction hypothesis holds.

The definition of $\tau_{val}$ suggests a case distinction on $e$:

- If $e \equiv \perp$, the conclusion holds trivially.

- If $e \equiv cst(c)$ (with $c \not\equiv \bot$), the conclusion is implied by

$$\begin{aligned}
&\forall b, m, d, pc, l, s : b \not\equiv \bot \wedge m \not\equiv \bot \Rightarrow \\
&(cexp(cst(c), l) \circ pc \preceq comp\langle b, m\rangle \Rightarrow \\
&exec(comp\langle b, m\rangle, \langle c\rangle \circ s, pc, d) \sqsubseteq \\
&exec(comp\langle b, m\rangle, s, cexp(cst(c), l) \circ pc, d))\ .
\end{aligned}$$

Looking at the machine instructions, we immediately find the following explicit condition on *cexp*, which implies the preceeding formula:

$$\begin{aligned}
\textbf{(Cst)}\quad &\forall c, l : c \not\equiv \bot \wedge l \not\equiv \bot \Rightarrow \\
&cexp(cst(c), l) \equiv \langle push(c)\rangle
\end{aligned}$$

- If $e \equiv x$, the conclusion is equivalent to

$$\begin{aligned}
&\forall b, m, d, pc, l, s : b \not\equiv \bot \wedge m \not\equiv \bot \Rightarrow \\
&(cexp(x, l) \circ pc \preceq comp\langle b, m\rangle \Rightarrow \\
&exec(comp\langle b, m\rangle, \langle d\rangle \circ s, pc, d) \sqsubseteq \\
&exec(comp\langle b, m\rangle, s, cexp(x, l) \circ pc, d))\ .
\end{aligned}$$

This formula is implied by the following explicit condition on *cexp*, which is again suggested by the machine instructions:

$$\textbf{(X)}\quad \forall l : l \not\equiv \bot \Rightarrow cexp(x, l) \equiv \langle pushA\rangle$$

- If $e \equiv if(e_0, e_1, e_2)$ (with $e_0, e_1, e_2 \not\equiv \bot$), the conclusion is equivalent to

$$\begin{aligned}
&\forall b, m, d, pc, l, s : b \not\equiv \bot \wedge m \not\equiv \bot \Rightarrow \\
&(cexp(if(e_0, e_1, e_2), l) \circ pc \preceq comp\langle b, m\rangle \Rightarrow \\
&\mathrm{exec}(comp\langle b, m\rangle, \\
&\quad \langle \textbf{if } test(v\, e_0 \perp d) \textbf{ then } v\, e_1 \perp d \textbf{ else } v\, e_2 \perp d \textbf{ fi}\rangle \\
&\quad \circ\, s, pc, d) \sqsubseteq \\
&\mathrm{exec}(comp\langle b, m\rangle, s, cexp(if(e_0, e_1, e_2), l) \circ pc, d))\ .
\end{aligned}$$

Looking at the machine instructions, we immediately find the following explicit condition on *cexp*:

$$
\begin{array}{ll}
\textbf{(If)} & \forall e_0, e_1, e_2, l : \\
& e_0 \not\equiv \bot \wedge e_1 \not\equiv \bot \wedge e_2 \not\equiv \bot \wedge l \not\equiv \bot \Rightarrow \\
& cexp(if(e_0, e_1, e_2), l) \equiv \\
& cexp(e_0, l \circ \langle 0 \rangle) \circ \langle cjump(l) \rangle \circ cexp(e_2, l \circ \langle 2 \rangle) \circ \\
& \langle jump(l \circ \langle 3 \rangle) \rangle \circ \langle lab(l) \rangle \circ cexp(e_1, l \circ \langle 1 \rangle) \circ \\
& \langle lab(l \circ \langle 3 \rangle) \rangle
\end{array}
$$

This formula implies the preceeding one by induction hypothesis, but an additional property is needed: we must assure that the labels $l$ and $l \circ \langle 3 \rangle$, to which the generated code may branch, do not occur in preceeding program parts. Therefore we require that no label is defined twice in a program:

$$
\begin{array}{ll}
\textbf{(Lab)} & \forall b, m : b \not\equiv \bot \wedge m \not\equiv \bot \Rightarrow \forall p, q, r, l, k : \\
& comp\langle b, m \rangle \equiv p \circ \langle lab(l) \rangle \circ q \circ \langle lab(k) \rangle \circ r \Rightarrow \\
& l \not\equiv k
\end{array}
$$

- If $e \equiv app(g, e_0)$ (with $g \not\equiv f$, $g \not\equiv \bot$, and $e_0 \not\equiv \bot$), the conclusion is equivalent to

$$
\begin{array}{l}
\forall b, m, d, pc, l, s : b \not\equiv \bot \wedge m \not\equiv \bot \Rightarrow \\
(cexp(app(g, e_0), l) \circ pc \preceq comp\langle b, m \rangle \Rightarrow \\
exec(comp\langle b, m \rangle, \langle cfct(g)(v\ e_0 \perp d) \rangle \circ s, pc, d) \sqsubseteq \\
exec(comp\langle b, m \rangle, s, cexp(app(g, e_0), l) \circ pc, d)) \ .
\end{array}
$$

The machine instructions together with the induction hypothesis again suggest an explicit condition on *cexp*, which implies the preceeding formula:

$$
\begin{array}{ll}
\textbf{(App)} & \forall g, e_0, l : g \not\equiv f \wedge g \not\equiv \bot \wedge e_0 \not\equiv \bot \wedge l \not\equiv \bot \Rightarrow \\
& cexp(app(g, e_0), l) \equiv \\
& cexp(e_0, l \circ \langle 0 \rangle) \circ \langle appcfct(g) \rangle
\end{array}
$$

- If $e \equiv app(f, e_0)$ (with $e_0 \not\equiv \bot$), the conclusion is trivially true.

**(End of inductive step.)**

**(End of base case.)**

**Inductive step.** Now we come to the inductive step of the fixed point induction on **rec** $r$ in specification **(Cexp)**. Analogously to analysis 2, we get the inductive step

$$\begin{array}{l}
\forall b, r : \\
(\forall m, d, e, pc, l, s : b \not\equiv \bot \wedge m \not\equiv \bot \Rightarrow \\
(cexp(e, l) \circ pc \preceq comp\langle b, m\rangle \Rightarrow \\
exec(comp\langle b, m\rangle, \langle val\; e\; r\; d\rangle \circ s, pc, d) \sqsubseteq \\
exec(comp\langle b, m\rangle, s, cexp(e, l) \circ pc, d))) \\
\Rightarrow \\
(\forall m, d, e, pc, l, s : b \not\equiv \bot \wedge m \not\equiv \bot \Rightarrow \\
(cexp(e, l) \circ pc \preceq comp\langle b, m\rangle \Rightarrow \\
exec(comp\langle b, m\rangle, \langle val\; e(val\; b\; r)d\rangle \circ s, pc, d) \sqsubseteq \\
exec(comp\langle b, m\rangle, s, cexp(e, l) \circ pc, d)))\ .
\end{array}$$

Assume that the induction hypothesis is true.

As planned in analysis 2, the conclusion is refined by fixed point induction on the outer $val$. Since the base case holds trivially, we immediately turn to the inductive step.

**Inductive step.** We obtain the following formula as inductive step:

$$\begin{array}{l}
\forall v : \\
(\forall m, d, e, pc, l, s : b \not\equiv \bot \wedge m \not\equiv \bot \Rightarrow \\
(cexp(e, l) \circ pc \preceq comp\langle b, m\rangle \Rightarrow \\
exec(comp\langle b, m\rangle, \langle v\; e(val\; b\; r)d\rangle \circ s, pc, d) \sqsubseteq \\
exec(comp\langle b, m\rangle, s, cexp(e, l) \circ pc, d))) \\
\Rightarrow \\
(\forall m, d, e, pc, l, s : b \not\equiv \bot \wedge m \not\equiv \bot \Rightarrow \\
(cexp(e, l) \circ pc \preceq comp\langle b, m\rangle \Rightarrow \\
exec(comp\langle b, m\rangle, \langle(\tau_{val}\; v)\; e(val\; b\; r)d\rangle \circ s, pc, d) \sqsubseteq \\
exec(comp\langle b, m\rangle, s, cexp(e, l) \circ pc, d)))
\end{array}$$

Let the induction hypothesis be true.

The definition of $\tau_{val}$ again suggests case distinction on $e$:

- If $e \equiv \bot$, the conclusion holds trivially.

- If $e \equiv cst(c)$, we get the identical development as in the corresponding case of $e$ within the base case of the induction on **rec** $r$.

- If $e \equiv x$, we get the identical development as in the corresponding case of $e$ within the base case of the induction on **rec** $r$.

  If $e \equiv if(e_0, e_1, e_2)$, then we develop the same specifications **(If)** and **(Lab)** as in the corresponding case above, by an analogous development.

  If $e \equiv app(g, e_0)$ (with $g \not\equiv f$), then we develop the same specification **(App)** as in the corresponding case above, by an analogous development.

  If $e \equiv app(f, e_0)$ (with $e_0 \not\equiv \bot$), the conclusion is equivalent to

$$\begin{aligned}
&\forall m, d, e, pc, l, s : b \not\equiv \bot \wedge m \not\equiv \bot \Rightarrow\\
&(cexp(app(f, e_0), l) \circ pc \preceq comp\langle b, m\rangle \Rightarrow\\
&exec(comp\langle b, m\rangle, \langle (val\, b\, r)(v\; e_0(val\, b\, r)d)\rangle \circ s, pc, d)\\
&\sqsubseteq exec(comp\langle b, m\rangle, s, cexp(app(f, e_0), l) \circ pc, d))
\end{aligned}$$

  Inspection of the machine instructions leads us to the following explicit condition on $cexp$, which implies the preceeding formula because of the induction hypothesis, and because of specification **(Lab)**:

$$\textbf{(Appf)} \quad \begin{aligned}
&\forall e_0, l : e_0 \not\equiv \bot \wedge l \not\equiv \bot \Rightarrow\\
&cexp(app(f, e_0), l) \equiv\\
&cexp(e_0, l \circ \langle 0\rangle) \circ \langle swap\rangle \circ \langle push(l)\rangle \circ\\
&\langle jump(body)\rangle \circ \langle lab(l)\rangle
\end{aligned}$$

**(End of inductive step.)**

**(End of inductive step.)**

**(End of fixed point induction.)**

The refinement by fixed point induction has lead to an explicit condition on *cexp* for each case of the argument expression. They suggest a recursive definition of *cexp*.

**Compiler function.** By use of the fixed point rule, one can immediately give a recursive definition that meets the specifications **(Cst)**, **(X)**, **(If)**, **(App)**, and **(Appf)**. (For better readability, we again use the notation of pattern matching, which was introduced in chapter 2.)

$$\begin{aligned}
&\mathrm{cexp}\,(cst(c), l) \equiv \langle push(c)\rangle\\
&\mathrm{cexp}\,(x, l) \equiv \langle pushA\rangle\\
&\mathrm{cexp}\,(if(e_0, e_1, e_2), l) \equiv cexp(e_0, l \circ \langle 0\rangle) \circ \langle cjump(l)\rangle \circ\\
&\qquad cexp(e_2, l \circ \langle 2\rangle) \circ \langle jump(l \circ \langle 3\rangle)\rangle \circ \langle lab(l)\rangle \circ\\
&\qquad cexp(e_1, l \circ \langle 1\rangle) \circ \langle lab(l \circ \langle 3\rangle)\rangle\\
&\mathrm{cexp}\,(app(g, e), l) \equiv cexp(e_0, l \circ \langle 0\rangle) \circ\\
&\qquad \textbf{if } g = f\\
&\qquad \textbf{then } \langle swap\rangle \circ \langle push(l)\rangle \circ \langle jump(body)\rangle \circ \langle lab(l)\rangle\\
&\qquad \textbf{else } \langle appcfct(g)\rangle \textbf{ fi}
\end{aligned}$$

The first conjunct of specification **(Prog)** suggests how to base *comp* on *cexp*:

$$\begin{aligned}
comp\langle b, m\rangle \equiv\ & \langle jump(main)\rangle \circ \langle lab(body)\rangle \circ cexp(b, \langle 0\rangle) \circ\\
& \langle return\rangle \circ \langle lab(main)\rangle \circ cexp(m, \langle 1\rangle)
\end{aligned}$$

Thus we have developed a program for *comp*.

**(End "Compiler function".)**

During the development we introduced some more specifications. They now remain to be proved for *cexp* and *comp* as they have been defined meanwhile.

Let us start with **(Prog)**. The first conjunct has directly been taken as definition of *comp*. The remaining two conjuncts are implied by the formula

$$\forall e, l : e \not\equiv \bot \wedge l \not\equiv \bot \Rightarrow cexp(e, l) \not\equiv \bot$$

which can be shown by structural induction on $e$.

In order to prove **(Lab)**, one uses the definition of *comp* in terms of *cexp*, and shows by structural induction on expressions that *cexp* generates distinct labels.

Finally, specification **(JumpMain)** is an immediate consequence of **(Lab)**.

These proofs conclude the compiler development.

## 3.4 From a denotational semantics to an operational semantics

In this example we develop an operational semantics from a denotational one by our method.

The literature contains some adequacy proofs of operational and denotational semantics (e.g. [32], [62], [16]). But they are based on natural operational semantics or on two quite similar semantic definitions. In addition, we do not assume the operational semantics to be given, but we develop it from the denotational semantics.

The development in the sequel has been mechanically verified. For further detail we refer to [34].

### 3.4.0 Syntax of the language

The sort

$$D$$

stands for the data elements of our example language. We assume that

$$true : D$$

$$false : D$$

are two constructors of $D$. There may be many more, which we do not specify. We only presuppose that $D$ carries a flat order, which can straightforwardly be formalized:

$$\forall x, y \in D : x \sqsubseteq y \Leftrightarrow x \equiv \bot \vee x \equiv y$$

The sort

$$X$$

stands for the variables of our example language. For our purpose, $X$ need not be further specified.

The sort

$$P$$

stands for the predefined function symbols of our example language. For convenience we assume that all function symbols of $P$ are unary.

The sort

$$U$$

stands for the user-definable function symbols of our example language. For convenience we assume that all function symbols of $U$ are unary.

The sort

$$Term$$

stands for the terms of our example language.

The sort $Term$ is generated by the following strict constructors:

$$cst : D \rightarrow Term$$

$$var : X \rightarrow Term$$

$$.\,(\,.\,) : P, Term \rightarrow Term$$

$$.\,(\,.\,) : U, Term \rightarrow Term$$

$$if\ .\ then\ .\ else\ .\ fi : Term, Term, Term \rightarrow Term$$

$$(\,.\,(\,.\,) :\ .\,)\,. : U, X, Term, Term \rightarrow Term$$

For better readability, the constructors have been overloaded.

We define the set $FV$ of free variables of a term as usual. It is a subset of $X \cup U$. Function definition is the only binding operator: in $(f(x) : t_0)t_1$ variables $f$ and $x$ that are free in $t_0$ are bound by $(f(x) : t_0)$. Since it is standard, we omit the formal definition of free variables.

Based on the definition of free variables, we define a predicate

$$CT$$

of type $Term$, which characterizes the set of closed terms: as usual, closed terms are terms without free variables.

In addition, we need a boolean function

$$W : Term \to Bool$$

which characterizes the subset of terms that are generated only by $cst$ and application $.\,(\,.\,) : P, Term \to Term$ of predefined function symbols. $W$ can be understood as the word algebra on which terms of the language $Term$ are built.

### 3.4.1 Denotational semantics

Environments associate data values with variables, and functions with user-definable function symbols:

$$Env \equiv List(X \times D) \times List(U \times (D \to D))$$

By $void$ we denote the empty environment, that is, the product of empty lists:

$$void \equiv (\varepsilon, \varepsilon)$$

The function

$$in$$

checks if a variable or function symbol is in an environment.

The function

$$\cdot\,[\,\cdot\,,\,\cdot\,]$$

denotes update of environments. As usual, we write $\eta[d/x, r/f]$ for the update of environment $\eta$ by data element $d$ for variable $x$, and by function $r$ for function symbol $f$.

The functions

$$lookupvar$$

and

$$lookupfct$$

yield the data element associated with a variable by an environment, and the function associated with a function symbol by an environment, respectively. If no data element or function is associated with a variable or with a function symbol in an environment, $lookupvar$ and $lookupfct$ yield $\perp$.

Since the formal definitions of all these functions are straightforward, we do not give them.

The function

$$I : P \to (D \to D)$$

assigns a continuous function to each predefined function symbol. We assume that for all $p \in P$ the associated function $I(p)$ is strict.

It is convenient to overload the symbol $I$: the function

$$I : W \to D$$

interprets terms of the word algebra as data elements, that is, it assigns semantics to terms of the word algebra. (Here $W$ stands for the set of all $t$ with $W\ t = tt$. The function $I$ is defined recursively on the structure of those terms:

$$I(cst(d)) = d$$
$$I(p(t)) = (I\ p)(I\ t)$$
$$I(\perp) = \perp$$

The function

$$\mathcal{T} : Term \rightarrow Env \rightarrow D$$

assigns a denotational semantics to terms. For better readability, we use semantic braces for $\mathcal{T}$. They are not to be confused with the braces at the meta-level.

Let $\mathcal{T}$ be strict in its first argument.

$$\begin{aligned}
&\mathcal{T}[\![cst(d)]\!]\eta \equiv d\\
&\mathcal{T}[\![var(x)]\!]\eta \equiv lookupvar\ x\ \eta\\
&\mathcal{T}[\![p(t)]\!]\eta \equiv (I\ p)(\mathcal{T}[\![t]\!]\eta)\\
&\mathcal{T}[\![f(t)]\!]\eta \equiv (lookupfct\ f\ \eta)(\mathcal{T}[\![t]\!]\eta)\\
&\mathcal{T}[\![if\ t_0\ then\ t_1\ else\ t_2\ fi]\!] \equiv \textbf{if}\ \mathcal{T}[\![t_0]\!]\eta = true\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then}\ \ \mathcal{T}[\![t_1]\!]\eta\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \ \textbf{if}\ \mathcal{T}[\![t_0]\!]\eta = false\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \textbf{then}\ \ \mathcal{T}[\![t_2]\!]\eta\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \textbf{else}\ \ \bot\ \textbf{fi}\ \ \textbf{fi}\\
&\mathcal{T}[\![(f(x) : t_0)t_1]\!]\eta \equiv (\textbf{rec}\ r.\ \lambda d.\ \mathcal{T}[\![t_0]\!](\eta[d/x, r/f]))(\mathcal{T}[\![t_1]\!]\eta)
\end{aligned}$$

Let $\tau_{\mathcal{T}}$ be the functional associated with the recursive definition of $\mathcal{T}$.

### 3.4.2 Operational basis

We specify that the operational semantics must be a term rewriting semantics. The semantics is then developed in the next section.

The function

$$normal : Term \rightarrow Bool$$

determines whether a closed term is in normal form.

We presuppose the following properties of *normal*: all constants are in normal form:

$$\forall d \in D : d \not\equiv \bot \Rightarrow normal(cst(d)) \equiv tt$$

Only terms of the word algebra can be normal forms:

$$\forall t \in Term : CT(t) \Rightarrow (normal\ t \equiv tt \Rightarrow W(t) \equiv tt)$$

If a function application is in normal form, then the argument term is in normal form:

$$\begin{array}{l}\forall p \in P, t \in Term : \\ W(t) \equiv tt \Rightarrow (normal(p(t)) \equiv tt \Rightarrow normal\ t \equiv tt)\end{array}$$

The function *normal* is undefined exactly for the undefined term:

$$\forall t \in Term : CT(t) \Rightarrow (normal\ t \equiv \bot \Leftrightarrow t \equiv \bot)$$

The structure of the term rewriting machine is given by the function

$$val : Term \rightarrow Term\ ,$$

which is defined only on closed terms. It is recursively defined as follows:

$$val \equiv \textbf{rec}\ v.\ \lambda t.\ \textbf{if}\ \ normal\ t\ \ \textbf{then}\ \ t\ \ \textbf{else}\ \ v(reduce\ t)\ \textbf{fi}$$

Let $\tau_{val}$ denote the functional associated with the recursive definition of *val*.

Our task is to develop a program for *reduce* such that the operational semantics corresponds to the denotational semantics in a way to be made precise below.

In addition, we are given an evaluator

$$eval : W \rightarrow W$$

for terms of the word algebra. We presuppose some properties of this evaluator: it preserves the interpretation of terms:

$$\forall t \in W : I(eval\ t) \equiv I\ t$$

It takes every term to a normal form:

$$\forall t \in W : t \not\equiv \bot \Rightarrow normal(eval\ t) \equiv tt$$

Terms that are already in normal form are not changed under *eval*:

$$\forall t \in W : normal\ t \equiv tt \Rightarrow eval\ t \equiv t$$

### 3.4.3 Development of an operational semantics

**Specification.** We require of the operational semantics the following property:

$$(\mathbf{S}) \qquad \forall t \in Term : CT(t) \Rightarrow \mathcal{T}[\![t]\!]void \sqsubseteq I(val[\![t]\!])$$

**(End "Specification".)**

**Fixed point analysis.** The only recursively defined object in the left hand side of the inequation of **(S)** is $\mathcal{T}$. Thus we analyse fixed point induction on $\mathcal{T}$. Since the base case is trivial, we immediately turn to the inductive step:

$$\begin{aligned}&\forall h :\\ &(\forall t : CT(t) \Rightarrow h[\![t]\!]void \sqsubseteq I(val[\![t]\!])) \Rightarrow\\ &(\forall t : CT(t) \Rightarrow (\tau_{\mathcal{T}}\ h)[\![t]\!]void \sqsubseteq I(val[\![t]\!]))\end{aligned}$$

In the conclusion, evaluation of $\tau_{\mathcal{T}}\ h$ leads to a case distinction on $t$. If $t$ is of the form $(f(x) : t_0)t_1$, then $(\tau_{\mathcal{T}}\ h)[\![t]\!]void$ is equal to

$$(\mathbf{rec}\ r.\ \lambda d.\ h[\![t_0]\!](void[d/x, r/f]))(h[\![t_1]\!]void)\ .$$

In general, $t_0$ is not a closed term, and $void[d/x, r/f]$ is not $void$. Therefore the induction hypothesis is not applicable; it is too weak.

**(End of fixed point analysis.)**

According to our development method, we look for a generalization of **(S)** so that fixed point induction on $\mathcal{T}$ is possible for this generalization. For that purpose we must make design decisions.

**Design decision.** We try to find a suitable generalization by analysing why fixed point induction on $\mathcal{T}$ failed. We have seen that $t$ must not be restricted to closed terms, and that the denotational value of $t$ must be taken in arbitrary environments, instead of just $void$. Hence we must consider an inequation

$$\mathcal{T}[\![t]\!]\eta \sqsubseteq I(val\, u)\ ,$$

where $t \in Term$, $\eta \in Env$, and $u$ is a closed term. It is self-evident that in this generality the inequation is not fulfillable by any $val$. Hence this inequation strengthens **(S)** too much.

Therefore we must weaken this inequation by adding a suitable premise. Since it is not fulfillable for arbitrary $t$, $\eta$ and $u$ as above, we need a suitable relation $\sim$ between $(t, \eta)$ on the one hand, and $u$ on the other hand. We are heading for a new specification of the form

$$\textbf{(G)} \qquad \begin{array}{l} \forall t, u \in Term, \eta \in Env : \\ CT(u) \wedge (t, \eta) \sim u \Rightarrow \mathcal{T}[\![t]\!]\eta \sqsubseteq I(val\; u) \; . \end{array}$$

Therefore we need a relation $\sim$ with the following properties:

- The formula **(G)** implies the original specification **(S)**, that is,

  $$\forall t \in Term : CT(t) \Rightarrow (t, void) \sim t$$

  holds.
- The formula **(G)** is fulfillable by some $val$.
- The formula **(G)** is refinable by fixed point induction on $\mathcal{T}$.

Since we are interested in an operational semantics that works by term rewriting, the free variables and function symbols of $t$ must be substituted by suitable terms and function declarations in $u$. Thus we state as first requirement on $\sim$:

$$(t, \eta) \sim u \Rightarrow \exists \sigma \in Subst : t\,\sigma \doteq u$$

Function declarations are of the form

$$f(x) : t$$

where $f \in U$, $x \in X$ and $t \in Term$. We write $Decl$ for the set of all function declarations. We write $CDecl$ for the set of all closed function declarations, that is, all function declarations in $Decl$ without free variables.

The set $Subst$ consists of functions from $X$ to $Term$, and from $U$ to $U$ and $Decl$. We omit the formal definition.

Since only closed terms are operationally evaluated, we immediately get the following restriction on substitutions: Application of substitutions leads to closed terms. Therefore free variables and free function symbols of a term are always substituted by closed terms and closed declarations, respectively. Hence application substitutions to terms can be defined without the usual renaming of bound variables in order to avoid name clashes. Application of a substitution $\sigma$ to a term $t$ is written in postfix-notation: $t\ \sigma$. It is strict and defined recursively on the structure of terms as follows:

$$
\begin{aligned}
&(cst(d))\sigma \equiv cst(d)\\
&(var(x))\sigma \equiv \sigma(x)\\
&(p(t))\sigma \equiv p(t\ \sigma)\\
&(f(t))\sigma \equiv (\sigma(f))(t\ \sigma)\\
&(if\ t_0\ then\ t_1\ else\ t_2\ fi)\sigma \equiv if(t_0\ \sigma)then(t_1\ \sigma)else(t_2\ \sigma)fi\\
&((f(x):t_0)t_1)\sigma \equiv (f(x):(t_0(\sigma[x/x,f/f])))(t_1\ \sigma)
\end{aligned}
$$

But this requirement on $\sim$ is not enough: substituting the free variables of $t$ by arbitrary terms and declarations will not ensure the required relation between the two semantics. In addition, a relation between the values that the environment $\eta$ assigns to the free variables of $t$, and the terms or declarations that are substituted for the free variables in $u$ is needed.

Let us first turn to the variables of $X$ that occur free in $t$. It seems reasonable to require for all $x \in X$ that are free in $t$: the value that $\eta$ associates with $x$ is less or equal to the value that is obtained by evaluating the term substituted for $x$ in $u$ operationally.

We summarize the mentioned properties of variables in a predicate $WV$ ("weaker in variable"):

$$
\begin{aligned}
&\forall \eta \in Env, \sigma \in Subst, x \in X :\\
&WV(\eta,\sigma,x) \Leftrightarrow CT(\sigma(x)) \wedge lookupvar\ x\ \eta \sqsubseteq I(val(\sigma(x)))
\end{aligned}
$$

Now we require a similar property for function symbols $f \in U$ that occur free in $t$: if $\eta$ associates a function $r$ with $f$, then $\sigma$ substitutes $f$ by a declaration $\Delta$ so that the following holds: $r$ is less or equal to the function that is obtained by operationally evaluating $\Delta$. If $\eta$ assigns no value to $f$, then $\sigma$ does not substitute $f$.

We formalize this property of function symbols by the following predicate $WF$ ("weaker in function"):

$$\begin{aligned}
&\forall \eta \in Env, \sigma \in Subst, f \in U : \\
&WF(\eta, \sigma, f) \Leftrightarrow \\
&((f\ in\ \eta) \Rightarrow\ CDecl(\sigma(f)) \wedge \\
&\qquad\qquad \forall d \in D, t \in Term : CT(t) \wedge d \sqsubseteq I(val\ t) \Rightarrow \\
&\qquad\qquad (lookupfct\ f\ \eta) \sqsubseteq I(val((\sigma(f))(t)))) \wedge \\
&(\neg(f\ in\ \eta) \Rightarrow \sigma(f) \equiv f)
\end{aligned}$$

We define the relation $\sim$ as follows:

$$\begin{aligned}
&\forall t, u \in Term, \eta \in Env : CT(u) \Rightarrow \\
&\exists \sigma \in Subst : t\ \sigma \equiv u \wedge \\
&\qquad\qquad (\forall x \in X : x \in FV(t) \Rightarrow WV(\eta, \sigma, x)) \wedge \\
&\qquad\qquad (\forall f \in U : f \in FV(t) \Rightarrow CDecl(\sigma\ f)) \wedge \\
&\qquad\qquad (\forall f \in U : WF(\eta, \sigma, f))
\end{aligned}$$

We call $\sigma$ a "substitution belonging to $(t, \eta)$ and $u$".

For the so defined relation $\sim$ it is obvious that **(G)** implies the original **(S)**.

**(End of design decision.)**

**Fixed point analysis, and fixed point induction.** The base case is trivially true. The inductive step for the new specification **(G)** is

$$\begin{aligned}
&\forall h : \\
&(\forall t, u, \eta : CT(u) \wedge (t, \eta) \sim u \Rightarrow h[\![t]\!]\eta \sqsubseteq I(val\ u)) \Rightarrow \\
&(\forall t, u, \eta : CT(u) \wedge (t, \eta) \sim u \Rightarrow (\tau_T\ h)[\![t]\!]\eta \sqsubseteq I(val\ u))\ .
\end{aligned}$$

Let $h$ be such that the induction hypothesis

$$\forall t, u, \eta : CT(u) \wedge (t, \eta) \sim u \Rightarrow h[\![t]\!]\eta \sqsubseteq I(val\ u)$$

holds.

Let $t$, $u$ and $\eta$ be such that

$$CT(u) \wedge (t, \eta) \sim u$$

holds.

Calculation of $(\tau_{\mathcal{I}}\ h)[\![t]\!]\eta$ immediately leads to a case distinction on $t$:

- If $t \equiv \bot$, the conclusion holds trivially.
- If $t \equiv cst(d)$ (with $d \not\equiv \bot$), we calculate:

$$\begin{array}{ll}
 & (\tau_{\mathcal{I}}\ h)[\![cst(d)]\!]\eta \sqsubseteq I(val\ u) \\
\Leftrightarrow & \quad \{\text{definition of } \tau_{\mathcal{I}}\} \\
 & d \sqsubseteq I(val\ u) \\
\Leftrightarrow & \quad \{\text{premise } (cst(d), \eta) \sim u \text{ implies } u \equiv cst(d)\} \\
 & d \sqsubseteq I(val(cst(d))) \\
\Leftrightarrow & \quad \{\text{definition of } val;\ normal(cst(d)) \equiv tt\} \\
 & d \sqsubseteq I(cst(d)) \\
\Leftrightarrow & \quad \{\text{definition of } I\} \\
 & true
\end{array}$$

- If $t \equiv var(x)$ (with $x \not\equiv \bot$), we calculate:

$$\begin{array}{ll}
 & (\tau_{\mathcal{I}}\ h)[\![var(x)]\!]\eta \sqsubseteq I(val\ u) \\
\Leftrightarrow & \quad \{\text{definition of } \tau_{\mathcal{I}}\} \\
 & lookupvar\ x\ \eta \sqsubseteq I(val\ u) \\
\Leftrightarrow & \quad \{\text{premise } (var(x), \eta) \sim u \text{ implies that there exists} \\
 & \quad \text{a substitution } \sigma \text{ such that } u \equiv \sigma(x) \text{ and} \\
 & \quad WV(\eta, \sigma, x)\} \\
 & true
\end{array}$$

- If $t \equiv p(t_0)$ (with $p \not\equiv \bot$ and $t_0 \not\equiv \bot$), let $\sigma$ be a substitution belonging to $(p(t_0), \eta)$ and $u$ (its existence follows from $(p(t_0), \eta) \sim u$). We calculate:

$$\begin{array}{ll}
 & (\tau_{\mathcal{T}} h)[\![p(t_0)]\!]\eta \sqsubseteq I(val\, u) \\
\Leftrightarrow & \quad \{\text{definition of } \tau_{\mathcal{T}}\text{; choice of } \sigma\} \\
 & (I\ p)(h[\![t_0]\!]\eta) \sqsubseteq I(val((p(t_0))\sigma)) \\
\Leftarrow & \quad \{(t_0, \eta) \sim t_0\sigma\text{; } CT(t_0\sigma)\text{; induction hypothesis}\} \\
 & (I\ p)(I(val(t_0\sigma))) \sqsubseteq I(val((p(t_0))\sigma))
\end{array}$$

The last inequation suggests a new specification for $val$, by which it is implied:

$$\textbf{(P)} \qquad \forall t : CT(t) \Rightarrow (I\ p)(I(val\ t)) \sqsubseteq I(val(p(t)))$$

- If $t \equiv f(t_0)$ (with $f \not\equiv \bot$ and $t_0 \not\equiv \bot$), let $\sigma$ be a substitution belonging to $(f(t_0), \eta)$ and $u$ (its existence follows from $(f(t_0), \eta) \sim u$). We calculate:

$$\begin{array}{ll}
 & (\tau_{\mathcal{T}}\ h)[\![f(t_0)]\!]\eta \sqsubseteq I(val\ u) \\
\Leftrightarrow & \quad \{\text{definition of } \tau_{\mathcal{T}}\} \\
 & (lookupfct\ f\ \eta)(h[\![t_0]\!]\eta) \sqsubseteq I(val\ u)
\end{array}$$

If $f\ in\ \eta \equiv ff$, then the last line is trivially true.

If $f\ in\ \eta \equiv tt$, we calculate:

$$\begin{array}{ll}
\Leftarrow & \quad \{(t_0, \eta) \sim (t_0\ \sigma)\text{; } WF(\eta, \sigma, f)\text{;} \\
 & \quad \text{induction hypothesis}\} \\
 & I(val((\sigma\ f)(t_0\ \sigma))) \sqsubseteq I(val\ u) \\
\Leftrightarrow & \quad \{\text{substitution; choice of } \sigma \text{ implies } (f(t_0))\sigma \equiv u\} \\
 & I(val((f(t_0))\sigma)) \sqsubseteq I(val((f(t_0))\sigma)) \\
\Leftrightarrow & true
\end{array}$$

- If $t \equiv if\ t_0\ then\ t_1\ else\ t_2\ fi$ (with $t_0, t_1, t_2 \not\equiv \bot$), let $\sigma$ be a substitution belonging to $(if\ t_0\ then\ t_1\ else\ t_2\ fi, \eta)$ and $u$ (its existence follows from $(if\ t_0\ then\ t_1\ else\ t_2\ fi, \eta) \sim u$). We calculate:

$$(\tau_{\mathcal{T}}\ h)[\![if\ t_0\ then\ t_1\ else\ t_2\ fi]\!]\eta \sqsubseteq I(val\, u)$$

$$\begin{array}{ll}
\Leftrightarrow & \{\text{definition of } \tau_{\mathcal{T}}\text{; choice of } \sigma\} \\
 & \textbf{if } h[\![t_0]\!]\eta = \mathit{true} \textbf{ then } h[\![t_1]\!]\eta \\
 & \qquad\qquad\qquad\qquad \textbf{else } \textbf{if } h[\![t_0]\!]\eta = \mathit{false} \\
 & \qquad\qquad\qquad\qquad\qquad\; \textbf{then } h[\![t_2]\!]\eta \\
 & \qquad\qquad\qquad\qquad\qquad\; \textbf{else } \perp \textbf{ fi } \textbf{ fi} \\
 & \sqsubseteq I(\mathit{val}(\mathit{if}\; t_0 \;\mathit{then}\; t_1 \;\mathit{else}\; t_2 \;\mathit{fi}\; \sigma)) \\
\Leftarrow & \{\text{induction hypothesis}\} \\
 & \textbf{if } I(\mathit{val}(t_0\, \sigma)) = \mathit{true} \textbf{ then } I(\mathit{val}(t_1\, \sigma)) \\
 & \qquad\qquad\qquad\qquad\qquad \textbf{else } \textbf{if } I(\mathit{val}(t_0\, \sigma)) = \mathit{false} \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \textbf{then } I(\mathit{val}(t_2\, \sigma)) \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } \perp \textbf{ fi } \textbf{ fi} \\
 & \sqsubseteq I(\mathit{val}(\mathit{if}\; t_0 \;\mathit{then}\; t_1 \;\mathit{else}\; t_2 \;\mathit{fi}\; \sigma))
\end{array}$$

The last inequation can immediately be refined by conjunction of the following two specifications:

$$\begin{array}{ll}
\textbf{(True)} & \forall t_0, t_1, t_2 \in \mathit{Term} : \\
 & t_0 \not\equiv \perp \wedge t_1 \not\equiv \perp \wedge t_2 \not\equiv \perp \wedge \\
 & CT(t_0) \wedge CT(t_1) \wedge CT(t_2) \wedge \\
 & (I(\mathit{val}\; t_0) = \mathit{true}) \equiv \mathit{tt} \Rightarrow \\
 & I(\mathit{val}\; t_1) \sqsubseteq I(\mathit{val}(\mathit{if}\; t_0 \;\mathit{then}\; t_1 \;\mathit{else}\; t_2 \;\mathit{fi}))
\end{array}$$

$$\begin{array}{ll}
\textbf{(False)} & \forall t_0, t_1, t_2 \in \mathit{Term} : \\
 & t_0 \not\equiv \perp \wedge t_1 \not\equiv \perp \wedge t_2 \not\equiv \perp \wedge \\
 & CT(t_0) \wedge CT(t_1) \wedge CT(t_2) \wedge \\
 & (I(\mathit{val}\; t_0) = \mathit{false}) \equiv \mathit{tt} \Rightarrow \\
 & I(\mathit{val}\; t_2) \sqsubseteq I(\mathit{val}(\mathit{if}\; t_0 \;\mathit{then}\; t_1 \;\mathit{else}\; t_2 \;\mathit{fi}))
\end{array}$$

- If $t \equiv (f(x) : t_0)t_1$, let $\sigma$ be a substitution belonging to $((f(x) : t_0)t_1, \eta)$ and $u$ (its existence follows from $((f(x) : t_0)t_1, \eta) \sim u$). We calculate:

$$\begin{array}{ll}
 & (\tau_{\mathcal{T}} h)[\![(f(x) : t_0)t_1]\!]\eta \sqsubseteq I(\mathit{val}\; u) \\
\Leftrightarrow & \{\text{definition of } \tau_{\mathcal{T}}\text{; choice of } \sigma\} \\
 & (\textbf{rec } r.\; \lambda d.\; h[\![t_0]\!](\eta[d/x, r/f]))(h[\![t_1]\!]\eta) \sqsubseteq
\end{array}$$

$$\begin{array}{ll} & I(val((f(x):t_0)t_1\sigma)) \\ \Leftrightarrow & \quad \{\text{property of substitutions}\} \\ & (\mathbf{rec}\ r.\ \lambda d.\ h[\![t_0]\!](\eta[d/x,r/f]))(h[\![t_1]\!]\eta) \sqsubseteq \\ & I(val((f(x):(t_0(\sigma[x/x,f/f])))(t_1\sigma))) \end{array}$$

Application of the induction hypothesis seems difficult, because without any knowledge of $r$ a substitution $\rho$ must be found such that $(t_0, \eta[d/x, r/f]) \sim t_0\ \rho$. Therefore we analyse fixed point induction on **rec** $r$. In the inductive step a property of the form $\forall d : r\ d \sqsubseteq \ldots$ is needed, where $r$ is the induction variable; but the induction hypothesis states only $r(h[\![t_1]\!]\eta) \sqsubseteq \ldots$. Therefore we generalize the last inequation in the following way:

$$\begin{array}{l} \forall d \in D, u \in Term : CT(u) \wedge d \sqsubseteq I(val\ u) \Rightarrow \\ (\mathbf{rec}\ r.\ \lambda d.\ h[\![t_0]\!](\eta[d/x,r/f]))d \sqsubseteq \\ I(val((f(x):(t_0(\sigma[x/x,f/f])))u)) \end{array}$$

Induction hypothesis: Let $r$ be such that

$$\begin{array}{l} \forall d \in D, u \in Term : CT(u) \wedge d \sqsubseteq I(val\ u) \Rightarrow \\ r\ d \sqsubseteq I(val((f(x):(t_0(\sigma[x/x,f/f])))u)) \end{array}$$

holds.

Let $d$ and $u$ be arbitrary with $CT(u)$ and $d \sqsubseteq I(val\ u)$.

$$\begin{array}{ll} & (\lambda d.\ h[\![t_0]\!](\eta[d/x,r/f]))d \sqsubseteq \\ & I(val((f(x):(t_0(\sigma[x/x,f/f])))u)) \\ \Leftrightarrow & \quad \{\lambda\} \\ & h[\![t_0]\!](\eta[d/x,r/f]) \sqsubseteq \\ & I(val((f(x):(t_0(\sigma[x/x,f/f])))u)) \end{array}$$

In order to apply the induction hypothesis about $h$, we need a substitution $\rho$ so that

$$(t_0, \eta[d/x, r/f]) \sim t_0\ \rho\ .$$

Our choice of $\sigma$, the predicate $WV$, and the premise about $d$ and $u$ suggest to take

$$\rho \equiv \sigma[u/x, \Delta/f]\ ,$$

where $\Delta$ is a still to be determined function declaration. In order to fulfill the predicate $WF$, we must choose $\Delta$ such that

$$\begin{aligned} & \forall c \in D, s \in Term : \\ & CT(s) \wedge c \sqsubseteq I(val\ s) \Rightarrow r\ c \sqsubseteq I(val(\Delta\ s)) \end{aligned}$$

holds. The induction hypothesis on $r$ immediately suggests

$$\Delta \equiv f(x) : (t_0(\sigma[x/x, f/f])) \ .$$

Now we can continue our calculation:

$$\begin{aligned} \Leftarrow \quad & \{\text{induction hypothesis on } h\} \\ & I(val(t_0\,\rho)) \sqsubseteq \\ & I(val((f(x) : (t_0(\sigma[x/x, f/f])))u)) \\ \Leftarrow \quad & \{\text{definition of } \rho\} \\ & I(val(t_0\,\sigma[u/x, f(x) : (t_0(\sigma[x/x, f/f]))/f])) \sqsubseteq \\ & I(val((f(x) : (t_0(\sigma[x/x, f/f])))u)) \end{aligned}$$

The last inequation is implied by the following formula:

$$\begin{aligned} \textbf{(Rec)} \quad & \forall t, u \in Term, x \in X, f \in U : \\ & CT((f(x) : t)u) \Rightarrow \\ & I(val(t[u/x, (f(x) : t)/f])) \sqsubseteq I(val((f(x) : t)u)) \end{aligned}$$

**(End of fixed point analysis, and fixed point induction.)**

Now we refine the specifications obtained above by applying our development method recursively to them.

**Refinement of (P).** We recursively apply our development method to **(P)**, refining it by fixed point induction on $val$. The base case holds because both $I$ and $I(p)$ are strict. Now we turn to the inductive step. Let $v$ be arbitrary so that the induction hypothesis

$$\forall t : CT(t) \Rightarrow (I\ p)(I(v\ t)) \sqsubseteq I(val(p(t)))$$

holds. Let $t$ be arbitrary so that $CT(t)$ holds. We calculate:

$$\begin{array}{ll} & (I\ p)(I(\tau_{val}\ v\ t)) \sqsubseteq I(val(p(t))) \\ \Leftrightarrow & \quad \{\text{definition of } \tau_{val}\} \\ & (I\ p)(I(\mathbf{if}\ normal\,t\ \mathbf{then}\ t\ \mathbf{else}\ v(reduce\,t)\ \mathbf{fi})) \sqsubseteq \\ & I(val(p(t))) \end{array}$$

We make a case distinction on $normal\,t$:

- If $normal\,t \equiv \bot$, then the last line holds trivially.

- If $normal\,t \equiv tt$, then the last line is equivalent to

  $$(I\ p)(I\ t) \sqsubseteq I(val(p(t)))\ .$$

  From $normal\,t \equiv tt$ follows $normal(p(t)) \not\equiv \bot$.

  If $normal(p(t)) \equiv tt$, we continue our calculation:

  $$\begin{array}{ll} \Leftrightarrow & \quad \{\text{definition of } val\} \\ & (I\ p)(I\ t) \sqsubseteq I(p(t)) \\ \Leftrightarrow & \quad \{\text{definition of } I\} \\ & true \end{array}$$

  If $normal(p(t)) \equiv ff$, we continue our calculation:

  $$\begin{array}{ll} \Leftrightarrow & \quad \{\text{definition of } val\} \\ & (I\ p)(I\ t) \sqsubseteq I(val(reduce(p(t)))) \end{array}$$

  Since $normal\,t \equiv tt$, we know that $p(t) \in W$ holds. The operational basis already provides an evaluator $eval$ for terms of the word algebra. According to the axioms, $eval$ reduces terms of the word algebra to normal forms such that their semantics is preserved. Therefore we define $reduce$ to be this evaluator in the current case:

  $$\textbf{(PWRed)} \qquad p(t) \in W \Rightarrow reduce(p(t)) \equiv eval(p(t))$$

- If $normal\, t \equiv ff$, we calculate:

$$
\begin{array}{ll}
 & (I\ p)(I(v(reduce\ t))) \sqsubseteq I(val(p(t))) \\
\Leftrightarrow & \quad \{normal\, t \equiv ff \Rightarrow normal(p(t)) \equiv ff; \\
 & \quad \text{definition of } val\} \\
 & (I\ p)(I(v(reduce\ t))) \sqsubseteq I(val(reduce(p(t))))
\end{array}
$$

In order to be able to apply the induction hypothesis, we state a new requirement on *reduce*:

$$
\textbf{(CT)} \quad \forall t : CT(t) \Rightarrow CT(reduce\ t)
$$

Now we can continue our calculation:

$$
\begin{array}{ll}
\Leftarrow & \quad \{\text{induction hypothesis}\} \\
 & (I(val(p(reduce\ t)))) \sqsubseteq I(val(reduce(p(t))))
\end{array}
$$

A comparison of the two sides immediately suggests the following specification:

$$
\textbf{(PN)} \qquad normal\, t \equiv ff \Rightarrow reduce(p(t)) \equiv p(reduce\ t)
$$

**(End of refinement of (P).)**

**Refinement of (True). Fixed point analysis, and fixed point induction.** We do fixed point induction on *val* in the premise of **(True)**. Since the negation of this premise is equivalent to

$$
(I(val\, t_0) = true) \equiv \bot \vee (I(val\, t_0) = true) \equiv false\ ,
$$

the formula

$$
\begin{array}{ll}
\textbf{(IH)} & \forall t_0, t_1, t_2 \in Term : \\
 & t_0 \not\equiv \bot \wedge t_1 \not\equiv \bot \wedge t_2 \not\equiv \bot \wedge \\
 & CT(t_0) \wedge CT(t_1) \wedge CT(t_2) \wedge \\
 & (I(v\ t_0) = true) \equiv tt \Rightarrow \\
 & I(val\, t_1) \sqsubseteq I(val(if\ t_0\ then\ t_1\ else\ t_2\ fi))
\end{array}
$$

is syntactically admissible in $v$.

The base case holds trivially, because $I$ is strict. Assume that the induction hypothesis **(IH)** holds. Let $t_0$, $t_1$ and $t_2$ be arbitrary terms such that the premises of **(IH)** hold, inclusive of

$$(I(\tau_{val}\ v\ t_0) = true) \equiv tt\ .$$

We calculate:

$$\begin{array}{ll} & (I(\tau_{val}\ v\ t_0) = true) \equiv tt \\ \Leftrightarrow & \quad \{\text{definition of } \tau_{val}\} \\ & (I(\textbf{if}\ normal\ t_0\ \textbf{then}\ t_0\ \textbf{else}\ v(reduce\ t_0)\ \textbf{fi}) = true) \\ & \equiv tt \end{array}$$

Let us make a case distinction on $normal\ t_0$.

- Because of the strictness of $I$ and the premise, $normal\ t_0 \equiv \bot$ is impossible.

- If $normal\ t_0 \equiv tt$, the premise reduces to $(I\ t_0 = true) \equiv tt$. The axioms on $I$ imply $t_0 \equiv cst(true)$. Now we refine the conclusion of the inductive step:

$$\begin{array}{ll} & I(val\ t_1) \sqsubseteq I(val(if\ t_0\ then\ t_1\ else\ t_2\ fi)) \\ \Leftrightarrow & \quad \{t_0 \equiv cst(true)\} \\ & I(val\ t_1) \sqsubseteq I(val(if\ cst(true) then\ t_1\ else\ t_2\ fi)) \\ \Leftrightarrow & \quad \{\ normal(if\ cst(true) then\ t_1\ else\ t_2\ fi) \equiv ff\} \\ & I(val\ t_1) \sqsubseteq \\ & I(val(reduce(if\ cst(true) then\ t_1\ else\ t_2\ fi))) \\ \Leftarrow & \quad \{\text{reflexivity of } \sqsubseteq\} \\ & reduce(if\ cst(true) then\ t_1\ else\ t_2\ fi) \equiv t_1 \end{array}$$

- If $normal\ t_0 \equiv ff$, the premise reduces to $(I\ v(reduce\ t_0) = true) \equiv tt$. Now we refine the conclusion of the inductive step:

$$\begin{array}{ll} & I(val\ t_1) \sqsubseteq I(val(if\ t_0\ then\ t_1\ else\ t_2\ fi)) \\ \Leftarrow & \quad \{\text{induction hypothesis, and } \textbf{(CT)}\} \\ & I(val(if\ reduce\ t_0\ then\ t_1\ else\ t_2\ fi)) \sqsubseteq \end{array}$$

$$
\begin{array}{ll}
 & I(val(if\ t_0\ then\ t_1\ else\ t_2\ fi)) \\
\Leftrightarrow & \quad \{\ normal(if\ t_0\ then\ t_1\ else\ t_2\ fi) \equiv ff\} \\
 & I(val(if\ reduce\ t_0\ then\ t_1\ else\ t_2\ fi)) \sqsubseteq \\
 & I(val(reduce(if\ t_0\ then\ t_1\ else\ t_2\ fi))) \\
\Leftarrow & \quad \{\text{reflexivity of } \sqsubseteq\} \\
 & reduce(if\ t_0\ then\ t_1\ else\ t_2\ fi) \equiv \\
 & if\ reduce\ t_0\ then\ t_1\ else\ t_2\ fi
\end{array}
$$

**(End of fixed point analysis, and fixed point induction.)**

**(End of refinement of (True).)**

**Refinement of (False).** The refinement is analogous to the refinement of **(True)**. Similarly we end up with the explicit specification of *reduce*:

$$reduce(if\ cst(false) then\ t_1\ else\ t_2\ fi) \equiv t_2$$

**(End of refinement of (False).)**

**Refinement of (Rec).** Let $t$, $u$, $x$ and $f$ be arbitrary with $CT((f(x) : t)u)$. Evaluation of $val(t[u/x, (f(x) : t)/f])$ leads to a case distinction on $normal(t[u/x, (f(x) : t)/f])$:

If $normal(t[u/x, (f(x) : t)/f]) \equiv tt$, then:

$$
\begin{array}{ll}
\Leftrightarrow & \quad \{\text{definition of } val;\ normal((f(x) : t)u) \equiv ff\} \\
 & I(t[u/x, (f(x) : t)/f]) \sqsubseteq I(val(reduce((f(x) : t)u))) \\
\Leftarrow & \quad \{\text{premise of this case}\} \\
 & reduce((f(x) : t)u) \equiv t[u/x, (f(x) : t)/f]
\end{array}
$$

If $normal(t[u/x, (f(x) : t)/f]) \equiv tt$, then:

$$
\begin{array}{ll}
\Leftrightarrow & \quad \{\text{definition of } val;\ normal((f(x) : t)u) \equiv ff\} \\
 & I(val(t[u/x, (f(x) : t)/f])) \sqsubseteq I(val(reduce((f(x) : t)u))) \\
\Leftrightarrow & \quad \{\text{definition of } reduce((f(x) : t)u)\} \\
 & I(val(t[u/x, (f(x) : t)/f])) \sqsubseteq I(val(t[u/x, (f(x) : t)/f])) \\
\Leftrightarrow & \\
 & true
\end{array}
$$

**(End of refinement of (Rec).)**

Now we are ready to give a recursive definition of *reduce* by application of the fixed point rule.

```
$reduce$ $\equiv$ **rec** $red.\ \lambda t.$
          **if** $W\ t$ **then** $eval\ t$
          **else** **if** $t = p(u)$ **then** $p(red\ u)$
               **else** **if** $t = if\ t_0\ then\ t_1\ else\ t_2$
                    **then** **if** $t_0 = cst(true)$ **then** $t_1$
                         **else** **if** $t_0 = cst(false)$ **then** $t_2$
                         **else** $if\ red\ t_0\ then\ t_1\ else\ t_2$ **fi**
                    **else** **if** $t = (f(x) : s)u$
                    **then** $s[u/x, (f(x) : s)/f]$
```

So we have developed an operational semantics. In particular, we did not use the knowledge that function declarations can be unfolded. The unfolding

$$reduce((f(x) : t)u) \equiv t[u/x, (f(x) : t)/f]$$

entered the development merely by calculation.

# Chapter 4

# A proof method for recursion

In this chapter we describe the development method that we have used in the examples of chapter 3.

The development task is as follows: A program for an unknown $f$ must be developed from a specification, which contains an inequation or an equation on $f$. This inequation or equation contains recursive definitions.

This class of specifications is practically important, as we have seen from the examples: into this class fall, for instance, the transformation of a nested recursion into a tail-recursive one, and compiler development.

As stated in the introduction, we want a development method that allows systematic development of programs from the mentioned specifications. We do not want developments that are found by "eureka". Moreover, development steps should be oriented at the shape of formulae whenever possible. Knowledge of the problem domain should enter development only when necessary.

The formal foundation, on which our method is based, has been provided in chapter 2. It is a predicate calculus with two rules for recursion: fixed point induction, and the fixed point rule. In particular, we have an unrestricted generalization rule to our disposal.

We start with a description of the viewpoint of program development, which we take in our development method for recursion. Then we turn to the particular development task described above. Finally, we discuss methodological reasons for which we have excluded certain rules for recursion from the calculus.

## 4.0 General development method

We develop programs from specifications in a predicate calculus with the two additional rules for recursion. We do so by equivalence transformations and by strengthening formulae. In this way, a specification of an unknown $f$ is refined until an explicit equation $f \equiv t$ is reached, where $t$ is a term. Such an equation is a program for $f$, which by construction implies the original specification of $f$.

Hence we develop programs by backwards proof. Instead of ending with "true" as proofs do, developments end with a program for the unknown.

As usual in developments, the consistency problem arises: strengthening a specification of $f$ may lead to a specification to which no $f$ exists that meets this strengthened specification. Therefore great care is needed in strengthening specifications.

## 4.1 Development from inequations

In this section we describe how programs are developed from specifications that contain inequations.

**Development task.** Let a formula $S$ be given as a specification of $f$. Assume that an inequation

$$t \sqsubseteq u$$

is a subformula of $S$, and that $f$ occurs in $u$ (and possibly somewhere else in $t$ and $S$). The task is to develop a program for $f$ that meets specification $S$.

**(End "Development task".)**

Our development method starts with fixed point analysis.

### 4.1.0 Fixed point analysis

During fixed point analysis one searches for a refinement by fixed point induction. Fixed point induction is planned, but not yet performed during fixed point analysis.

The description of fixed point analysis is devided into two parts: the analysis process, and the interpretation of its results.

**Analysis.** Let $\mathbf{rec}\ x.\, r$ be a subterm of $t$ such that $\mathbf{rec}\ x.\, r$ is not allowed to appear in the final program. It is analysed where fixed point induction on $\mathbf{rec}\ x.\, r$ leads.

**Choice of induction formula.** The first step consists in finding an induction formula $A$ such that $A[\mathbf{rec}\ x.\, r/x]$ is syntactically identical with $S$.

The formula $A$ must be carefully chosen if a free variable $y$ of $r$ is universally quantified in $S$. Note that in this case $A$ is not just $S$ with $x$ at the position of $\mathbf{rec}\ x.\, r$: then the substitution $A[\mathbf{rec}\ x.\, r/x]$ would not lead to $S$ since the bound variable $y$ would be renamed before the substitution is done. Therefore, typically, the generalization rule must be applied to $S$ in order to remove the explicit universal quantification of $y$. (Note that we apply rules backwards.) Thus $y$ becomes a free variable of the new specification. Then fixed point analysis starts by choosing $A$ for the new specification.

But the generalization rule must only be applied if necessary. As many explicit universal quantifications as possible should remain in the formula: the more explicit universal quantifications remain in the formula, the stronger the induction hypothesis is.

If the same term $\mathbf{rec}\ x.\, r$ occurs more than once in $t$, one should first concentrate on a single one of its occurrences. That is, only one occurrence is substituted, and hence only fixed point induction on this single occurrence is analysed.

In addition, one must ensure that the chosen formula $A$ is syntactically admissible in $x$.

**(End "Choice of induction formula".)**

Having found a formula $A$, one proceeds with the base case, and with the inductive step of the fixed point induction rule as follows:

- Fixed point analysis is recursively applied to the base case. (Note that, although one recursion has been removed, the base case may still contain other recursion operators.)
- For the inductive step it is analysed, whether the induction hypothesis is fully applicable (after equivalence transformations). Depending on the result, one proceeds as follows:
  - If the induction hypothesis is fully applicable, it is not yet applied, but fixed point analysis is recursively applied to the conclusion of the inductive step. Recursion operators that would be removed by application of the induction hypothesis, however, need not be considered in the subsequent fixed point analysis.
  - If the induction hypothesis is not fully applicable, the recursion under consideration is left in the term, and fixed point analysis is applied to a different unwanted recursion term on the left hand side of the inequation.

**(End "Analysis".)**

**Results.** The analysis process may lead to the following results:

- A sequence of nested fixed point inductions has been found so that no unwanted recursion remains on the left hand side of the inequation, that is, all induction hypotheses are fully applicable. The further proceeding depends on the form of the left hand side that arises from application of the induction hypotheses:
  - If application of the induction hypotheses takes one closer to a non-formalized goal (e.g. tail-recursive form), then this sequence of fixed point inductions is selected for the development, but not yet performed. The development is continued with context reduction (section 4.1.1).

  - If the form contradicts such an additional informal requirement, an auxiliary function is introduced (section 4.1.2).

- An unwanted recursion remains on the left hand side, that is, an induction hypothesis is not fully applicable. Then these recursions are left in the term. For the successful fixed point inductions one proceeds as before:

- If application of the induction hypotheses takes one closer to a non-formalized goal (e.g. tail-recursive form), then this sequence of fixed point inductions is selected for the development, but not yet performed. The development is continued with a design decision (section 4.1.3).

- If the form contradicts such an additional informal requirement, an auxiliary function is introduced (section 4.1.2) or a design decision is made (section 4.1.3).

The results have been listed from the most convenient one to the least convenient. So, if fixed point analysis offers several sequences of fixed point inductions, then those are preferred that correspond to a situation mentioned earlier in the list.

**(End "Results".)**

Depending on the result of fixed point analysis, one proceeds with one of the next steps:

- context reduction

- auxiliary function

- design decision

- fixed point induction

### 4.1.1 Context reduction

If the unknown $f$ in the right hand side of the inequation is surrounded by context, one should at next try to remove as much of this context as possible.

A means to do so is to introduce the same context on the left hand side of the inequation, and then to remove (parts of) this context by the monotonicity argument.

After reducing the context, one must re-analyse whether the fixed point induction that has been selected during fixed point analysis is still possible. In general, this will not be the case. Therefore one must generalize the context-reduced specification so that the original fixed point induction works again. That is, one must make a design decision, however, with a particular fixed point induction in mind; one tries to generalize the inequation in order to make this fixed point induction work.

At first sight, it might seem unreasonable to do fixed point analysis first, and context reduction thereafter. The reason for this order is as follows: It is easier to find a working fixed point induction for the original inequation than having to find a suitable generalization at the same time. Having a special fixed point induction in mind helps to find the generalization.

So, if the context of the unknown could be reduced, but the original fixed point induction has become impossible, one proceeds with a design decision. In all other cases one proceeds with fixed point induction.

### 4.1.2 Auxiliary function

One has found a working fixed point induction, but its application would lead to a formula that does not agree with additional properties required of the program for the unknown $f$.

In such a situation one abstracts the term that prevented one from applying fixed point induction. This abstraction is achieved by introduction of one or more auxiliary functions.

The auxiliary function has an argument that is intended to represent the induction variable. When developing the body of the auxiliary function, one should ask: “Why did the shape of the term prevent application of the induction hypothesis?” Then the identified shape is built into the body of the auxiliary function. Hence the definition of an auxiliary function is guided by the shape of the previous formula.

The auxiliary function is used in a specification of a new unknown $g$. This specification is an inequation or equation.

Based on the specification of the new unknown $g$, one must express the old unknown $f$ in terms of $g$. In order to do so, one refines the specification of $f$ by making use of the new specification of $g$ until a term for $f$ is found; this term will contain $g$.

Then the whole development method is recursively applied to the new specification of the new unknown $g$.

### 4.1.3 Design decision

One has found a sequence of successful fixed point inductions (possibly the empty sequence), but in the left hand side of the inequation a recursion is still left, which is not allowed to appear in the final program for $f$.

Then one must make a design decision. The recursion remained in the inequation, because fixed point induction on it failed during fixed point analysis. Inspection of the reason of this failure will help to find a design decision. Of course the new specification must be such that it implies the old one, that is, it must be a refinement of the old specification.

Often a generalization of the inequation is needed, as is well-known from induction proofs. The preceeding fixed point analysis can guide the generalization.

Sometimes a conjunct can be added to the old specification; thus the old fixed point inductions still work, now followed by a fixed point induction on the recursion that could not be eliminated from the inequation before.

After a design decision has been made, the development method is recursively applied to the new specification.

### 4.1.4 Fixed point induction

When all these steps have been carried out (as far as they have been necessary), the planned fixed point induction can be done.

Fixed point induction leads to a number of new specifications of the unknown, which in the next step are turned into a recursive definition.

### 4.1.5 Recursive definition

Typically at the end of the development one has an inequation with $f$ isolated on the right hand side of $\sqsubseteq$, where $f$ may also occur in the term on the left hand side. By the fixed point rule, such a specification can immediately be turned into a recursive definition of $f$.

### 4.1.6 Proofs

Specifications that have not been refined, must be proved for the program developed for $f$. This can be done at the end of the development, or during development: one shows that the actual specification implies the remaining ones. One faces the usual danger of having introduced inconcistencies, as described above.

## 4.2 Development from equations

When a specification contains an equation instead of an inequation, the development task can immediately be reduced to a development from two inequations: the given equation

$$t \equiv u$$

can be rewritten into the conjunction

$$t \sqsubseteq u \wedge u \sqsubseteq t \ .$$

Then one selects the inequation, where the unknown occurs on the right hand side. The development method of section 4.1 is applied to this inequation; the remaining inequation must be proved for the developed program.

Of course, concentration on one inequation can lead to a solution that does not fulfil the second inequation. But this danger of introducing inconsistencies is unavoidable in specification refinement.

It is an intrinsic property of general methods that in special cases developments exist that are shorter and more direct than the development constructed by the method. Assume that in an equation a function $f$ is involved, which is defined by direct structural recursion: the argument of $f$ is of a sort whose elements are generated by a set of constructors, and $f$ is recursively applied only to direct components of its argument. Then a development by structural induction could be shorter than a development by the general method. But a development by fixed point induction on $f$ would only split the development into two parts; for each of the inequations it would precisely reflect the structural induction. Special cases that occur often in practical applications could be identified, and a special submethod could be formulated for them.

## 4.3 Discussion

Now we will discuss the methodological reasons for which we have excluded certain rules from our calculus, which are known from the literature.

**Least (pre-) fixed point rule.** The least fixed point rule

$$t[y/x] \equiv y \Rightarrow \mathbf{rec}\ x.\, t \sqsubseteq y$$

and the least pre-fixed point rule

$$t[y/x] \sqsubseteq y \Rightarrow \mathbf{rec}\ x.\, t \sqsubseteq y$$

are well-known from the literature (e.g. [37], [26]).

Although perfectly fulfilling the criteria for rules of chapter 1, these two rules have been excluded from our calculus. Remember that we want a concise calculus; we can best omit rules that we do not need for methodological reasons.

Assume that a program for $y$ must be developed from the specification

$$\mathbf{rec}\ x.\, t \sqsubseteq y\ .$$

Then, following our development method, we would do fixed point induction on **rec** $x$. The base case is trivially fulfilled, and the inductive step is

$$x \sqsubseteq y \Rightarrow t \sqsubseteq y \ .$$

Application of the induction hypothesis leads to the refined specification

$$t[y/x] \sqsubseteq y \ .$$

This is exactly the specification we would have obtained by application of the least pre-fixed point induction rule. The least fixed point rule would have led to an even stronger specification.

**(End "Least (pre-) fixed point rule".)**

Another well-known technique is transformation by unfolding and folding.

**Unfold/fold-transformations.** The unfold/fold-technique is due to Burstall and Darlington [10]. The unfold-rule and the fold-rule have already been introduced in chapter 2. We repeat them here for convenience:

In our notation we can write the unfold-rule as follows:

$$\mathbf{rec}\ x.\ t[x/y] \equiv \mathbf{rec}\ x.\ (t[t[x/y]/y])$$

The fold rule can be written as follows:

$$\frac{\mathbf{rec}\ x.\ t[u/y] \equiv \mathbf{rec}\ x.\ u}{\mathbf{rec}\ x.\ t[u/y] \sqsupseteq \mathbf{rec}\ x.\ t[x/y]}$$

Both rules fulfil the criteria for rules of chapter 1. We have omitted them from the calculus, because they are not necessary in our development method:

In the literature (e.g. [54], [1]) the unfold/fold-technique is often applied in a rather relaxed way: as is well-known, by folding one can obtain a smaller value (cf. the conclusion of the fold-rule). Speaking operationally, termination may get lost by folding; the transformations establish only partial correctness. But in most proofs and developments in the literature (e.g. [54], [1]) equality $\equiv$ is written instead of $\sqsupseteq$ when the fold-rule is applied. The danger of losing termination is only mentioned. It is also mentioned that therefore termination must be proved; but then no termination proofs are done.

Instead of proving termination after the development, we develop programs from the inequation that implies termination: we develop a program for $f$ from the inequation $t \sqsubseteq f$. This property is sometimes called *robust correctness*. If $t$ has a defined value, then so has $f$. But if $t \equiv \perp$, then $f$ is allowed to take any value whatsoever. Speaking operationally, termination is guaranteed by development; the missing property, which we must prove afterwards, is that $f$ does not terminate more often than $t$. Contrastingly, unfold/fold-transformations develop $f$ from the partial correctness formula $f \sqsubseteq t$. Unfortunately, termination is not proved in most publications that use the unfold/fold-technique.

Another problem with the unfold/fold-technique is that there is no method for its goal-directed application. There do exist a number of strategies (e.g. generalization and tupling strategies) for unfold/fold-transformations; but they rather coexist independently of each other: it is far from obvious, which one of them is best applied to a problem at hand. Moreover, applicability of those strategies is rather restricted, since they were invented for transformation of recursion into linear form.

In addition, unfold/fold-techniques use a lot of operational reasoning: often, some computation traces are computed, and solutions are derived from those examples. We are of the opinion that instead of reasoning operationally, one should derive solutions by application of laws, and by exploration of the shape of formulae. Therefore solutions should neither be derived from examples.

**(End "Unfold/fold-transformations".)**

# Chapter 5

# Conclusion

The present work has tried to make development of recursive definitions from specifications more systematic. We have considered specifications that contain inequations or equations on terms with recursive definitions. For the unknowns of such specifications, recursively defined programs are developed in a refinement style. It has turned out that even in non-trivial examples solutions can to a large extent be calculated.

Let us briefly review the main results of the previous chapters.

We started with program development in general, and searched for criteria for the usefulness of calculi. It turned out that proof design strongly depends on the calculus. Simplicity turned out to be very important at all levels of a calculus: the syntax of terms and formulae must be tailored to manipulation, rather than to manifestation of truths once and for all. Applicability of rules is substantially influenced by their simplicity: applicability of a rule must be easily perceivable. The careful composition of rules into a calculus contributes much to good proof design. A mere aggregation of rules will not be sufficient. And also the interplay of rules has a great impact on proof design: good proof design needs a simple interplay of rules in the calculus.

Then we applied the criteria in order to get a recursion calculus. A recursion operator **rec** was introduced into the language, which can be written into terms. According to the criteria, we preferred the recursion operator to function declarations as known from programming languages. We introduced only two rules for recursion into the calculus: the fixed point induction rule, and the fixed point rule. This choice was confirmed by all our examples. The two rules for recursion were added to a predicate calculus. In developments, we applied the calculus backwards, and, in particular, we applied the fixed point induction rule backwards. We have chosen a predicate calculus since it has simple rules, and a simple proof structure in the sense of chapter 1.

We have been able to apply the same method to all our examples. Even the more difficult examples (such as the nested recursion, the compiler development, and the development of an operational semantics from a denotational one) could be treated systematically, without deep insights into the problem domain. When design decisions were needed, they were largely guided by the shape of formulae. So, wide parts of the developments were calculational.

In the presented method, development starts with an analysis whether the specification is amenable to fixed point induction. The next step in the development depends on the outcome of the analysis: in the best case, fixed point induction can immediately be applied; it leads to a refined specification. If fixed point induction is not immediately applicable or unreasonable, the next step is selected according to the shape of the formula, in which the fixed point induction failed. Although requiring thought, all these steps are guided by the formulae that resulted from fixed point analysis. In our examples, the formulae gave valuable hints for the design decisions. After a design decision has been made, the development method is recursively applied to the refined specification.

We conclude with an outlook to possible future work.

A number of steps in the development process are routine, for instance, generation of formulae during fixed point analysis, and calculation of programs after a suitable fixed point induction has been found. Therefore, the method could well be supported by machine. Thus the programmer could concentrate on those parts that need thought and consideration. Design of such a mechanical support system seems to be straightforward, because the method clearly divides routine steps from those that need thought.

# Bibliography

[1] Arsac, J., Kodratoff, Y.: Some techniques for recursion removal from recursive functions. ACM TOPLAS, Vol. 4, No. 2, April 1982, 295-322

[2] Bauer, F.L., Ehler, H., Horsch, A., Möller, B., Partsch, H., Paukner, O., Pepper, P.: The Munich Project CIP. Vol. II: The Program Transformation System CIP-S. LNCS 292, Springer, Berlin 1987

[3] Bauer, F.L., Wössner, H.: Algorithmische Sprache und Programmentwicklung. Springer 1984, Berlin, Heidelberg, New York, Tokyo, 2. Auflage (in German)

[4] Berghammer, R., Ehler, H., Zierer, H.: Towards an algebraic specification of code generation. Science of Computer Programming 11, 45-63 (1988)

[5] Bird, R.S.: The promotion and accumulation strategies in transformational programming. ACM TOPLAS, Vol. 6, No. 4, October 1984, 487-504

[6] Bjørner, D.: Rigorous development of interpreters and compilers. In: Bjørner, D., Jones, C.B. (eds.) Formal Specification and Software Development. 271-320, Prentice Hall, 1982

[7] Broy, M.: Deductive program development: evaluation in reverse polish notation as an example. In: Broy, M., Wirsing, M. (eds.): Methods of Programming. LNCS 544, Springer, Berlin, 1991

[8] Broy, M.: Experiences with Software Specification and Verification Using LP, the Larch Proof Assistant. Technical Report SRC 93, Digital Systems Research Center, Palo Alto, California, 1992

[9] Broy, M.: Functional specification of time-sensitive communicating systems. In: Broy, M. (ed.) Programming and Mathematical Method. NATO ASI Series F: Computer and Systems Sciences, Vol. 88, 325-367, Springer, Berlin 1993; also: ACM Transactions on Software Engineering and Methodology. Vol. 2, No. 1, January 1993, 1-46

[10] Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. Journal of the ACM, Vol. 24, No. 1, January 1977, 44-67

[11] Burstall, R.M., Landin, P.J.: Programs and their proofs: an algebraic approach. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence 4, Edinburgh University Press, 1969

[12] Cartwright, R.: Recursive programs as definitions in first order logic. SIAM J. Comput., Vol. 13, No. 2, May 1984, 374-408

[13] Chirica, L.M., Martin, D.F.: An approach to compiler correctness. International Conference on Reliable Software, Proceedings, 1975

[14] Choppy, C., Guiho, G., Kaplan, S.: A Lisp compiler for FP language and its proof via algebraic semantics. LNCS 185, Springer, Berlin

[15] Cohn, A.: High level proof in LCF. In: Joyner, W.H. (ed.) 4th Workshop on Automated Deduction, 1979, 73-80

[16] Cohn, A.: The equivalence of two semantic definitions: a case study in LCF. SIAM J. Comput., Vol. 12, No. 2, May 1983, 267-285

[17] Courcelle, B.: Equivalences and transformations of recursive definitions. 26th Annual Symposium on Foundations of Computer Science, 1985, 354-359

[18] Courcelle, B.: Fundamental properties of infinite trees. Theoretical Computer Science 25, 1983, 95-169

[19] Courcelle, B.: Recursive applicative program schemes. In: Leeuwen, J. van (ed.) Handbook of Theoretical Computer Science, 459-492, Elsevier Science Publishers B.V., 1990

[20] Cousot, P., Cousot, R.: Inductive definitions, semantics and abstract interpretation. 19th POPL 1992, 83-94

[21] Dershowitz, N., Jouannaud, J.P.: Rewrite systems. In: Leeuwen, J. van (ed.) Handbook of Theoretical Computer Science, 243-320, Elsevier Science Publishers B.V., 1990

[22] Despeyroux, J.: Proof of translation in natural semantics. Symposium on Logic in Computer Science, Cambridge, Massachusetts, 1986, 193-205

[23] Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs 1976

[24] Dijkstra, E.W.: The unification of three calculi. In: Broy, M. (ed.): Program Design Calculi. NATO ASI Series F: Computer and Systems Sciences, Vol. 118, 197-231, Springer, Berlin 1993

[25] Dybjer, P.: Using domain algebras to prove the correctness of a compiler. In: Mehlhorn, K. (ed.) STACS 85, Proceedings. LNCS 182, 98-108, Springer, Berlin 1985

[26] Dybjer, P., Sander, H.P.: A functional programming approach to the specification and verification of concurrent systems. Formal Aspects of Computing 1, 303-319, 1989

[27] Gardiner, P.H.B., Pandya, P.K.: Reasoning algebraically about recursion. Science of Computer Programming 18, 271-280, 1992

[28] Gasteren, A.J.M. van: On the Shape of Mathematical Arguments. In: Goos, G., Hartmanis, J. (eds.) Lecture Notes in Computer Science, vol. 445, 1990

[29] Gentzen, G.: Untersuchungen über das logische Schließen. Mathematische Zeitschrift 39. I: 176-210, II: 405-431

[30] Gries, D.: Influences (or lack thereof) of formalism in teaching programming and software engineering. In: Dijkstra, E.W. (ed.) Formal Development of Programs and Proofs, 229-236, Addison Wesley 1990

[31] Gries, D.: The Science of Programming. Springer, New York, 1981

[32] Gunter, C.A.: Semantics of Programming Languages. The MIT Press, Cambridge, Massachusetts, London, England, 1992

[33] He, J., Bowen, J.: Compiling specification for ProCoS language $PL_0^R$. Internal ProCoS report OU HJF 6/3, February 1991

[34] Hinkel, U.: Maschineller Beweis der Korrektheit eines Interpreters. M.Sc. Thesis (in German), Technische Universität München, 1992

[35] Hoare, C.A.R.: Algebra and models. In: Broy, M. (ed.): Program Design Calculi. NATO ASI Series F: Computer and Systems Sciences, Vol. 118, 161-195, Springer, Berlin 1993

[36] Hoare, C.A.R.: Mathematics of programming. In: Colburn, T.R. et al. (eds.) Program Verification, 135-154

[37] Hoare, C.A.R., Hayes, I.J., He, J., Morgan, C.C., Roscoe, A.W., Sanders, J.W., Sorenson, I.H., Spivey, J.M., Sufrin, B.A.: Laws of programming. In: Broy, M. (ed.) Programming and Mathematical Method. 95-122, Nato Asi Series F, Vol. 88, Springer 1992

[38] Hoare, C.A.R., He, J.: Refinement algebra proves correctness of compilation. In: Broy, M. (ed.) Programming and Mathematical Method. 245-269, Nato Asi Series F, Vol. 88, Springer 1992

[39] Hußmann, H.: A case study towards algebraic verification of code generation. AMAST91, Iowa, 1991

[40] Kott, L.: A system for proving equivalences of recursive programs. LNCS 87, 63-69

[41] Kott, L.: Unfold/fold program transformations. In: Nivat, M., Reynolds, J.C. (eds.) Algebraic methods in semantics. 1982

[42] Manna, Z.: Mathematical Theory of Computation. McGraw-Hill, New York 1974

[43] Manna, Z., Waldinger, R.: The Logical Basis for Computer Programming. Vol. 2: Deductive Systems. Addison-Wesley 1990

[44] McCarthy, J., Painter, J.: Correctness of a compiler for arithmetic expressions. Proceedings of Symposia in Applied Mathematics, Vol. 19, ed.: J.T. Schwartz

[45] McGowan, C.: An inductive proof technique for interpreter equivalence. In: Rustin, R. (ed.) Formal Semantics of Programming Languages.1972

[46] Mendelson, E.: Introduction to Mathematical Logic. Van Nostrand Company, 2nd edition, 1979

[47] Millo, R.A. de; Lipton, R.J.; Perlis, A.J.: Social processes and proofs of theorems and programs. In: Colburn, T.R. et al. (eds.) Program Verification, 297-319, Kluwer Academic Publishers, 1993

[48] Möller, B.: Higher-order Algebraic Specifications. Habilitationsschrift, Technische Universität München, February 1987

[49] Morris, F.L.: Advice on structuring compilers and proving them correct. POPL73

[50] Morris, J.H.: Another recursion induction principle. Communications of the ACM, Vol. 14, No. 5, May 1971, 351-354

[51] Mosses, P.D.: Denotational Semantics. In: Leeuwen, J. van (ed.) Handbook of Theoretical Computer Science, 575-631 Elsevier Science Publishers B.V., 1990

[52] Nelson, G.: Some generalizations and applications of Dijkstra's guarded commands. In: Broy, M. (ed.) Programming and Mathematical Method. NATO ASI Series F: Computer and Systems Sciences, Vol. 88, 157-191, Springer, Berlin 1993

[53] Paulson, L.C.: Logic and computation - Interactive proof with Cambridge LCF. Cambridge Tracts in Theoretical Computer Science 2, Cambridge University Press, 1987

[54] Pettorossi, A., Proietti, M.: Rules and strategies for program transformation. In: Möller, B., Partsch, H., Schuman, S. (eds.) Formal Program Development. LNCS 755, 263-304, Springer, Berlin 1993

[55] Polak, W.: Compiler Specification and Verification. LNCS 124, Springer 1981

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

[57] Pepper, P.: A simple calculus for program transformation (inclusive of induction), Science of Computer Programming 9, 221-262, 1987

[58] Raoult, J.-C., Vuillemin, J.: Operational and semantic equivalence between recursive programs. Journal of the ACM, Vol. 27, No. 4, October 1980, 772-796

[59] Scherlis, W.L, Scott, D.S.: First steps towards inferential programming. In: Colburn, T.R. et al. (eds.) Program Verification, 99-133, Kluwer Academic Publishers, 1993

[60] Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific J. Math. 5, 285-309, 1955

[61] Thatcher, J.W., Wagner, E.G., Wright, J.B.: More on advice on structuring compilers and proving them correct. Theoretical Computer Science 15, 223-249, 1981

[62] Winskel, G.: The Formal Semantics of Programming Languages - An Introduction. The MIT Press, Cambridge, Massachusetts, 1993

[63] Young, W.D.: A mechanically verified code generator. Journal of Automated Reasoning 5, 493-518, 1989