93

Experiences with Software Specification and Verification Using LP, the Larch Proof Assistant

Manfred Broy

November 12, 1992

Systems Research Center

DEC's business and technology objectives require a strong research program. The Systems Research Center (SRC) and three other research laboratories are committed to filling that need.

SRC began recruiting its first research scientists in l984—their charter, to advance the state of knowledge in all aspects of computer systems research. Our current work includes exploring high-performance personal computing, distributed computing, programming environments, system modelling techniques, specification technology, and tightly-coupled multiprocessors.

Our approach to both hardware and software research is to create and use real systems so that we can investigate their properties fully. Complex systems cannot be evaluated solely in the abstract. Based on this belief, our strategy is to demonstrate the technical and practical feasibility of our ideas by building prototypes and using them as daily tools. The experience we gain is useful in the short term in enabling us to refine our designs, and invaluable in the long term in helping us to advance the state of knowledge about those systems. Most of the major advances in information systems have come through this strategy, including time-sharing, the ArpaNet, and distributed personal computing.

SRC also performs work of a more mathematical flavor which complements our systems research. Some of this work is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. The rest of this work explores new ground motivated by problems that arise in our systems research.

DEC has a strong commitment to communicating the results and experience gained through pursuing these activities. The Company values the improved understanding that comes with exposing and testing our ideas within the research community. SRC will therefore report results in conferences, in professional journals, and in our research report series. We will seek users for our prototype systems among those with whom we have common research interests, and we will encourage collaboration with university researchers.

Robert W. Taylor, Director

Experiences with Software Specification and Verification Using LP, the Larch Proof Assistant

Manfred Broy

November 12, 1992

Manfred H. B. Broy is at the Institut für Informatik, Technische Universität München, Postfach 20 24 20, 8 München 2, Germany

E-mail: broy@informatik.tu-muenchen.de

The author was partially supported by the German Ministry of Research and Technology (BMFT) as part of the compound project "KORSO - Korrekte Software" and by the German Research Community (DFG) project SPECTRUM

Abstract

We sketch a method for deduction-oriented software and system development The method incorporates formal machine-supported specification and verification as activities in software and systems development. We describe experiences in applying this method. These experiences have been gained by using the LP, the Larch proof assistant, as a tool for a number of small and medium size case studies for the formal development of software and systems. LP is used for the verification of the development steps. These case studies include

- quicksort,
- the majority vote problem,
- code generation by a compiler and its correctness,
- an interactive queue and its refinement into a network.

The developments range over levels of requirement specifications, designs and abstract implementations. The main issues are questions of a development method and how to make good use of a formal tool like LP in a goal-directed way within the development. We further discuss of the value of advanced specification techniques, most of which are deliberately not supported by LP and its notation, and their significance in development. Furthermore, we discuss issues of enhancement of a support system like LP and the value and the practicability of using formal techniques such as specification and verification in the development process in practice.

Contents

Chapter 1

Introduction

Achieving reliable, provably correct programs and system designs seems impossible. Formal methods of specification and verification are advocated for doing the impossible. However, so far, in practice, formal methods have been considered too difficult and too costly. Moreover, many advocates of formal methods of program development have dealt only with small examples, often just toy examples. When more complex examples have been treated the complexity has been only in their logical structure, not their size. So there have been (and are) considerable doubts about, whether formal techniques can be of help in practice and if they scale up. Nevertheless, in recent years there has been considerable interest in formal hardware verification. Serious attempts have been undertaken to verify hardware designs (cf. [MacKenzie 91]).

1.1 Significance of Formal Verification

Formal methods are advocated as helpful for improving the reliability of software systems. In the early days of formal methods, the verification of programs (often called "post mortem verification") was one of the main issues. However, it soon became obvious that it is much more appropriate not to wait until a program has been fully developed to verify it, but to start from a given formal specification of the problem and then to construct program and correctness proof step by step[1]. This makes the verification simpler (and in many applications is the only way to manage the construction of the correctness proof at all) and, even more important, directly supports the construction of a correct program. Program transformations have been advocated for a rigorous machine-supported top down approach along these lines (cf. [CIP 84]).

[1]Most programs designed in a conventional way turn out to be incorrect, anyhow, so a verification cannot be successful.

1.2 Significance of Formal Specification

Even the most advanced verification techniques cannot solve the problem of reliability of software as such. The reliability of "correct" (formally verified) programs crucially depends on the adequacy of their formal specifications. Only if the requirements are properly captured formally, can the verification make sense[2]. Clearly, the assumption of a given formal specification is academic. In practical applications, it is a major task in development to capture all requirements and to work out a formal requirement specification from vague and often contradictory informal specifications. Moreover, it is often necessary to change and refine the initial specification during the development process (if tolerable from the application point of view) taking new insights into account and to get more useful, more efficient or more elegant solutions.

So the derivation of requirement specifications plays a crucial role in development. It is necessary to pay more attention to the derivation of an adequate requirement specification. Consequently, *requirement engineering* or *specification engineering* is one of the most important areas of applying formal methods.

The use of formal techniques, including verification techniques, has advantages at the level of requirement engineering. However, it needs a quite different type of development method. And as a consequence it needs quite different support tools.

Many applications of software engineering, especially in technical domains, have elaborate models and theories. This is called *domain specific knowledge*. In some domains this knowledge is formalized. Often, however, the domain specific knowledge is formalized only partially or not at all. This brings additional problems for programming and formal methods. The domain specific knowledge has to be captured by the software engineer and brought into forms such that this knowledge can be used both for a formal specification and for the representation in a program.

In particular, when applying formal methods, it cannot be the task of the software engineer to replay all the theoretical proofs of the specific domain of application. What is needed is the capture of the theories and theorems of the application domain by formalisms that are adequate for the further development and where the validity of this formalization can be checked in a *validation* step as easily as possible. What is needed is a well-defined interface between the world of application and the world of programming. Axiomatic specification can provide such an interface.

1.3 Using LP: The Case Studies

Larch (cf. [Guttag et al. 85]) has been proposed as a family of axiomatic specification languages. Larch interface languages for a variety of programming languages (cf.

[2]So the remark above about the expected incorrectness of programs can be repeated when talking about the validity of specifications: most specifications do not capture the problem adequately, anyhow.

[Garland, Guttag 90], [Guttag, Horning 86a], [Guttag, Horning 86b]) are now available. For the technical details of the LP, the Larch proof assistant, (cf. [Garland, Guttag 91]), an introduction to Larch and how to use it practically we refer to the cited reports. A comprehensive presentation of Larch and LP is given in [Guttag, Horning 93].

The LP is a tool that supports proofs about axiomatic specifications. LP is based on the concept of rewriting. Larch and also LP follow the philosophy to keep everything as simple as possible.

LP does not provide a decision procedure. LP mainly provides a convenient notation and machinery for defining rewrite rules by axiomatic techniques and applying them. All proofs are carried out by applying the rewrite rules, proofs by case splitting, induction, contradiction, and application of rules of inference. Also the logical derivations are carried out by applying rewrite and deduction rules.

The user of LP has to guide the system. LP can find rather simple proofs (which nevertheless can be quite time consuming when carried out by hand) without user interaction. The construction of a proof and the debugging of the specification and the implementation are done hand in hand.

The following four examples of the use of the LP as a tool for requirement engineering and deduction-oriented program development are treated:

- quicksort,
- the majority vote problem,
- a compiler and its correctness,
- an interactive queue and its refinement into a network.

Formal specifications are given for the examples and steps of developments as well as proofs of their correctness. Throughout the examples, the properties and adequacy of the LP as a tool for program development are discussed.

This study has been carried out to also gain further insights what kind of advanced specification languages and support systems are needed when working with rigorously formal methods in program development. The author is currently in charge of a project for developing a deduction oriented program development methods and an advanced specification language for its support. The specification language under development is called SPECTRUM (cf. [SPECTRUM 92]). One purpose of this study is to understand and document where advanced specification concepts and languages like SPECTRUM may have impacts and advantages when using them in a development oriented style.

Chapter 2

What LP Does Not Have

When designing a specification language or a verification support system, there are always conflicting goals. One goal is to keep everything as simple and small as possible. Then things are easy to learn, to understand and to start to use. Another goal is to provide a lot of functionality and a lot of support for specific steps of development and proof. Then powerful support can be given, even in difficult cases of development.

2.1 Basic Philosophy of Larch and LP

Larch and LP are deliberately kept extremely simple. This has a great number of advantages. First of all, Larch and LP are very simple to learn, especially for someone familiar with the basics of algebraic specifications. The LP notation for specification supports sorts, function symbols (even in infix notation), variables and equations. In contrast to classical logics, in LP there are two versions of equality "=" which is the equality test and "==" which is the main operator of an equation. Semantically, there is no difference. Operationally, with respect to the calculus, there is a big difference.

The LP notation is a specification language of the first generation. Of the fancy features, as described below, LP includes only a limited infix notation and function symbol overloading.

Larch and LP are based on the concept of *loose semantics*. Moreover, the logics at the level of axioms and the logics at the level of Booleans are identified. Therefore, although LP (version 2.1) does not have conditional equations in the sense of rewrite rules, it does have full propositional equational logic, since equality as an operation is always available. One is not allowed to write

$$x == y \wedge y == z \Rightarrow x == z$$

However, one may write instead:

$$\big((x = y \wedge y = z) \Rightarrow x = z\big) == true$$

LP does not support, however, the following specification concepts that can be found in a number of advanced specification languages.

2.2 Advanced Specification Concepts

The specification and design language SPECTRUM is under development at the Technical University of Munich (see [SPECTRUM 91]).

Like LP, SPECTRUM is based on the concept of loose semantics. This means that specifications are seen as systems of signatures and axioms. Every algebra of proper signature that fulfills the axioms is accepted as model. No general constraints like initiality or terminality is preimposed. We consider loose semantics as more appropriate for development and refinement oriented specification techniques. In loose semantics one may refine specifications by adding properties in terms of additional axioms to specifications with the effect that the class of models of the refined specification is always included in the class of models of the original one. In terms of logic the proof systems are monotone: adding sorts, function symbols, and properties never invalidates logical properties of the given specifications. This is not true for initial or terminal semantics. Induction techniques are available in both in LP and in SPECTRUM only if they are explicitly stated by special axioms.

SPECTRUM is intended to support in addition to the concepts of the LP specification formalism the following features:

- a logic of *partial functions and definedness*: often in software specifications partial operations occur. In Larch and LP it is suggested to handle these partial functions by total functions and by underspecification. This means that the value of a function is deliberately left unspecified for some arguments. However, due to the totality assumption it is known that the function has a regular value for those arguments. This assumption does not work, however, in connection with full recursion (compare the compiler example and the example of the interactive queue). Therefore SPECTRUM includes a full logic of partial functions and definedness.
- *higher order elements*: often it is helpful to reason about functions as elements (compare the example of compiler correctness). This gives a lot of expressive power. Therefore SPECTRUM supports functional sorts and λ-notation. However, the carrier sets associated with functional sorts are not, in general, the full function spaces but just subsets of the classical mathematical function spaces.
- *fixed point theory* and *infinite objects*: often it is useful to define elements by recursion which technically means as fixed points of monotonic or even

continuous functions. This leads to "infinite" objects and fixed point reasoning (compare the examples of compiler correctness, and of interactive queues). Such concepts are not explicitly supported by LP, but are in SPECTRUM.

- full *predicate logic of quantifiers*: for abstract specifications quantifiers are extremely helpful. Also the logical reasoning often gets more transparent when quantifiers are used. SPECTRUM supports full first-order predicate logic with arbitrary nesting of existential and universal quantifiers. This extension is planned for LP.

- *polymorphism*: when dealing with general concepts of specifications like sequences it is convenient to allow to write $Seq\ X$ for arbitrary sorts X (so that one can write $Seq\ Nat$ as well as $Seq\ Char$ and even $Seq\ Seq\ Nat$) and to use always the same function names with polymorphic axioms such as ($first(x@s) = x$ for all x of arbitrary sort) for manipulating these sequences. This makes specifications shorter and more readable (see [Cardelli, Wegner 85], [Cardelli 87]). In LP overloading is supported, which allows to use the same function names for different sorts, but for sequences over different sorts always new sort names have to be invented and axioms (and proofs) have to be repeated (see the example of compiler correctness where sequences are used for representing labels, programs, stacks etc.). SPECTRUM supports full *ad-hoc-polymorphism*[1] (a polymorphic form of overloading), which includes also parametric polymorphism.

- *inheritance* and *subsorting*: there are situations, where sorts are conveniently defined as subsorts of other sorts (let Nat be a subsort of Int) and the operations are inherited. LP does not support subsorting nor inheritance. In the examples, neither subsorting nor inheritance were seen to be extremely helpful[2]. However, subsorting and inheritance may show their full power in large scale commercial applications using object-oriented techniques. The current version of SPECTRUM includes inheritance and subsorting in a restricted form, although we have some doubts about their significance.

- advanced *combining forms* for specifications as a concept for specifying in the large: for specifying larger systems appropriate concepts for composing specifications are needed. LP does not support any forms of composing specifications. Therefore, it is fair to say that LP supports only reasoning in the small. This does not apply to Larch, where a number of concepts for composing specifications are included. SPECTRUM provides a number of operators for composing specifications. An important issue is a logical framework for reasoning about composed specifications (called *reasoning in the large*). This is also a goal of SPECTRUM (cf. [Wirsing 91]). In the examples treated in this report, reasoning in the large

[1] It is not so clear what *parametric polymorphism* looks like at the specification level where it is not possible to distinguish between a defining and an applied occurrence of a function symbol.

[2] In the example of the interactive queue, it might be helpful to see the set of data messages as a subsort of the set of messages including the request signal.

was hardly needed, since the examples are relatively small. Only in the cases of compiler correctness, support for reasoning in the large might have been of some convenience.

- *parameterization*: parameterized specifications are considered an important topic by many researchers in axiomatic specification techniques. Nevertheless, explicitly parameterized specifications can be completely replaced by adequate concepts for composing specifications. Consequently, neither Larch nor LP nor SPECTRUM offer explicit parameterization for specifications.

All the listed concepts are believed to be helpful for certain specification tasks especially when dealing with large and complicated systems. However, they also make the language and the design calculus more complicated and more difficult to learn and to use.

It is the purpose of this study to understand better the implications of the simplicity of Larch and LP and where certain concepts that are not available turn out actually to be painfully missing in the development process. The treated examples were chosen especially to study aspects of development that are related to this question.

Chapter 3

Method

Formal techniques are not easy to understand, to learn, and to use. To be able to apply particular formal techniques, one has to understand the basic theory behind them. Furthermore, one has to learn how to apply the rules of the formal technique correctly. Finally, one has to understand how to apply a couple of rules to achieve a particular goal. Structuring the work towards an overall goal into subgoals and organizing the work in steps to achieve these subgoals is what a *method* should provide.

3.1 Avoiding Methodological Strait-Jackets

A method is not supposed to make life harder, although, unfortunately, many methods proposed in the area of formal techniques do make it harder.

Numerous beautiful formal derivations and verifications for impressively complex examples have been published as milestones of systematic software design and applying formal techniques. Quite clearly, the authors did not find the elegant derivations they finally presented in their papers in their first attempts. In most cases, the authors worked very hard to find finally the elegant solution, and its nice presentation as published. They were surely messing around, trying this and that, following dead ends, writing down specifications or assumptions that were inconsistent or otherwise flawed. This is not bad. It is the way solutions in science and technology are generally found and explored. Writing down an incorrect hypothesis or giving an inconsistent set of axioms is not harmful – if careful analysis then finds the flaws.

Of course, it would be inadequate (and sometimes offensive) to publish all the failing and incorrect attempts in a development in addition to the final solution. However, it is not very honest to present the final polished version of a derivation as if it had been found in a straightforward top down manner right away. And it can be harmful to advocate that development should be done just in the way the derivation was presented.

Every innocent victim that tries to use such a method will come to the conclusion that he/she is a fool or, more rightly, that the author cheated. It is not surprising that software engineers often get frustrated in this way when trying out formal techniques in practice.

In the following a methodological framework is introduced, very much along the lines of experiences that I gained in using formal techniques on small to medium size case studies. Observing my own proceeding when tackling a problem I came to the conclusion that appropriate methods should leave room for freedom and flexibility in experimenting with solutions and trying out ideas, but should also help to organize the work such that the rigor of formal techniques pays and that after a successful ending of the development there remain no doubts about its correctness and that finally a proved solution is established.

Of course, a method cannot guarantee to guide the developer to a solution[1]. However, methods should support the developer in his/her task such that the validity of the development is assured after all steps of a development have been carried out successfully following the method.

3.2 A Method for Deduction Oriented Design

In deduction oriented program design the phases of requirement capture, design, verification, and the construction of an implementation are integrated and supported by the goal-directed use of formal techniques. All these activities are merged in an appropriate goal directed proceeding. Such a method is sketched in the following.

We use well established *algebraic* or more general *axiomatic* techniques for representing specifications. We use the usual notion of a signature, that is a family of sorts (names for carrier sets) and a family of function symbols (names for functions). Furthermore, in specifications we write formulas for a given signature. We use a logical calculus that allows to deduce (prove) formulas from given ones. As usual in logics for sets of formulas Δ and Θ:

$$\Delta \vdash \Theta$$

we write in order to express that the formulas in Θ can be logically deduced from the formulas in Δ by the deductive theory. The deductive theory is assumed to be monotonic, such that from

$$\Delta \vdash \Theta$$

and

$$\Delta \subseteq \tilde{\Delta}$$

we may be conclude

$$\tilde{\Delta} \vdash \Theta$$

This requirement of monotonicity of the deductive theory is fulfilled in LP.

[1] Otherwise, the design could be completely automated.

In principle, we propose a method that works with just three classes of development steps called *enrich*, *verify*, and *revise*. We consider specifications $S = (\Sigma, \Delta, \Theta)$, where

- Σ is a signature, that is a family of sorts (names for carrier sets) and a family of function symbols (names for functions) with given functionalities,
- Δ is a set of formulas (based the signature Σ) called *axioms*,
- Θ is a set of formulas (based the signature Σ) called *theorems*, which can be deduced from the axioms, formally $\Delta \vdash \Theta$. This requirement is to be checked using LP.

We call specifications enjoying these properties *well-formed*. The idea of axioms and theorems is directly supported in Larch by the "**implies**" clause.

We are interested in an incremental method for constructing specifications. Therefore we assume that we start with a specification where the set of theorems is empty and then step by step prove theorems. By proving a logical formula before putting it into the set of theorems we can be sure that the derived specifications are always well-formed.

We use the following three classes of development steps:

- *enrich*: in an enrichment step we freely add sorts, functions, and axioms to the specification under development. Of course, this is dangerous in the sense that we might obtain an inconsistent specification that way. However, we will never transform a well-formed specification into a nonwell-formed specification, since our deductive theory is monotonic.
- *derive*: a derivation is used for proving a theorem. This way we can add a theorem to the set of theorems. A special case is the elimination of an axiom by its derivation from the remaining axioms. In this case we have identified the axiom as being redundant and we can delete it from the axioms and add it to the theorems. In both cases, we turn a well-formed specification into a well-formed specification by a derivation.
- *revise*: often a development will lead to a stage where certain design decisions in the requirement capture and in the design are recognized as inadequate with respect to the requirements or simply as inconsistent. Then, to get rid of these problems in a revision, we set back the development to an earlier version of the development process. This is such a simple idea that in many methods it is not mentioned explicitly. However, if we recognize a decision (an enrich step) as inadequate and take it back only after a long sequence of development steps, it is unacceptable for us to throw away all the work carried out in between. Certainly some of the work has to be thrown away. But large portions of the work done after the inadequate enrichment might be usable ("reused") in the revised development.

In practice, a strong support of revision as a step in development may be the most important prerequisite for the cost effective applicability of formal methods.

We have carried out the case studies given in the following along the lines of a stepwise development method including roughly the following substeps:

- domain theory specification,
- requirement capture and specification,
- design specification,
- implementation.

For some of the case studies, not all steps of this development scheme are carried out actually in the following.

Larch and LP are used in combination with interface specification languages designed for individual programming languages such as C or Modula 3. In contrast, we follow a more purist approach, and also present programs at the level of LP. Intuitively speaking, a program at the level of LP is a specification where all the axioms are of a constructive form, that is, can be interpreted as rewrite rules or recursive declarations.

3.3 How to Ensure Consistency

A specification is called contradictory, if from its axioms every formula can be deduced. In particular, then $true == false$ can be deduced in LP. A contradictory specification does not have a model and is therefore inconsistent. Notice that we do not accept a mathematical structure as a model where $true == false$ is valid. The fundamental problem with the axiomatic approach to program and system development is the problem of ensuring adequacy and proving the consistency of a specification. An academic solution to this problem would be that consistency is proved by giving a specification where all axioms are in a constructive form (for which consistency can be assumed by fixed point arguments). In practical developments (even for toy examples) inconsistent specifications are often written. This is not an unsolvable problem. As soon as inconsistencies are identified in the course of a development a *revision* is carried out by backtracking to a previous (hopefully consistent) version and then taking a different path of enrichment.

Of course, we are only interested in specifications that are not contradictory. Unfortunately, there is no way to prove the consistency of a specification within LP. A proof $true \neq false$ for a specification in LP does not say that the specification is consistent. It might be the case that in addition we can prove $true = false$. So inconsistency, but not consistency can be proved in LP.

This is a severe problem. If auxiliary functions and predicates are used in a proof, then the proof may be valid only because of contradictions in the specifications of the auxiliary functions. As a consequence, a proof in LP using auxiliary functions is, strictly speaking, useless, if we cannot argue that the auxiliary equations of the functions do not introduce inconsistencies.

Let us therefore briefly discuss the consistency of specifications. Clearly, we may assume that a specification without any axioms is consistent. But specifications without axioms are not very interesting. So let us ask, whether there is a way to enrich a consistent specification while maintaining consistency.

One possible way to make sure that consistency is maintained is to consider only axioms of a special form. An equation of the form

$$f(x_1, ..., x_n) = t$$

where $x_1, ..., x_n$ are variables and t is an arbitrary term that does not contain f is called an *explicit equation* for f. If in a specification for a set of functions only one explicit axiom is given for each function in the set, then the functions are called *explicitly specified*.

A way for specifying functions quite similar to explicit specifications are specifications by structural recursion. A function symbol f is said to be specified by *structural recursion*, if f is described by axioms of the form ($1 \leq i \leq m$):

$$C_i \Rightarrow f(E_i) = t_i$$

where the function symbol f does not occur in the terms[2] E_i.

A (consistent) model with given signature Σ is called *fixed point algebra*, if for any functional enrichment by arbitrary systems of function symbols explicitly specified by equations over Σ there exists an extension of that model (without adding elements to sorts) such that the explicit equations are fulfilled.

The models of LP are not fixed point algebras, in general. For enrichments of LP specifications by functions specified by structural recursion the proposition that consistency is maintained requires a proof showing that for all i, j, $1 \leq i \leq m$, $1 \leq j \leq m$ (let x and y be fresh variables):

$$C_i \wedge C_j \wedge x = E_i \wedge y = E_j \Rightarrow t_i = t_j$$

This proof, however, can be quite difficult, if the function symbol f occurs in the terms t_i.

[2] Often, it is required that only so called constructor functions occur in the terms E_i.

3.4 Focus of the Case Studies

Our main interest in the little case studies presented in the following is not so much the detailed description of all the steps and technicalities of the verification of the theorems using LP, but rather the explanation of the overall structure of the developments and proofs.

There are several phases of development that may be, or in the sense of a rigorous method, must be, supported by proofs. The proofs can be structured into the overall proof structure, also called the *proof architecture*, and a number of smaller straightforward proofs of subtheorems. We do not indicate and discuss technically, how we verified the simpler subtheorems in LP.

Our main interest in the following is in discussing the techniques of LP, and where the simplicity of LP forces one sometimes to find time-consuming auxiliary constructions for expressing the required structures.

We also present in the following all the notation exactly as it appears in our proofs in LP and do not give nicely edited formulas using the potentials of LaTeX, since it might be more interesting for the reader to see the formulas as they are seen at the screen of a terminal when working with LP.

Chapter 4

Quicksort

In this chapter we develop a requirement specification and a program for quicksort and verify the program (the algorithm presented in equational form) with respect to the specification. We present both the program and the specification in LP.

4.1 The Domain Theory for Sorting

We start the development by providing a domain theory for sorting. It mainly consists in the theory of linearly ordered sets and of sequences.

We do not give the axioms for natural numbers and sequences here. They can be looked up in the appendix. Just note, $i@s$ denotes the sequence obtained by putting the element i in front of s, $s\hat{\ }r$ denotes the concatenation of the two sequences s and r, $nbr(s)$ denotes the number of elements, also called the length, of sequence s, $i\#s$ denotes the numbers of copies of i in the sequence s, and es denotes the empty sequence.

4.2 The Requirement Specification for Sorting

We start by giving the requirement specification for sorting. We introduce functions $issort$ and $sorted$.

```
dec op   sorted : Seq -> Bool
         issort : Seq, Seq -> Bool
..
assert   issort(s, r) => ( (i#s) = (i#r) & sorted(r))     (1)

         sorted(es)
```

```
          sorted(i@es)
          sorted(i@(j@s)) = ((i =< j) & sorted(j@s))
..
```

Of course we would rather have written in full predicate logic:

$$issort(s, r) = \forall i \in Nat : (i\#s) = (i\#r) \wedge sorted(r)$$

However, because of the restriction of LP to logical expressions without quantification, we are not allowed to specify $issort$ that way in LP. We may write, however, in LP quantification in when-clauses and add the following axiom for the predicate $issorted$:

```
assert  when (forall i) (i#s) = (i#r) & sorted(r)          (2)
        yield issort(s, r)
..
```

This deduction rule alone, however, and the axiom (1) alone, provide only a loose specification of the predicate $issort$. By (1) only constraints for the positive part of $issort$ are given. The assignment $issort(s) = false$ for all s is consistent with (1). By (2) only the constraints for the negative part of $issort$ are given. The assignment $issort(s) = true$ for all s is consistent with (2). The logically strongest predicate[1] that fulfills this axiom is exactly the predicate we are interested in. However, if we are able to prove a property just based on the deduction rule above, the proof is valid for all predicates and therefore also for the strongest one. This is an interesting aspect of loose specifications that deserves further methodological analysis.

If we use both axioms (1) and (2) for $issort$, then $issort$ is uniquely characterized. It is an inconvenience in LP that due to missing quantifier notation we have to write two axioms to define the function $issort$, when it is much more adequately and more readably defined by one axiom.

The quantifier for $issort$ can be avoided by introducing the data structure of bags and by mapping sequences s by a function mb to bags $mb(s)$. Then the specification of $issort$ reads:

$$issort(s, r) = (mb(s) = mb(r) \wedge sorted(s))$$

but again we pay the price of introducing a new specification and an additional function. Also the axiom of the function $sorted$ looks nicer if specified by quantifiers:

$$sorted(s) \equiv \forall s_1, s_2 \in Seq, i, j \in Nat : s = s_1\hat{\ }i\hat{\ }j\hat{\ }s_2 \Rightarrow i \leq j$$

In addition, using these forms of explicit specifications one can be sure not to introduce any inconsistencies.

Based on the introduced predicates the requirement specification of $quicksort$ can be given.

[1]Note, false is stronger than true.

```
dec op  quicksort : Seq -> Seq
```

The basic requirement specification for *quicksort* then reads:

```
assert  issort(s, quicksort(s))                (*)
```

We do not add this requirement specification for the function *quicksort* to the LP specification, although this would be perfectly correct for obtaining a sufficient requirement specification for sorting. We rather go on with the development and give a constructive description of *quicksort* and prove (*).

If higher order concepts were available (which is not the case in LP), we could introduce a predicate:

$$sorter : (Seq \to Seq) \to Bool$$

specified by the axiom:

$$sorter(f) = \forall s \in Seq, i \in Nat : (i\#s) = (i\#f(s)) \land sorted(f(s))$$

Then we can specify *quicksort* by the following formula:

$$sorter(quicksort)$$

The ability to write predicates on functions provides possibilities to express the relationship between requirement and design specifications more explicitly within the logic.

4.3 Design Specification

The next step is to give a more algorithmic ("constructive") description of *quicksort*. For doing this we use two functions *lepart* ("leftpart") and *ripart* ("rightpart") that filter out those elements of a sequence that are less than a given element or not less than a given element respectively. It reads as follows:

```
dec op  lepart, ripart : Nat, Seq -> Seq

assert  quicksort(es) = es
        quicksort(i@s) =
           quicksort(lepart(i, s))^(i@quicksort(ripart(i, s)))
..
```

Again we might have preferred to introduce a predicate on functions:

$$isquick : (Seq \to Seq) \to Bool$$

where

$$isquick(f) = \big(\forall i, s : f(es) = es \wedge f(i@s) = f(lepart(i,s))\hat{\ }(i@f(ripart(i,s)))\big)$$

Then we can express the correctness condition more explicitly by

$$isquick(f) \Rightarrow sorter(f)$$

The equations for *quicksort* as they are given here can be understood to provide an inductive definition of *quicksort*. They can also be seen as recursive definitions as used in a functional program for computing *quicksort*. However, in contrast to the classical semantic interpretation of the semantics of programming languages, where a *least* solution (a least fixed point) is associated with a recursive equation, in the loose semantics of LP any solution can be chosen in a model. That in the case of *quicksort* the solution is uniquely determined (in terms of fixed point theory: there exists just one fixed point for the equation above) may not be obvious at a first sight[2].

4.4 Implementation and Verification

In proceeding top down we might give only nonconstructive specifications for the functions *lepart* and *ripart* such as the following formulas that can be read as the requirement specifications for the functions *lepart* and *ripart*.

```
(i#ripart(j, s)) = if(j =< i, i#s, 0)
(i#lepart(j, s)) = if(i < j, i#s, 0)
```

In a more explorative, and therefore more realistic, scenario the necessity to assert or to prove these propositions about *lepart* and *ripart* during the correctness proof for *quicksort* might only be discovered while attempting to carry out this proof.

As with *quicksort*, we do not add these requirement specifications to our specification, but rather give constructive specifications for the functions *lepart* and *ripart* such as:

```
assert
lepart(i, es) = es
lepart(i, (j@s)) = if( j < i, j@lepart(i, s), lepart(i, s))

ripart(i, es) = es
ripart(i, (j@s)) = if( i =< j, j@ripart(i, s), ripart(i, s))
..
```

[2]In fixed point algebras for functions that are specified just by one equation of the form $f(x) = \ldots$ solutions do always exist, or, in other words, adding functions with just explicit equations to a consistent fixed point algebra does not introduce inconsistencies. This does not hold in LP, but in SPECTRUM. This will be discussed in more detail in the conclusions.

It is a straightforward proof by induction on s in LP to show that the constructive specifications imply the nonconstructive formulas given above for the functions $lepart$ and $ripart$. The nonconstructive formulas are also useful in the correctness proof for $quicksort$.

In the proof of the correctness of quicksort, we need a number of auxiliary lemmas. Of course, the need for these lemmas is again perhaps only discovered when trying to carry out the correctness proof for $quicksort$. Nevertheless, we list the lemmas here separately.

```
when i < j yield (i#ripart(j, s)) = 0
when j =< i yield (i#ripart(j, s)) = (i#s)
when i < j yield (i#lepart(j, s)) = (i#s)
when j =< i  yield (i#lepart(j, s)) = 0

nbr(lepart(j, s)) =< nbr(s)
nbr(ripart(j, s)) =< nbr(s)
```

These lemmas turned out to be not particularly difficult to prove in LP. All require induction on s. The proofs are rather straightforward.

For proving that the result of quicksort is sorted we have to prove that $lepart$ and $ripart$ do a proper split according to the size of the elements in their arguments. For being able to formulate this property we introduce the following auxiliary operators.

```
dec op   <<  : Seq, Nat -> Bool
         =<< : Nat, Seq -> Bool
..
assert   es << i
         ((j@s) << i) = ((j < i) & (s << i))

         i =<< es
         (i =<< (j@s)) = ((i =< j) & (i =<< s))
..
```

We would rather have specified the operators in a more descriptive explicit style by axioms of the form:

$$(s << i) = \forall j : 0 < (j\#s) \Rightarrow j < i$$

This is not possible in LP due to the very restricted ways of using quantifiers. Again the consistency of the explicit specification is obvious, while the implicit specification given above requires a short analysis.

It is straightforward, however, that the consistency of our specification is not destroyed by the axioms used to specify these auxiliary operators, since they are defined inductively on the set of sequences. Based on the introduced operators we prove a couple of further lemmas by induction on s. They read as follows.

```
((i =< j) & (j =<< s)) = ((i =< j) & (i =<< s) & (j =<< s))
sorted(i@s) = ((i =<< s) & sorted(s))

when i =< j yield i =<< ripart(j, s)
when j =< i yield lepart(j, s) << i
when i =< j yield (i =<< lepart(j, s)) = (i =<< s)
when j < i yield lepart(j, s) << i
when j < i yield (ripart(j, s) << i) = (s << i)

((r^s) << k) = ((r << k) & (s << k))

k =<< ripart(k, s)
lepart(k, s) << k

(i =<< (r^s)) = ((i =<< r) & (i =<< s))

sorted(s^(i@r)) = (sorted(s^(i@es)) & sorted(i@r))
(sorted(j@s) & ((j@s) << i)) => sorted((j@s)^(i@es))
(sorted(s) & (s << i)) => sorted(s^(i@es))
```

Having all these lemmas available we have nearly all at hand to complete the proof of the main theorem. This proof consists of two independent parts: it requires the proof of

```
prove   sorted(quicksort(s))
```

as well as of

```
prove   (i#quicksort(s)) = (i#s)
```

We start with the latter part. Unfortunately the proof is far from being straightforward in LP. Certainly, it has to be a proof by induction. However, an induction proof using the fact that sequences are generated by the empty sequence and the operation *append* cannot be used directly. In LP a direct application of this induction principle for proving for instance (as part of proving (*)):

```
prove   (i#quicksort(s)) = (i#s)
```

is carried out by proving of the following two subgoals.

```
prove   (i#quicksort(es)) = (i#es)
prove   (i#quicksort(sc)) = (i#sc) =>
                             (i#quicksort(m@sc)) = (i#m@sc)
..
```

According to the conventions of LP sc denotes a constant in LP. The first proposition is trivial. Its proof in LP is straightforward. Unfortunately, the second proposition cannot be proved (without induction), because $quicksort(m@sc)$ is rewritten by the second axiom for $quicksort$ as given above into a form that does not match

$$(i\#quicksort(sc)) = (i\#sc)$$

Therefore, we prove the correctness of quicksort by induction on the natural numbers. We carry out following proof tasks in LP by induction on m:

```
prove  (nbr(s) =< m) => ((i#quicksort(s)) = (i#s))
```

Here $nbr(s)$ denotes the number of elements in the sequence s (the length or size of s). The proofs require induction over the natural number m and in the induction step then induction over the sequence s. After this formula has been proved, m can be instantiated by $nbr(s)$ which makes the premise $true$ and yields the required theorem.

Finally for proving that the result of $quicksort$ is in fact sorted we prove the following auxiliary lemmas by the same technique as above.

```
nbr(s) =< m => (k =<< quicksort(s)) = (k =<< s)
nbr(s) =< m => (quicksort(s) << k) = (s << k)
```

Given these lemmas we prove using LP the second basic formulas indicating the correctness of quicksort.

```
prove   (nbr(s) =< m) => sorted(quicksort(s))
```

A critical point to be mentioned again is the demonstration of the consistency of the axioms for the auxiliary functions like << and =<<. Although the consistency of the defining axioms is quite obvious, consistency is not and cannot be formally proved within LP. Therefore the correctness of the machine checked proof for $quicksort$ relies on the consistency of these specifications.

It is quite obvious that it cannot be expected that even a very sophisticated verifier could prove the correctness of $quicksort$ without human interaction. The number of auxiliary lemmas and operators is rather high and too specific.

Chapter 5

Majority Vote

The solution to the majority vote problem is a typical small, but logically intricate, algorithm. The program published in [Misra, Gries 82] for computing the majority of an array of elements is one of those programs that are hard to understand when given only in optimized form. The program reads as follows (let $b[0 : n-1]$ be the array of elements for which the absolute majority is computed):

$$
\begin{array}{lllll}
\multicolumn{5}{l}{i, c := 0, 0;} \\
\textbf{do } i \neq n \rightarrow & \textbf{if} & v = b[i] & \rightarrow & c, i := c+2, i+1 \\
 & \square & c = i & \rightarrow & c, i, v := c+2, i+1, b[i] \\
 & \square & c \neq i \wedge v \neq b[i] & \rightarrow & i := i+1 \\
 & \textbf{fi} & & & \\
\textbf{od} & & & &
\end{array}
$$

The postcondition for this program reads as follows:

Only v may occur more than $n \div 2$ times in b.

Even after studying the verification of this program, given in the assertion method style, it may be clear that, but not why, the algorithm works. This algorithm, which we are going to develop using specification and verification techniques, was discovered by Boyer and Moore (see [Boyer, Moore]).

We are interested in the algorithm as a nice example of specification and verification. Our main concern is giving a convincing and simple to understand presentation of the basic ideas used in the algorithm. We give here a deduction-oriented derivation for this example and a correctness proof for the derivation steps that was constructed interactively with the help of LP.

5.1 The Domain Theory

We basically use numbers and bags as our domains. The specification of numbers is given in the appendix. The specification of bags is given in the following.

```
declare sort Bag, Data

dec var  b, b0, b1, b2 : Bag
dec var  d, d0, d1, d2 : Data

dec op   ebag    : -> Bag
         @@      : Data, Bag -> Bag

         makebag : Data -> Bag
         ++      : Bag, Bag -> Bag
         --      : Bag, Bag -> Bag

         nbr     : Bag -> Nat
         #       : Data, Bag -> Nat
..
assert  ac ++
assert  Bag generated by ebag, @@
assert
        d1 @@ (d2 @@ b) = d2 @@ (d1 @@ b)

        d#ebag          = 0,
        (d#(d0 @@ b))   = if( d = d0,  succ(d#b),  d#b)
        (d#(b1 ++ b2)) = ((d#b1) + (d#b2))

        nbr(ebag)   = 0
        nbr(d @@ b) = succ(nbr(b))

        makebag(d) = (d @@ ebag)

        b -- ebag = b
        (d @@ b) -- (d @@ b0) = b -- b0
        (d#b) = 0 => b -- (d @@ b0) = b -- b0
..
```

The following theorems can be easily obtained by simple proofs in LP.

```
        (nbr(b) = 0)  = (b = ebag)

        (not(b = ebag)) = (0 < nbr(b))
```

```
        nbr(b1 ++ b2) = nbr(b1) + nbr(b2)

        b ++ ebag = b
        (b1 ++ (d @@ b2)) = d @@ (b1 ++ b2)

        ((d @@ b1) = (d @@ b2)) = (b1 = b2)

        (d#b) = 0 => b -- makebag(d) = b

        (b -- (b1 ++ b2)) = (b -- b1) -- b2
        b1 -- b1 == ebag

        (b++b0) -- b0 = b
        ((d @@ b) -- (d @@ ebag)) = b
        when 0 < (d#(b -- b0)) yield 0 < (d#b)
```

An interesting extension of bags is obtained by the introduction of a function *any* that selects an element from a nonempty bag and a complementary function *drop* that drops this element from the bag.

```
dec op  any  : Bag -> Data
        drop : Bag -> Bag
assert  any(d @@ b) = d | (not(b = ebag) & any(d @@ b) = any(b))
        drop(b) = b -- makebag(any(b))
```

According to the semantic models of LP specifications the functions *any* and *drop* are not nondeterministic, but underspecified. In every model of bags we have that for every bag b the element $any(b)$ is the same. But in different models $any(b)$ can compute different results. We obtain the following theorem:

```
prove  b = ebag | b = any(b) @@ drop(b)
```

This theorem is proved in a straightforward manner in LP.

5.2 The Requirement Specification

In the requirement specification we describe the basic problem we want to deal with. Given a bag b an element i is called an *absolute majority* of b, if the following formula holds:

$$nbr(b) < 2 \times (i\#b)$$

Obviously if there exists an absolute majority it is unique. The basic requirement of the majority problem is to compute the absolute majority of a bag in linear time, if it exists. If a bag has no absolute majority we call it *anarchic*. The majority vote problem can be rephrased as follows: develop an algorithm for computing the function

$$major : Bag \to Data$$

that fulfills the following requirement: if the bag b is not anarchic, then the function $major$ returns the element which is the absolute majority in b.

In LP the requirement specification along these lines reads as follows:

```
dec op  major    : Bag -> Data
        anarchic : Bag -> Bool
..
assert  (nbr(b) < (2 * (d#b))) => not(anarchic(b))
        not(anarchic(b)) => (nbr(b) < (2 * (major(b)#b)))
..
```

Contraposition for the first axiom gives:

```
prove   anarchic(b) => ((2 * (d#b)) =< nbr(b))
```

It is not obvious that the specification of the predicate $anarchic$ is complete. The completeness of the specification can be shown as follows. We distinguish two cases. Assume that that there exists an absolute majority d for a bag b. Then according to the first axiom $anarchic(b)$ is $false$. Now assume that that there does not exist an absolute majority for a bag b. Then according to the second axiom $anarchic(b)$ is $true$, since otherwise $major(b)$ would deliver a majority for b. Therefore, assuming that the specification is not contradictory, the predicate $anarchic$ is uniquely specified.

A better specification of anarchic is obtained, if we use existential quantification:

$$anarchic(b) = \neg\exists d \in Data : nbr(b) < 2(d\#b)$$

By this specification the predicate $anarchic$ trivially is uniquely determined. Furthermore it is clear that by the specification no contradictions are introduced. The function $major$ in the LP specification can be seen as the Skolem function for the existential quantifier in this explicit specification of the predicate $anarchic$. Again this explicit version of a specification for the predicate $anarchic$ seems more readable.

Notice, if a bag b is anarchic, nothing is required about the value $major(b)$. The function $major$ is deliberately not uniquely specified. The majority vote problem is underspecified.

5.3 Design Specification

In the design specification we formulate the basic ideas for a solution. In the case of the majority vote the basic idea is to compute the majority by splitting the given bag b

into two bags such that one of these bags is anarchic and the other one is homogeneous. A bag is called *homogeneous*, if all its elements are equal. In LP the notion of a homogeneous bag is easily defined.

```
dec op  homo : Bag -> Bool

assert  (homo(b) & (0 < (d#b))) => (any(b) = d)
        (not(d1 = d2) & (0 < (d1#b)) & (0 < (d2#b))) =>
                                                   not(homo(b))
..
```

Again an explicit specification of the predicate *homo* can be given as follows when using quantifiers:

$$homo(b) \equiv \forall d_1, d_2 \in Data : 0 < (d_1 \# b) \wedge 0 < (d_2 \# b) \Rightarrow d_1 = d_2$$

Next let us define the splitting of bags. We introduce the sort *PairofBag* for pairs of bags.

```
dec sort PairofBag
dec op   cb : Bag, Bag -> PairofBag
         p1, p2 : pairofbag -> Bag
..
assert   p1(cb(b1, b2)) = b1
         p2(cb(b1, b2)) = b2
..
```

We introduce the splitting operation called dis (for “dissection”) and give the basic axioms for it.

```
dec op   dis : Bag -> PairofBag
assert   p1(dis(b)) ++ p2(dis(b)) = b
         homo(p2(dis(b)))
         anarchic(p1(dis(b)))
..
```

It is not obvious that the specification of the function dis is free of contradictions. In other words, it is not obvious that there exists a function dis that fulfills the specifications above. We do not enter into the discussion of the consistency of the specification of dis here. We return to that question in the following section. At the moment, we just want to be sure about the correctness of the design idea, provided it is consistent. The correctness of the design idea is indicated by the following main theorem.

```
prove   not(anarchic(b)) => (nbr(b) < (2 * (any(p2(dis(b)))#b)))
```

The theorem shows that $any(p2(dis(b)))$ has exactly the property required for $major(b)$ and therefore we may solve our problem by defining:

$$major(b) = any(p2(dis(b)))$$

For proving the main theorem in LP we proved a few additional theorems in LP. Three of them are:

```
prove   ((d#p1(dis(b))) + (d#p2(dis(b)))) = (d#b)
prove   not(anarchic(b)) => not(p2(dis(b)) = ebag)
prove   (nbr(b) < ( 2 * (d#b))) => ( 0 < (d#p2(dis(b))))
```

Based on these theorems the main theorem showing the correctness of the design specification has been proved in LP.

5.4 Implementation

In the design specification the function dis was only specified. Now we give a constructive specification for it. We do not include the assertions given in the requirement specification, but start again with a specification based on the specification of bags. We introduce a function $scan$ and specify it by an explicit axiom. The function $scan$ computes the second component of the result of the function dis.

```
declare op scan : Bag -> Bag
```

The dissection of a bag along the lines described in the previous section is not uniquely determined. We now give a constructive description of the function $scan$ based on the functions any and $drop$.

For the empty bag the function $scan$ obviously has to deliver the empty bag as result:

$$scan(ebag) = ebag$$

The empty bag is both homogeneous and anarchic.

Let now a nonempty bag b be given. For an algorithm to compute the function $scan$ we look for a method to compute $scan(b)$ from $scan(drop(b))$. Inductively let us assume that $scan(drop(b))$ is homogeneous and that $drop(b) - scan(drop(b))$ is anarchic. We distinguish two cases:

(1) The bag $a\ @@\ scan(drop(b))$ is homogeneous; then we simply define

$$scan(b) = a\ @@\ scan(drop(b))$$

since clearly $a\ @@\ scan(drop(b))$ is homogeneous and $b - scan(b)$ is anarchic provided $drop(b) - scan(drop(b))$ is anarchic, which we may assume by inductive arguments.

(2) The bag a @@ $scan(drop(b))$ is not homogeneous; then we define

$$scan(b) = drop(scan(drop(b)))$$

Clearly then $scan(b)$ is homogeneous provided $scan(drop(b))$ is homogeneous. Moreover $b - scan(b)$ is anarchic provided $drop(b) - scan(drop(b))$ is anarchic, since $b - scan(b)$ is obtained from $drop(b) - scan(drop(b))$ by adding two different elements. If we add two different elements to a bag that is anarchic the resulting bag is anarchic, too.

These considerations lead to following constructive axioms for $scan$:

```
assert
scan(ebag)               = ebag
scan(any(b) @@ drop(b)) = if(homo(any(b) @@ scan(drop(b))),
                               any(b) @@ scan(drop(b)),
                               drop(scan(drop(b))) )
..
```

Based on the function $scan$ we can define the function dis as given in the previous section in a straightforward way:

$$dis(b) = cb(b - scan(b), scan(b))$$

Then the first specifying equation for dis is trivially fulfilled. Now we could go on and prove the other two equations based on the specifications of the functions *homo* and $anarchic$. However, that would not show that our specification is consistent, since the axioms for the functions *homo* and $anarchic$ might be inconsistent. Therefore, we prove instead the consistency of the axioms of these functions, too, by giving constructive specifications for the functions *homo* and $anarchic$ and then proving the defining axioms as theorems. So we give a new specification based just on bags.

```
dec op  major           : Bag -> Data
        anarchic, homo : Bag -> Bool
..
assert  major(b) = any(scan(b))

        homo(ebag)
        homo(makebag(d))
        homo(d @@  (d0 @@ b)) = ((d = d0) & homo(d0 @@ b))
..
dec op  comp : Bag, Nat -> Bool
assert  comp(ebag, i) = true
        comp(makebag(d), i) = (succ(0) < i)
        comp(d @@ b, i) = (((d#(d @@ b)) =< i) & comp(b, i))
```

```
        anarchic(b) = comp(b, nbr(b))
..
```

Based on this specification we proved the basic theorems of the correctness of the the design and the consistency of the specification in LP:

```
prove  not(anarchic(b)) => (nbr(b) < (2 * (major(b)#b)))
prove  (nbr(b) < (2 * (d#b))) => not(anarchic(b))
prove  (homo(b) & (0 < (d#b))) => (any(b) = d)
prove  (not(d1 = d2) & (0 < (d1#b)) & (0 < (d2#b))) =>
                                                     not(homo(b))
```

The proofs are not particularly difficult to carry out in LP. They mostly are done by induction on b and by cases.

The theorems above prove not only the correctness, but also the consistency, of the developed solution.

The procedural program as given in [Misra, Gries 82] can easily be obtained from the constructive equation for *scan* by representing bags by arrays and homogeneous bags by a pair consisting of a data element and a number, where the number indicates how often the given data element occurs in the represented homogeneous bag. In the program given at the beginning of this chapter the homogeneous bag is represented as follows: $b[i]$ represents the element in the homogeneous bag (if it is not empty) and $c - i$ represents the number of elements in it.

Chapter 6

Code Generation

The verification of the correctness of compilers is a tempting problem which is far from being trivial. Compiler correctness is essential, since, if a compiler is not correct, then the correctness of a program in the source language is not sufficient to guarantee the correctness of the generated target language code.

We concentrate in the following on correctness of code generation and ignore all aspects of compilers having to do with scanning, parsing and context checking. To keep our case study manageable, we use a fairly simple source language and a quite simple target language.

A similar specification of code generation has been verified by Heinrich Hußmann (see [Hußmann 91]) using the RAP system (see [Geser, Hußmann 91]) and the TIP verifier (see [Fraus, Hußmann 91]) on top of it, but full recursion was not treated, there.

6.1 Correctness Issues in Code Generation

Compiler correctness is difficult for the following two reasons. First of all, compilers are rather large and complex pieces of software. Their specification includes a specification of all the details of two programming languages, namely the source and the target language, including context-free syntax, context conditions[1], and semantics. The specification of a compiler must say that syntactically well-formed programs written in the source language are translated into syntactically correct programs in the target language and that the results of executing the generated programs are equivalent to the results of evaluating the source programs.

A functional language that provides full computability such that all computable functions can be expressed necessarily introduces the possibility of recursion and looping

[1] Often called also “static semantics”, which is bad terminology.

computations. The critical problem of looping computations and recursion is that a call of a recursive function may lead to an infinite computation. We speak of *divergence* and of a diverging (evaluation of) function application. It is well known that in a programming language supporting full computability the problem of *diverging computations* cannot be avoided.

In the proof of the correctness of code generation we wish to prove that converging function calls are correctly evaluated by the generated code. For diverging calls we cannot expect that the execution of the generated code converges. So, although the function that does the code generation is total (every syntactically correct program is translated into a finite assembler code program) the code generator necessarily translates (at least certain) diverging programs into diverging code. Divergence is generally undecidable and therefore there is no hope for a compiler that recognizes divergence and treats it by some form of error handling. The possibility of divergence makes the correctness arguments for code generation essentially more difficult.

6.2 The Semantic Basis

In this section we give the specification of the semantic basis that is used both for the source language and for the target language of our compiler.

The semantic basis consists of the sorts and operations that are used to give meaning to our source and target language. Since LP is a first order language we have to encode higher order functions by first order functions, introducing sorts that stand for sets of functions. We use following semantic basis:

```
declare sort
             Data,                          % data elements
             Fct,                           % functions PairDD -
> Data
             Fsb,                           % set of function symbols
             FctFctFct,                     % functionals Fct -
> Fct
             PairDD                         % pairs of data
..
declare var  d, d0, d1, d2 : Data
             o, o0, o1, o2 : Fct
             f, f0, f1, f2 : Fsb
             t, t0, t1, t2 : FctFctFct
             a, a0, a1, a2 : PairDD
..
declare op
         !    : Fct, PairDD -> Data   % function application
         cFct : Fsb -> Fct            % interpretation of Fsb
```

```
test : Data -> Bool           % data as conditions

cons : Data, Data -> PairDD   % pairing function

!    : FctFctFct, Fct -> Fct  % functional application

fix  : FctFctFct -> Fct       % fixed point operator

Def  : Data -> Bool           % Definedness of data
Def  : PairDD -> Bool

bottom : -> Data              % undefined
```

..

The general rules of extensionality of functions (functions are identical if for all arguments their results are identical) and of the fixed point operator ($fix(t)$ is a fixed point of the function t) are formulated by the following axioms:

```
assert   PairDD generated by cons
assert   when (forall o) (t1!o) = (t2!o) yield t1 = t2
assert   when (forall a) (o1!a) = (o2!a) yield o1 = o2
assert   (t!fix(t)) = fix(t)
```

For treating divergence in fixed point theory, it is common to introduce a special semantic element $\perp$ called *bottom*, representing the result of diverging computations. Then all carrier sets D (in our case the set $Data$ of data elements and the set $PairDD = Data \times Data$) are extended by $\perp$:

$$D^{\perp} = D \cup \{\perp\}$$

On every extended set $D^{\perp}$ the so called flat ordering $\sqsubseteq$ is introduced as follows (for $x, y \in D^{\perp}$) :

$$x \sqsubseteq y \equiv (x = \perp \vee x = y)$$

In the proof, we replace this ordering by a predicate:

$$Def : D \to Bool$$

where

$$Def(x) \equiv (x \neq \perp)$$

Then we write

$$Def(x) \Rightarrow x = y$$

instead of

$$x \sqsubseteq y$$

This allows us to make better use of the mechanisms in LP.

This semantic basis of an uninterpreted sort of data elements and binary functions is used to describe the semantics both of the source and the target language.

6.3 The Source Language

The source language in our example is a functional language, called FP in the following, with recursive function declaration. For simplicity we consider a language where all function symbols are binary with fixed interpretations. Each function symbol is associated with a single binary function. Only one function symbol can be declared recursively. Moreover, we use only a fixed pair of "standard" identifiers, namely x_1 and x_2, for parameters. This allows us to write a code generator without having to go into a discussion of symbol tables. Nevertheless, the code generator confronts us with all the principal problems of treating recursion.

```
declare sort   Exp,                % expressions
               Fp                  % functional programs
..
declare var    e, e0, e1, e2, e3, e4 : Exp
               fp : Fp
..
declare op                         % the functional language FP
cst    : Data -> Exp
x1, x2 : -> Exp
eif    : Exp, Exp, Exp -> Exp
eap    : Fsb, Exp, Exp -> Exp

letrec : Fsb, Exp, Exp -> Fp

val    : Exp, Fsb, Fct -> Fct     % interpretation of FP

fu     : Fsb, Exp -> FctFctFct    % functional for FP programs
mean   : Fp -> Fct                % meaning function
mean   : Fsb, Exp -> Fct
..
```

A functional program has always the following simple form

```
letrec(f, e, e0)
```

In a notation providing more syntactic sugar that reads

$$\lambda x1, x2 : e0 \;\; where \;\; letrec\, f = \lambda x1, x2 : e$$

For instance a function f that computes $x_1 * (x_1 + 1) * ... * (x_2 - 1) * x_2$ can be defined as follows:

```
letrec(f,
       eaf(less(x2,x1), cst(1), mult(x1, f(add(x1,cst(1)), x2))),
       f(x1, x2)
      )
```

The semantics of FP is formalized by the following axioms. First it is stated how expressions and FP programs are generated. This is used later as the basis for proofs by structural induction on the structure of terms. Then a specification of the definedness predicate Def is given, followed by the specification of the function val. Finally the function fu is specified and the meaning function $mean$ for programs.

```
assert   Exp generated by cst, x1, x2, eif, eap
assert   Fp generated by letrec
assert                                 % definedness axioms
Def(cons(d1, d2))          = (Def(d1) & Def(d2))
Def(f!a)                   => Def(a)     % strictness assumption
Def(val(cst(d), f, o)!a)   = (Def(d) & Def(a))
Def(val(x1, f, o)!a)       = Def(a)
Def(val(x2, f, o)!a)       = Def(a)

Def(val(eif(e0, e1, e2), f, o)!a) =
   (Def(val(e0, f, o)!a) &
   ((test(val(e0, f, o)!a) & Def(val(e1, f, o)!a)) |
   (not(test(val(e0, f, o)!a)) & Def(val(e2, f, o)!a))))

Def(val(eap(f0, e1, e2), f, o)!a) =
     (Def(val(e1, f, o)!a) &
      Def(val(e2, f, o)!a) &
      Def(if(f0 = f, o, cFct(f0))!
             cons( val(e1, f, o)!a, val(e2, f, o)!a)))

(val(cst(d), f, o)!a) = if (Def(a), d, bottom)
(val(x1, f, o)!cons(d1, d2)) = if(Def(d1) & Def(d2), d1, bottom)
(val(x2, f, o)!cons(d1, d2)) = if(Def(d1) & Def(d2), d2, bottom)
(val(eif(e0, e1, e2), f, o)!a) =
   if(Def(val(e0, f, o)!a),
      if(test(val(e0, f, o)!a),
                val(e1, f, o)!a, val(e2, f, o)!a),
      bottom)
(val(eap(f0, e1, e2), f, o)!a) =
 (if(f0 = f, o, cfct(f0))!cons(val(e1, f, o)!a, val(e2, f, o)!a))
```

```
..
assert  (fu(f, e)!o)  = val(e, f, o)
assert  mean(letrec(f, e, e0)) = val(e0, f, mean(f, e))
        mean(f, e) = fix(fu(f, e))
..
```

Function application is strict in FP such that an application of a function always evaluates to *bottom* if one of the arguments is *bottom*. We use an uninterpreted functional semantic basis for describing the meaning of the functional programming language. Hence our proof of compiler correctness can be applied to any semantic structure with strict basic functions.

6.4 The Target Language

The target language is a simple assembler language ASP for a computer that has a stack and exactly two storage cells for carrying the values of actual parameters of the functional programs. It is not difficult to extend both the target language and the target machine to a more elaborate concept of storage.

The most delicate question for the assembler language is the treatment of the labels for the goto statements. We decided to use symbolic addresses in gotos. Therefore a set of labels is introduced. Labels are strings of markers where markers are elements of the set {0, 1, 2, 3}.

```
declare sort
         Asp,                                 % assembler programs
         Com,                                 % commands of ASP
         Mark                                 % markers for labels
         Label                                % labels
..
declare  op                                   % commands
         lab           : Label -> Com
         apcst         : Fsb -> Com
         jump, cjump   : Label -> Com
         push1, push2  : -> Com
         return, swap  : -> Com
         pushD         : Data -> Com
         pushL         : Label -> Com
                                              % assembler programs
         null : -> Asp
         @    : Com, Asp -> Asp
         ^    : Asp, Asp -> Asp
```

```
        init : -> Label                    % labels
        @ : Mark,  Label -> Label

        0, 1, 2, 3 : -> Mark
..
```

The meaning of ASP is specified by an abstract machine called ASPM. We introduce the sort of configurations representing the control and the data state space for this machine. A configuration consists of

- the global program being executed,
- the stack of data and labels,
- the program counter,
- a pair of data.

Usually a machine executes a program text with the help of a program counter represented by, say, a natural number. The number identifies a position in the program. Initially the counter is 0 and by every instruction it is updated. For our machine we prefer not to introduce natural numbers explicitly to represent program counters. We use a very abstract representation instead. In our machine the program counter is represented by an ASP program that is a postfix of the global program text. Of course, it is rather simple to replace this form of a program counter given by a postfix in the global program by a natural number pointing at the position of the postfix in the global program.

For defining the operational semantics of ASP and ASPM a single-step function is introduced that specifies the steps of execution. The function *goto* computes for a given label and a given ASP program the continuation, which is again an ASP program. The specification of the function *step* that specifies one execution step of the machine follows classical patterns.

```
declare sort
             Stack,                 % stacks of labels and data
             Label,                 % sets of labels
             Conf                   % configurations
..
declare op
             empty :                          -> Stack
             @     : Data, Stack              -> Stack
             @     : Label, Stack             -> Stack

             step  : Conf                     -> Conf
```

```
             goto  : Label, Asp              -> Asp

             cnf   : Asp, Stack, Asp, PairDD -> Conf
..
declare var
             p, p0, p1, p2, pc : Asp
             c, c0, c1, c2     : Com
             m, m0, m1, m2     : Label
             s, s0, s1, s2     : Stack
             k, k0, k1, k2     : Mark
             cf, cf0           : Conf
..
assert
step(cnf(p, s, lab(m)@pc, a)) =
                          cnf(p, s, pc, a)
step(cnf(p, d0@(d@s), apcst(f)@pc, a)) =
                          cnf(p, (cFct(f)!cons(d0, d))@s, pc, a)
step(cnf(p, d0@(d@s), swap@pc, cons(d1, d2))) =
                          cnf(p, d1@(d2@s), pc, cons(d0, d))
step(cnf(p, d@(m@(d1@(d2@s))), return@pc, a)) =
                          cnf(p, d@s, goto(m, p), cons(d1, d2))
step(cnf(p, s, jump(m)@pc, a)) =
                          cnf(p, s, goto(m, p), a)
Def(d) =>
step(cnf(p, d@s, cjump(m)@pc, a)) =
              if(test(d), cnf(p, s, goto(m, p), a),
                          cnf(p, s, pc, a))
step(cnf(p, s, pushD(d)@pc, a)) =
                          cnf(p, d@s, pc, a)
step(cnf(p, s, pushL(m)@pc, a)) =
                          cnf(p, m@s, pc, a)
step(cnf(p, s, push1@pc, cons(d1, d2)))  =
                          cnf(p, d1@s, pc, cons(d1, d2))
step(cnf(p, s, push2@pc, cons(d1, d2))) =
                          cnf(p, d2@s, pc, cons(d1, d2))
..
```

The execution of a goto statement uses the function *goto*. It is easily axiomatized.

```
assert  goto(m, null)        = null
        goto(m, lab(m0)@p)   = if(m = m0, p, goto(m, p))
        goto(m, apcst(f)@p)  = goto(m, p)
        goto(m, swap@p)      = goto(m, p)
        goto(m, return@p)    = goto(m, p)
```

```
        goto(m, jump(m0)@p)   = goto(m, p)
        goto(m, cjump(m0)@p)  = goto(m, p)
        goto(m, pushD(d)@p)   = goto(m, p)
        goto(m, pushL(m0)@p)  = goto(m, p)
        goto(m, push1@p)      = goto(m, p)
        goto(m, push2@p)      = goto(m, p)
..
```

The execution of a program is carried out by iterating the execution steps until the program counter becomes *null*. The iteration of executing steps is defined recursively. The function *exec* iterates the step function until the program counter is *null*. For particular programs this execution may run forever.

The function *cycle* as specified in the following takes as an argument an iteration function w mapping configurations to configurations. If a configuration c is terminal (if the program counter is *null*), then $cycle(w)(c) = c$, otherwise $cycle(w)(c) = w(step(c))$. The least fixed point of the function *cycle* has the fixed point property $fix(cycle) = cycle(fix(cycle))$. Hence, if a configuration c is terminal (if the program counter is *null*), then $fix(cycle)(c) = cycle(fix(cycle))(c) = c$, otherwise $cycle(fix(cycle))(c) = fix(cycle)(step(c))$. This shows that the application of the function $fix(cycle)$ to a configuration c leads to an iterated application of the function *step* until the program counter of the configuration is *null*. If this never happens, then the execution does not terminate.

```
declare sort
        FctCC,                  % functions Conf -> Conf
        FctFCCFCC               % functions FctCC -> FctCC
..
declare op
        exec    :                     -> FctCC

        fix     : FctFCCFCC           -> FctCC
        cycle   :                     -> FctFCCFCC

        !       : FctFCCFCC, FctCC -> FctCC
        !       : FctCC, Conf         -> Conf
..
declare var
        w, w0, w1, w2 : FctCC
        r, r0, r1, r2 : FctFCCFCC
..
assert
        (r!fix(r)) = fix(r)
        exec =  fix(cycle)
        ((cycle!w)!cnf(p, s, null, a)) = cnf(p, s, null, a)
```

```
        ((cycle!w)!cnf(p, s, c@pc, a)) =
                              (w!step(cnf(p, s, c@pc, a)))
..
```

The result of a computation is the top element of the stack when the execution comes to a halt. It can be obtained as follows.

```
declare op
         result : Conf -> Data
assert   result(cnf(p, d@s, pc, a)) = d
```

As we have already pointed out, for our simple machine ASPM a control state is represented by a pair (p_0, p_1) where p_0 is the "global program" and p_1 is a postfix of p_0 representing the program counter. We write $p_1 \preceq p_0$ to indicate that p_1 is a postfix of p_0. If p_1 is a postfix of p_0 then p_1 uniquely determines a natural number i with $0 \leq i \leq length(p_0)$ which can be seen as the more realistic representation of a control counter. It is not difficult to replace the pair (p_0, p_1) by a pair (p_0, i) where i is a natural number with $0 \leq i \leq length(p_0)$.

6.5 Code Generation

Code generation can be easily expressed in LP by equations for the source language constructs. In the following the compilation function is specified in LP. For a FP program $letrec(f, e, e0)$ code is produced that contains code (generated from the expression e) for the recursively declared function identified by the function symbol f and for the expression $e0$. The produced code always contains the code for the recursively declared function followed by the code for for evaluating the expression $e0$. We use the label $init$ to label the code for the recursively declared function the label $1@init$ to label the code for for evaluating the expression $e0$. The introduced functions serve the following purposes (let e be an expression, f be a function symbol, and m be a label).

$trans(e, m, f)$	yields the ASP code for computing the value of e using the label m (and contains only labels obtained from m by prefixing m by markers); for all function symbols in the expression e distinct from the function symbol f, code is produced that corresponds to calls of the respective primitive function, while for the function symbol f, recursive calls are generated,
$comp(p)$	yields the ASP code generated for program p,
$rectrans(f, e)$	yields the ASP code for the recursive declaration of the function symbol f by e,
$etrans(e, f)$	yields the ASP code for computing the value of e.

The LP specification of these functions is straightforward.

```
declare op
        trans     :  Exp, Label, Fsb -> Asp
        comp      :  Fp              -> Asp
        etrans    :  Exp, Fsb        -> Asp
        rectrans  :  Fsb,  Exp       -> Asp
..
assert
trans(cst(d), m, f) = (pushD(d) @ null)
trans(x1, m, f)     = (push1 @ null)
trans(x2, m, f)     = (push2 @ null)

trans(eif(e0, e1, e2), m, f) =
                 ( trans(e0, 0@m, f) ^     % condition
                 ( cjump(m)          @
                 ( trans(e2, 1@m, f) ^     % else-part
                 ( jump(3@m)         @
                 ( lab(m)            @     % then-part
                 ( trans(e1, 2@m, f) ^
                 ( lab(3@m)          @
                   null               )))))))

trans(eap(f0, e1, e2) , m, f) =
                   trans(e2, 2@m, f) ^
                 ( trans(e1, 1@m, f) ^
      if(f0 = f,   swap              @  % recursive call
                 ( pushL(m)          @
                 ( jump(init)        @
                 ( lab(m)            @
```

```
                   null               ))),

                   apcst(f0)        @
                   null               ))

comp(letrec(f, e, e0)) = rectrans(f, e) ^ etrans(e0, f)

rectrans(f, e) = ( lab(init)               @
                 ( trans(e, (0@init), f)   ^
                 ( return                  @
                   null                    )))

etrans(e0, f) =  ( trans(e0, (1@init), f)  ^
                   null                    )
..
```

The code generator generates first the code for the recursive function declaration and then the code for the expression. Recall that there is only one recursive function per program.

All labels are different in the generated code. This is used by the following assertions. We introduce the postfix relation between ASP programs explicitly as follows.

```
declare op   =< : Asp, Asp -> Bool              % postfix ordering
assert       p =< p
assert       when ((p0^pc) =< p) yield (pc =< p)
assert       when ((c@pc) =< p)  yield (pc =< p)
```

The rule for computing continuations then reads as follows.

```
assert when ((lab(m)@p1) =< comp(fp))
       yield goto(m , comp(fp)) = p1
..
```

This assertion can easily be proved with the axioms given for the function *goto*. It is certainly part of the correctness proof for code generation to prove these assertions. We do not describe here how to carry out this proof, but rather concentrate on the main proof of correctness of code generation.

6.6 Verification

Having specified the compiler function we can assign a meaning to programs in the source language by translating them to ASP programs, executing them and taking the result as the meaning of the source program. In LP this is expressed as follows.

```
declare op   cmean    :  Fp -> Fct
             xi       :  Fsb, Exp, Exp, FctCC, Stack -> Fct
..
assert
cmean(letrec(f, e, e0)) = xi(f, e, e0, exec, empty)

(xi(f, e, e0, w, s)!a) =
   result(w!cnf(comp(letrec(f, e, e0)), s, etrans(e0, f), a))
..
```

Compiler correctness now can be specified as follows. Our compiler is correct, if for every FP program and every pair of arguments for this program, whenever this program terminates, so does the execution of the generated code and the results coincide.

Of course, before starting a more sophisticated proof, an idea, how to organize the proof, must be found. We speak of the *architecture* of a proof. Such a proof idea often needs careful investigation of the specifications under consideration, especially for finding out as early as possible whether a proof idea does not work.

If the specifications are large and complex, a tuned notation is extremely helpful. Therefore we introduce in addition to the LP notation a more sophisticated mathematical one that hopefully makes the discussion of the proof principles more comprehensible. The introduction of such a tuned mathematical shorthand is of significant help in finding the correctness proof.

Let us first formulate the correctness condition for code generation. Let $e, e0$ be expressions and f be a function symbol. The program

$$letrec(f,\ \ e0,\ \ e1)$$

stands for the function (note, once more, the parameters are always denoted by the identifiers $x1, x2$; therefore every expression containing only these two variables can be read as representing a binary function in the arguments $x1$ and $x2$):

$$\lambda x1, x2 : e0 \ \ \textbf{where} \ \ letrec\ f = \lambda x1, x2 : e$$

The recursive declaration for f associates a function $\tilde{f}$ with the function symbol f by the fixed point operator where

$$\tilde{f} = fix(\tau_f^e) \ \ \textbf{where} \ \ \tau_f^e[\tilde{f}] = val(e, f, \tilde{f})$$

Here $val(e, f, \tilde{f})$ stands for the value obtained by evaluating the expression e while using the function $\tilde{f}$ as the interpretation of the function symbol f.

The functional *cycle* is abbreviated by σ in the following. It has the functionality

$$\sigma : (Conf \to Conf) \to (Conf \to Conf)$$

where for all configurations c:

$$(\sigma[w])(c) = step(c)$$

if the program counter of the configuration c is not $null$ and

$$(\sigma[w])(c) = c$$

otherwise. If the program counter is $null$, $result(c)$ gives the top data element of the stack in configuration c.

The fundamental theorem indicating the correctness of the compiler reads as follows (remember, in the specification of ASPM the function $\mathit{fix}(\sigma)$ was abbreviated by *exec*):

$$val(e0, f, fix(\tau_f^e)) = \xi_f^{e,e0}(fix(\sigma))$$

where

$$\xi_f^{e,e0}(g).a = result(g(\mathit{cnf}(p, s, p0, a)))$$

where a is a pair of arguments and

$$p = rectrans(f, e)\hat{\ }p0$$

$$p0 = etrans(e0, f)$$

The core of the correctness proof for code generation is therefore a proof of the equivalence of two recursively defined functions.

Basically, as well-known from fixed point theory, there are the following proof rules available for proving properties about recursively defined functions. Let τ be a continuous function.

Fixed Point Rule:

$$fix(\tau) = \tau[fix(\tau)]$$

Least Fixed Point Rule:

$$g = \tau[g] \Rightarrow fix(\tau) \sqsubseteq g$$

Computational Induction:

$$\sqcup\{f_i : i \in N\} = fix(\tau) \text{ where } f_0 = \Omega,\ f_{i+1} = \tau[f_i]$$

Here Ω stands for the function $\lambda x : \bot$.

When trying to carry out a proof it is very important to recognize whether a proof technique might be successful or not. It is a major decision for the correctness proof for the code generator which of the rules are to be applied.

Since LP is equation oriented, a proof based just on the fixed point rule would be most convenient. However, a careful inspection of our basic theorem shows that a proof purely based on the fixed point rule cannot work. $fix(\sigma)$ represents repetitive recursion. Therefore, if an application $fix(\sigma)$ diverges for a given argument nothing can be said about the value that other fixed points of σ may yield for this argument.

Consider for the moment the natural numbers for the sort Data and the rather stupid recursive declaration:

$$letrec\ f(x1, x2) = f(x1, x2) + 1$$

There is no solution for f besides $f = \Omega$. For the function $exec$ where $exec = fix(\sigma)$ we obtain:

$$\begin{gathered} exec(cnf(p, s, trans(eap(f, x1, x2)), a)) = \\ exec(cnf(p, \tilde{s}, trans(eap(f, x1, x2))\hat{\ }c, a)) \end{gathered}$$

where $\tilde{s}$ is obtained from s by pushing a number of labels and parameter values onto the stack s and c is some code.

Certainly there are many functions that fulfill the equation for $exec$ above. So pure fixed point reasoning cannot be sufficient for proving the correctness of $fix(\sigma)$. It is the least fixed point of the function σ that should have the required property. Therefore we use the following rule that is a generalization of transformational induction. Let κ be a continuous function.

Transformational Induction:

$$(\forall \tilde{f} : \kappa[\tilde{f}] \sqsubseteq \xi \Rightarrow \kappa[\tau[\tilde{f}]] \sqsubseteq \xi) \Rightarrow \kappa[fix(\tau)] \sqsubseteq \xi$$

The correctness of this rule can be proved in a straightforward way by computational induction (for τ). From

$$\forall \tilde{f} : \kappa[\tilde{f}] \sqsubseteq \xi \Rightarrow \kappa[\tau[\tilde{f}]] \sqsubseteq \xi$$

we obtain for $\tilde{f}_0 = \Omega$, $\tilde{f}_{i+1} = \tau[\tilde{f}_i]$ by induction on i that

$$\kappa[\tilde{f}_i] \sqsubseteq \xi$$

Therefore

$$\kappa[fix(\tau)] = \kappa[\sqcup\{\tilde{f}_i : i \in N\}] \sqsubseteq \xi$$

This rule, in contrast to computational induction, has the advantage that we do not have to introduce the natural numbers explicitly into the proof.

By this rule we can show that, whenever $fix(\tau_f^e)(d1, d2)$ converges, so does $fix(\sigma)(cnf(..., cons(d1, d2))$ and both coincide. This means that we prove the somewhat weaker theorem:

$$val(e0, f, fix(\tau_f^e)) \sqsubseteq \xi_f^{e,e0}(fix(\sigma))$$

To prove this theorem by induction on the structure of the expressions we need an appropriate embedding.

We formulate this embedding and based on it the correctness theorem. In the following we consider the program

$$letrec(f, \quad e0, \quad e1)$$

in the source language FP; let the following abbreviations be introduced:

$$\begin{array}{ll} p_0 & = comp(letrec(f, e_0, e_1)) \\ \tau & = fu(f, e_0) \\ \nu[\tilde{f}, e] & = val(e, f, \tilde{f}) \\ \zeta_m[e] & = trans(e, m, f) \\ \kappa[s, p_1] & = \lambda a : fix(\sigma)(cnf(p_0, s, p_1, a)), \end{array}$$

Note, f stands for a function symbol while $\tilde{f}$ stands for a function. The correctness of code generation is implied by the following formula:

$$\forall e : \Theta[fix[\tau], e]$$

where $\Theta[\tilde{f}, e]$ is an abbreviation for:

$$\zeta_m[e]\hat{\ }p_1 \preceq p_0 \Rightarrow \kappa[\nu[\tilde{f}, e](a)\hat{\ }s, p_1](a) \sqsubseteq \kappa[s, \zeta_m[e]\hat{\ }p_1](a)$$

A first attempt to carry out the proof is to use computational induction to prove:

$$\tilde{f} \sqsubseteq \tau[\tilde{f}] \wedge \Theta[\tilde{f}, e] \Rightarrow \Theta[\tau[\tilde{f}], e]$$

by induction on the structure of the expression e. When trying to carry out this proof, then we have to face the following nasty problems.

- In the induction step for the case $e = eap(f, e_3, e_4)$ we have to prove the theorem for $e = e_0$ where e_0 is the body of the recursive declaration. For e_0 an induction hypothesis is not available, since e_0 is not a subexpression of e, in general, and therefore the proof gets stuck.

In our first version of the proof of compiler correctness this problem was overcome by structuring the proof into two steps treating recursive calls and nonrecursive calls separately. Then a complicated rephrasing of the semantics of FP was used such that the compiler correctness for this semantics could be proved in LP as well as the equivalence of the orignal semantical definition for FP with the rephrased one.

After the proof was finished, Birgit Schieder studied it and discovered a more direct proof that avoided the rephrasing of the semantics of the source language, by proving a more general formula using quantifiers explicitly (see [Schieder 92]). As Schieder pointed out to me the proof can be greatly simplified by proving the following main theorem

$$\tilde{f} \sqsubseteq \tau[\tilde{f}] \wedge \forall e : \Theta[\tilde{f}, e] \Rightarrow \forall e : \Theta[\tau[\tilde{f}], e]$$

instead of the theorem above. Clearly, this theorem also implies by computational induction:

$$\forall e : \Theta[fix[\tau], e]$$

but it turns out that it is much simpler to prove. Technically it means that we apply computational induction first and then apply structural induction on the structure of the expressions, rather than the other way around. In LP the main theorem reads as follows:

```
prove
when
(forall a, s, p1, m1, e)
((Def((val(e, f, o) ! a) ) &
((trans(e, m1, f) ^p1) =< p0))
=>
((exec!cnf(p0, s, trans(e, m1, f) ^p1, a)) =
(exec!cnf(p0, (val(e,f,o)!a) @ s, p1, a)))) -> true,
p0 -> comp(letrec(f,e0,e1))
yield
((Def(val(e, f, o) ! cons(d1, d2)) &
((trans(e, m, f)^p1) =< p0) )
=>
(exec!cnf(p0, s, trans(e, m, f) ^p1, cons(d1, d2) )) =
(exec!cnf(p0, (val(e,f,o)!cons(d1,d2) ) @ s, p1, cons(d1, d2))))
..
```

The proof requires some definedness reasoning that is introduced in the following paragraph.

6.7 Logic of Definedness

When proving theorems by computational induction we have to reason about definedness. The introduction of the definedness reasoning into the LP specification reads as follows.

```
declare op   =< : Data, Data -> Bool
assert       (d1 =< d2) = ((d1 = bottom) | (d1 = d2))
assert       Def(d1) = not(d1 = bottom)
```

For functions the partial ordering is introduced as follows:

```
declare op   =< : Fct, Fct -> Bool
assert       when (forall a) (o1 ! a) =<  (o2 ! a) yield o1 =< o2
assert       o1 =< o2 => (o1!a) =< (o2!a)
```

Furthermore the monotonicity of the involved functions is required.

```
assert       (a1 =< a2) => ((o!a1) =< (o!a2))
assert       (o1 =< o2) => ((t!o1) =< (t!o2))
```

These rules are used in the correctness proof.

6.8 Proof of the Main Theorem

Finally we proved the main theorem in LP using the axioms of definedness. The proof itself is not particularly difficult. It just is a long proof by structural induction taking into account all the forms expressions in FP can have.

Proving the correctness of a compiler is a quite serious proof task. It requires in particular techniques from domain and fixed point theory. The consistency of the LP specification for the compiler correctness proof is not obvious at all. It is basically justified by the consistency of fixed point theory. Also the correct application of the rules from fixed point theory has not been checked within LP.

Proving compilers correct is a task that has been investigated by many researchers (see the pioneering paper [London 71], and furthermore, for instance, [Manasse, Nelson 84], [Bevier 89], [Hußmann 91], [Reeves 91], [Rus 90]), however, a rigorous formal proof of the correctness of recursive functions was always considered to be one of the more difficult aspects.

The correctness proof for code generation involves, in general, the following forms of induction:

1. computational induction,
2. induction on the structure of the expressions.

In the proof presented above, both induction principles were combined to prove a carefully selected generalized theorem, from which the main theorem of compiler correctness can be deduced.

Chapter 7

Interactive Queues

LP is a functional specification language. Properties about interactive systems can be proved with the help LP, if we give functional representations for such systems. Of course, we can define state transition systems by functional means and therefore treat such systems with the help of LP. Another way to reason about interactive systems, also supported by LP, is offered by functional system descriptions along the lines of [Broy 90].

In the following sections we specify an interactive queue, give an implementation in terms of states and an implementation by an infinite network. We prove the correctness of the implementations within LP. The example is taken from [Broy 88] where the theoretical background is described in detail.

7.1 The Domain Theory

We use the the data types of streams and of messages. For simplicity we just use natural numbers as messages (for every natural number i by $dt(i)$ we denote the message consisting of i) and a special message rq as a request signal. The set of messages reads as follows:

```
declare sort   Message
declare var    m, m1, m2 :        Message
declare op     rq        :     -> Message
declare op     dt        : Nat -> Message
assert         (rq = dt(i)) = false
assert         Message generated by rq, dt
```

Streams are sequences of messages. The specifications of streams reads in LP as follows:

```
declare sort  Stream
declare var   s, r, s1, s2, r1, r2 : Stream
              m, m1, m2            : Message
..
declare op    es :                 -> Stream     % empty stream
              ^ : Message, Stream -> Stream     % prefixing
              ^ : Stream, Stream  -> Stream     % concatenation
..
assert        es^s = s
              s^es = s
              (m^s)^r = (m^(s^r))

              ((m^s) = (m1^s1)) = ( m = m1 & s = s1 )
              (es = (m^s)) = false

              (s^r)^s1 = s^(r^s1)
              (s^s1 = s^s2) = (s1 = s2)
              (s1^s = s2^s) = (s1 = s2)
..
assert        stream generated by es, ^:Message, Stream -> Stream
```

The sort *Stream* as introduced here includes only finite streams, not infinite streams as the domain of streams as used in [Broy 90].

7.2 The Requirement Specification

The behavior of interactive system components can be described using the concept of streams. We do not go deeper in this subject and just refer to [Broy 90].

An interactive queue that is initially empty can be specified based on the concept of streams in LP as follows. Here the predicate *norq* is used as an auxiliary construct. The proposition $norq(s)$ is *true*, if and only if the stream s does not contain the signal rq.

```
declare op   q    : Stream -> Stream
declare op   norq : Stream -> Bool
assert
             q(rq^s) = rq^q(s)
             norq(r)    => q(dt(i)^(r^(rq^s))) = dt(i)^q(r^s)
             norq(s)    => q(s) = es

             norq(es)       = true
             norq(rq^s)     = false
```

```
             norq(dt(i)^s) = norq(s)
             norq(r^s)     = norq(r) & norq(s)
..
```

The specification of the function q basically expresses the following properties:

- when a request signal rq is received while the queue is empty then this signal is sent back and the queue remains empty;
- when a request signal rq is received after receiving a data message $dt(i)$ followed by a finite sequence r of data messages, then the first data message in the input stream, which is $dt(i)$, is sent in response;
- as long as there are no request signals in the input there is no output.

This describes the behavior of a queue in terms of finite histories of interactions. The function q is uniquely specified this way.

In the following we give two constructive descriptions for interactive queues. The first one is based on a state concept. The second one uses an infinite network of cells.

7.3 State Based Implementation

A more constructive description of q is obtained by basing q on an auxiliary function qs that is specified in a more state-oriented form as follows

```
declare op   qs : Stream, Stream -> Stream
assert       qs(es, es)          = es
             qs(es, rq^s)        = rq^qs(es, s)
             qs(dt(i)^r, rq^s)   = dt(i)^qs(r, s)
             qs(r, dt(i)^s)      = qs(r^(dt(i)^es), s)
..
```

The first parameter of qs should be seen as a sequence (or more appropriately as an element of sort *Queue*) rather than as a stream. It can be understood as the *state* of the interactive queue. A more adequate way to express this would be to associate with qs the functionality

$$qs : Queue \rightarrow (Stream \rightarrow Stream)$$

where *Queue* is the sort of sequences equipped with the operations for queues.

The function qs provides an implementation of the function q. This is proved in LP by proving:

```
prove    q(s) = qs(es, s)
```

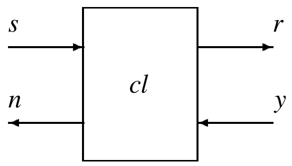

Figure 7.1: Graphical representation of the component *cl*

For carrying out this proof we proved following auxiliary theorems in LP.

```
prove     norq(r) & norq(s) => qs(r, s)     = es
prove     norq(r1) =>          qs(r^r1, s) = qs(r, r1^s)
prove     norq(r) =>           q(r^s)      = qs(r, s)
```

All these proofs are carried out in LP by rather straightforward induction on r and s.

7.4 Interactive Queues as Infinite Networks of Cells

A queue as specified above can be implemented by an infinite (or better an unbounded) network of storage cells, each of which can store at most one data item.

7.4.1 Storage Cells

A storage cell is an interactive component that has two input channels and two output channels. It can be modeled by a function that receives two input streams and produces two streams as output. We describe a storage cell by a ternary function cl. In an application $cl(m, s, y)$ the first argument m represents the state of the cell (the message stored in the cell). The other two parameters are the input streams. The most adequate functionality for cl therefore would be:

$$Message \to (Stream \times Stream \to Stream \times Stream)$$

This functionality can be obtained from the functionality of cl by currying. A graphical representation of a cell cl is given in Figure 7.1. Initially, the cell is empty. This is modeled by specifying that rq is stored in the cell initially.

The behavior of a cell can be informally described as follows. If the cell that has currently stored the data message rq, which, as pointed out, indicates that the cell is empty, gets a message m on channel s then:

- if $m = rq$, then the signal rq is echoed back on channel n to indicate that the cell is empty and therefore cannot satisfy the request and the value rq remains the value stored in the cell;
- if $m = dt(i)$, then the data element $dt(i)$ becomes the value stored in the cell.

If the cell that has currently stored the data message $dt(j)$ gets a message m on channel s then:

- if $m = rq$, then the data element $dt(j)$ is sent on channel n in response and a request is sent on channel r; the message received in response to this request on channel y gets the new value stored in the cell;
- if $m = dt(i)$, then the data element $dt(i)$ is sent further on channel y; the value $dt(j)$ remains the value stored in the cell.

This behavior can be expressed in LP as follows (clh is introduced just for technical reasons as an auxiliary function):

```
declare sort PairofStream
declare op   cons   : Stream, Stream -> PairofStream
             p1, p2 : PairofStream -> Stream
             cl     : Message, Stream, Stream -> PairofStream
             clh    : Stream, Stream -> PairofStream
..
assert       p1(cons(s, r)) = s
             p2(cons(s, r)) = r

             cl(m, es, r) = cons(es, es)

             cl(rq, dt(i)^ s, r) = cl(dt(i), s, r)
             cl(rq, rq^s, r) =
              cons(rq^p1(cl(rq, s, r)), p2(cl(rq, s, r)))

             cl(dt(j), rq^s, r) =
              cons(dt(j)^p1(clh(s, r)), rq^p2(clh(s, r)))
             clh(s, m^r) -> cl(m, s, r)
             clh(s, es) = cons(es, es)
             cl(dt(j), dt(i)^s, r) =
              cons(p1(cl(dt(j), s, r)), dt(i)^p2(cl(dt(j), s, r)))
..
```

For these functions a number of theorems are proved in LP that will be used in the correctness proofs for the network given below:

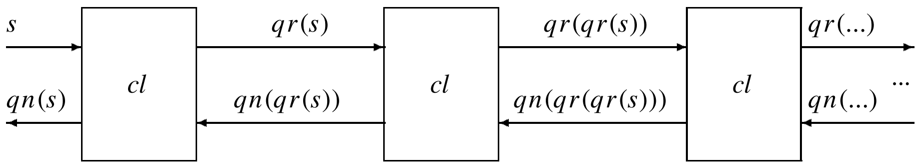

Figure 7.2: Graphical representation of the network of cells

```
prove  norq(r) => p1(cl(m, r, s1)) = es
prove  norq(r) => p1(cl(dt(i), r^(rq^s1), r1)) =
                                   (dt(i)^p1(clh(s1, r1)))
prove  norq(r) => p1(cl(dt(i), r^s1, r1)) =
                                   p1(cl(dt(i), s1, r1))

prove  norq(r) => p2(cl(dt(i), r, s1)) = r
prove  norq(r) => p2(cl(rq, dt(i)^(r^s1), s)) =
                                   r^p2(cl(rq, dt(i)^s1, s))
```

These theorems actually show interesting properties of the cell. They allow a better understanding of the behavior of the cell.

7.4.2 The Network

An infinite network of cells can be used to implement an interactive queue. A graphical representation of such an infinite network of cells is given in Figure 7.2. The network can be logically described by a recursive definition for streams.

We introduce two function symbols qn and qr. For a stream s the stream $qn(s)$ denotes the output produced by the network and the stream $qr(s)$ denotes the stream produced inside the network as output by the left cell in the network used as input for its right neighbor.

```
declare op   qn, qr: Stream -> Stream
```

The network can be described by recursive equations for qr and qn in terms of the cell function cl:

$$(qn(s), qr(s)) = cl(rq, s, qn(qr(s)))$$

We do not simply use these equations as assertions in LP. The reason is as follows: in the theoretical framework as given in [Broy 88] fixed points are guaranteed to exist by the monotonicity of the functions involved. Therefore we simply could state qr and qs by the equations shown above. Since we did not introduce the theoretical framework of fixed point theory explicitly, we do not give fixed point definitions for qn and qr. We just introduce predicates as abbreviations for the fixed point equations.

```
declare op   fpr : stream -> bool
assert       fpr(s) == (qr(s) = p2(cl(rq, s, qn(qr(s)))))
declare op   fpn : stream -> bool
assert       fpn(s) == (qn(s) = p1(cl(rq, s, qn(qr(s)))))
```

The correctness of the network given above is proved in two steps. In the first step we prove partial correctness, also called *safety* properties. In the second step we prove *liveness* properties.

7.4.3 Safety

The network is basically described by equations for streams. We prove safety properties by giving explicit specifications for the functions qn and qr proving that the functions qn and qr as specified above form a fixed point of the equation describing the network.

Again an auxiliary function qc is introduced just for convenience for specifying qr.

```
declare op   qn, qr: Stream -> Stream
             qc: Stream, Stream -> Stream
assert       qn(es)    = es
             qn(rq^s1) = (rq^qn(s1))
             norq(s1) => qn(s1) = es
             norq(r) =>
               qn(dt(i)^(r^(rq^s))) = (dt(i)^qn(r^s))

             qr(s) = qc(es, s)

             qc(es, (dt(i)^s)) = qc((dt(i)^es), s)
             qc(es, (rq^s)) = qc(es, s)

             qc(r, es) = es
             qc(dt(j)^s, rq^s1) = (rq^qc(s, s1))
             qc(dt(j)^s, dt(i)^s1) =
                        (dt(i)^qc((dt(j)^s)^(dt(i)^es), s1))
             norq(r) => qc(es, (dt(i)^r)) = r
             norq(r) => qc(dt(i)^(s1), r^s) =
                                      r^qc(dt(i)^(s1^r), s)
..
```

For proving the required safety properties of the network we prove that the specified functions qn and qr are indeed fixed points. For carrying out this proof in LP we proved the following lemmas:

```
prove  fpr(es)
prove  fpr(s) => fpr((rq^s))
prove  fpr((dt(i)^es))
prove  norq(r) => fpr(dt(i)^r)
prove  fpr(s) => fpr((dt(j)^(rq^s)))
prove  norq(r) => (fpr(dt(i)^(r^s)) =>
                          fpr(dt(j)^(dt(i)^(r^(rq^s)))))

prove  fpn(es)
prove  fpn(s) => fpn((rq^s))
prove  norq(r) => fpn((dt(i)^r))
prove  fpn(s) => fpn((dt(j)^(rq^s)))
prove  norq(r) => (fpn(dt(i)^(r^s)) =>
                          fpn(dt(j)^(dt(i)^(r^(rq^s)))))
```

These theorems then allow us to prove the basic safety property:

```
prove  fpn(s)
prove  fpr(s)
```

This concludes the safety proof. In the proof of the safety properties a number of properties of the function *cl* are required as well as certain properties of the function clh. For doing the proofs certain skolem functions have been introduced. Their description is given in the appendix. Note, again, the proof gets more tedious, since in LP we cannot assume that there are fixed points of the functions involved. Simply giving the axioms

$$fpn(qn(s)) \wedge fpr(qr(s))$$

as definitions for qr and qs might already have introduced an inconsistency in LP. In SPECTRUM the semantic theory ensures that monotonic functions have fixed points and therefore a consistency proof is not necessary.

7.4.4 Liveness

We prove the liveness properties for the network by proving that the output has the required length.

This is probably not the most elegant way of proving liveness. However, since we were interested in getting experience with LP in carrying out those types of proofs, we constructed the proof this way. For doing the proof we introduce two auxiliary functions.

```
declare op   nbr, nbrrq : stream -> nat
assert       nbr(es)  = 0
             nbr(m^s) = succ(nbr(s))
..
assert
             nbrrq(es)       = 0
             nbrrq((rq^s))   = succ(nbrrq(s))
             nbrrq(dt(i)^s) = nbrrq(s)
..
```

Then we have: $nbr(s)$ denotes the number of elements, the length, of stream s. $nbrrq(s)$ denotes the number of occurrences of the signal rq in the stream s.

For proving the liveness properties we do not include the assertions about qn and qr as given explicitly in the safety part, but assert that the functions qn and qr are fixed points[1]:

```
assert    fpr(s) & fpn(s)
```

Now, we observe that for the function q as introduced in the requirement specification we have (a proof by induction is straightforward):

$$nbr(q(s)) = nbrrq(s)$$

In the safety part we have already shown that q is a fixed point. It remains to show that it is the least fixed point. Therefore it is sufficient to show the following lemma:

$$fpr(s) \wedge fpn(s) \Rightarrow nbrrq(s) \leq nbr(qn(s))$$

The liveness proof is quite straightforward. It uses a number of properties of cl and clh that are listed in the following. Again these properties are verified by straightforward (mostly induction) proofs in LP. Finally we prove in LP:

```
prove nbr(p2(cl(m, s, r))) =< nbr(s)
prove nbr(p2(cl(rq, m ^ s, r))) < nbr(m ^ s)
prove nbrrq(p2(cl(m, s, r))) < succ(nbr(r)) =>
                                  nbr(p1(cl(m, s, r))) = nbrrq(s)
```

Based on these properties we proved the following theorem in LP by induction on i:

```
prove nbr(s) < i => nbrrq(s) =< nbr(qn(s))
```

This theorem immediately implies:

[1]Since we have already proved in the safety part that fixed points exist we can be sure that this does not cause inconsistencies.

```
prove nbrrq(s) =< succ(nbr(qn(s)))
```

This concludes the liveness part of the correctness proof for the implementation of the interactive queue by a network.

7.4.5 Fixed Point Reasoning

A more elegant and simpler proof for the correctness of the network is obtained by defining the function qn as follows:

$$qn(s) = p_1(fix\ \lambda c : cl(rq, s, qn(p_2(c))))$$

where s denotes a stream and c denotes a pair of streams and p_1 and p_2 denote the first and second projection function for pairs of streams and fix denotes the fixed point operator. The following rule is justified by fixed point arguments (let $\tilde{c}$ be a pair of streams and $\tilde{c}\hat{}c$ denote the elementwise concatenation of the pairs of streams $\tilde{c}$ and c and f be a function from pairs of streams to pairs of streams):

$$fix\ \lambda c : \tilde{c}\hat{}f(c) = \tilde{c}\hat{}fix\ \lambda c : f(\tilde{c}\hat{}c)$$

Using this rule the three defining equations for interactive queues as given in section 7.2 can be derived in LP in a rather straightforward way by fold/unfold techniques for the recursive definition of qn as given above. Since LP does not support λ-notation and higher order functions (such as fix), some parts of λ-calculus had to be axiomatized explicitly in LP to carry out the proof along these lines. Although we have carried out these proofs in LP, too, we do not give the LP version of the proof, because we do not wish to go into the axiomatization of λ-calculus within LP here.

This concludes the example of the interactive network of storage cells implementing an interactive queue.

Chapter 8

Conclusion

All the examples of developments as treated in the previous chapters were carried out using LP to prove all the theorems. The purpose of this work was not, of course, to be sure about the correctness of the examples, but rather to get experience in deduction oriented program and system development using a tool like LP.

Finally, we want to discuss more general aspects of formal techniques in software and system development and draw at least a number of conclusions in the light of the treatment of the examples.

8.1 Discussion of Formal Techniques

The most basic questions, generally asked in connection with formal techniques and verification support systems, are listed in the following:

(1) Is formal specification and verification possible, in principle?

(2) What does it cost?

(3) How does it pay?

(4) Can it be applied, in practice?

It was, of course, apart from technical and scientific aspects, one of the purposes of this study is to obtain input for answers to these questions.

The answer to question (1) is definitely "yes" from my point of view. I cannot imagine a piece of software, where formal specification and verification is, in principle, not possible. Our specification techniques and verification techniques are developed far enough, today. However, certainly there are applications where formal specifications may not be adequate[1]. And whether specification and verification is practically possible and useful when working with large programs is an open question. The difficulties of scaling up formal techniques is a crucial problem.

The second question cannot be answered very generally. The study for the examples given in the previous sections was carried out during a stay of 14 weeks at SRC in Palo Alto during the summer and autumn 1991. The work on the verification and specification of the examples certainly took less than 4 manweeks. Of course, the examples studied are toy examples and therefore the time for constructing the proofs seems rather long. On the other hand, this time includes all the steps of the problem specification. Moreover, if in a more practical software production environment similar applications are treated again and again, then specifications and proof techniques may be reused in a much higher degree. Formal techniques may help to produce high quality software such that reuse is more attractive than for software nowadays. If in a particular area of application the domain theory is formalized by specifications and a number of theorems and proof techniques have been established, then the overhead in tackling new, but related development tasks is drastically reduced.

The answer to question (3) is even more difficult. Certainly, a lot of bugs in program development can be caught by specification and verification techniques. However, it is necessary to emphasize that a formally specified and verified program is not necessarily correct in the pragmatic sense. The specification may not capture the requirements properly, the proof itself may be carried out not properly (even when using support systems errors can occur in stating the axioms, not to speak about the correctness of the support system itself). On the other hand, when observing how much time and energy is wasted in testing, debugging and maintaining incorrect badly documented software, we may conclude that even when formal techniques are expensive they may pay.

Question (4) again leads to two answers. In principle, formal techniques can be practically applied, right now. However, when trying to apply formal techniques, in practice, the following problems will give software engineers pioneering in the application of formal methods a hard time:

- lack of experience and well educated people,
- lack of a well worked out methods,
- lack of tool support for organizing the work.

Nevertheless, for a number of areas of application formal techniques might nevertheless already be applicable and cost-effective.

[1] What is the formal specification of a pattern recognition procedure for handwritten letters or for spotting koala bears in the jungle.

8.2 Areas of Enhancement for LP

When using a tool like LP an interesting question concerns aspects for further development. It is certainly important to underline here that the suggestions of further development are not to be misunderstood as a criticism of the design decisions that have led to the current version of LP. I am perfectly in agreement with a strategy to start with a simple and basic tool first, to gather some experience and than do a next step ("little steps for little feet" as Jim Horning says).

We have already discussed the significance of language features in chapter 2. Here a lot can be added, especially features already available in Larch, to make the LP notation more comfortable. In the following we rather concentrate on questions of deduction and proof support.

8.2.1 More Power for the Proof Machinery

First of all, it has to be stated already that LP is a proof machinery that needs a lot of interaction. LP gives practically no hints for finding proofs with respect to the proof structure. The basic proof idea has to be completely provided by the user. However, LP is much more than just a proof checker, although it can be and has been used that way. When carrying out the basic proof steps, the information displayed in the interactive development of a proof is valuable for designing the proof. It, in particular, gives hints why a certain proof step fails, and in some cases, may indicate that the conjecture that is to be proved may be not a theorem, after all.

At the moment, often it takes a painfully large number of steps of interaction (including failing attempts) to carry out quite simple proofs of rather obvious propositional formulas. Here time consuming proofs by cases are often needed. Certainly, additional proof machinery that can be invoked, when a level of detail is reached during the construction of a proof that allows to complete the proof by purely propositional reasoning and by instantiating a number of axioms would be very helpful. This could be done by an efficient decision procedure.

When dealing with constructive specifications, induction proofs become crucial. In LP, induction on the term structure is well supported. More sophisticated techniques of induction can be envisaged, including complete induction and induction over terms, as well as variables.

Another source of frustration is the amount of input needed by the LP system when instantiating formulas. Although the "critical pairs" concept can help here, I found its use rather indirect. More explicit commands offered by LP would be welcome.

8.2.2 Support of More Refined Logical Theories

LP supports just a subset of first order predicate logic plus induction. This is quite powerful and, in principle, sufficient. More sophisticated logical concepts like higher order functions, full quantifiers, logic of partial functions, fixed point theory and much more can be implemented by specifications on top of LP.

However, this proceeding has disadvantages. As it can be seen by the compiler example, it makes a difference, whether a sort for representing functions is introduced or whether functional sorts are available. The same applies for reasoning about partial functions.

It would be nice to make a number of theory and logic transformers available on top of LP, such that by invoking these transformers more refined logical theories are made available to the users.

In any case full first order predicate logic with full treatment of quantifiers should be supported.

8.2.3 Advanced Proof Support

Certain proof techniques are very well supported by LP. For instance, all kinds of rewriting proofs can be easily carried out. Other proof techniques are less adequately supported.

For instance, proofs where for a transitive relation $\leq$ a proposition

$$t_0 \leq t_n$$

is proved by giving a sequence of terms t_i such that:

$$t_0 \leq t_1 \leq ... \leq t_n$$

have to be carried out in LP by instantiating the law of transitivity over and over again.

Such proofs are needed in many applications of computer science. Here more tuned proof techniques might be provided.

Often it is helpful to declare a number of abbreviations for formulating a theorem and carrying out a prove. Such local declarations might also be supported.

Abbreviations for formulas are also often helpful to make proofs more transparent.

8.2.4 Supporting the Theory Management

When dealing with formal specifications and verification very soon we have to deal with a large number of units of information, such as the axioms of the domain theories, already verified lemmas, and local assumptions in proofs. This brings two kinds of problems: storing the information (and the different versions) in an adequately structured way and displaying the information during the construction of a proof. Both problems are crucial when dealing with larger examples.

So far in LP the specifications and proofs are stored in files, the file management is done by the user. Axioms are named and can be displayed by name. More associative forms of information retrieval might be helpful.

Of course, it would be more convenient, if all the file management and the storage and retrieval of information in interactive specifications and proofs were taken care of by a more sophisticated LP interface.

The question of revisions and replay of proofs may play a decisive role in developments. At the moment, when some basic axioms in LP specifications are modified, the only thing a user can do is rerun all his/her proofs, identifying problems and fixing them, if possible. More sophisticated support can be imagined here.

8.2.5 Methodological Support

LP provides a specification language and an interactive verification system based on rewriting. So far, it does not provide any method nor even hints how to use such a tool in practical applications.

I believe strongly that for practical applications the question of how to organize the work, and when to specify what and to how much detail, and when to prove what, is decisive. More work and more support is needed here.

The problem of consistency should be taken care of. A method should point out which conditions are to be proved to ensure consistency of a specification under development.

8.3 Lessons Learned

Working with a mechanical interactive proof support system is, in spite of all frustrations, fun. It provides many valuable insights, not only about proof support systems, but also about the organization of proofs and the structuring of specifications.

As it turns out, many theoretical issues are much less relevant than we expected when practically working with a system. Other aspects that are theoretically less significant become more relevant. The feedback obtained by carrying out even moderate size case studies is mandatory for ensuring practical relevance of theoretical work.

More experimentation is badly needed. Only if more case studies and additional practical projects are done in trying out the formal techniques, can it be hoped to bring formal techniques closer to practice in the long run.

Acknowledgements

This work has been carried out during my research stay at DIGITAL Systems Research Center. The excellent working environment and stimulating discussions with the colleagues at SRC, especially Jim Horning, Greg Nelson, and Kevin Jones, are gratefully acknowledged. Thanks go also to Steve Garland and John Guttag who provided with the Larch proof assistant a tool that allows to put a number of theoretical dreams closer to reality.

Appendix A

Appendix: Specifications Used

In the following we give a number of specifications as they have been used in the examples.

A.1 Specification of the Natural Numbers

An LP specification of the natural numbers reads as follows:

```
declare  sort Nat
declare  var i, j, k : Nat
declare  op  0, 1, 2 :           -> Nat
             succ    : Nat       -> Nat
             +, *    : Nat, Nat -> Nat
             =<, <   : Nat, Nat -> Bool
..
assert  Nat generated by 0, succ
assert  ac +
assert  ac *
assert  1 = succ(0)
        2 = succ(1)
        (succ(i) = 0) = false
        (succ(i) = succ(j)) = (i=j)

        (i+0) = i
        (i+succ(j)) = succ(i+j)

        (0 * j) = 0
```

```
        (succ(i) * j) = (j + (i * j))

        (i =< i) = true
        (succ(i) =< i) = false
        (succ(i) =< succ(j)) = (i =< j)

        (i =< j) => (i =< succ(j))
        not(i =< j) => not(succ(i) =< j)

        ((i =< j) & (j =< i)) = (i = j)
        ((i =< j) & (j =< k)) => i =< k

        (i < j) = (succ(i) =< j)

        (i * (j + k)) = ((i * j) + (i * k))

        (i < (i + k)) = (0 < k)
        (succ(i) * j) = (j + (i * j))

        (i =< (i+k))
        (succ(i) < (k + i)) = (1 < k)
        not(0 < i) = (0 = i)
..
```

Some of these axioms are in fact theorems that can be proved, mainly by induction, in a straightforward way.

A.2 Specification of Sequences

The specification of sequences that is used in the example of quicksort reads as follows:

```
declare sort Seq, Nat
declare var  s, r, s1, s2, r1, r2 : Seq
             m, m1, m2 : Nat
..
declare op   es  :           -> Seq
             @   : Nat, Seq -> Seq

             rt  : Seq       -> Seq
             ft  : Seq       -> Nat

             ^   : Seq, Seq -> Seq
```

```
             nbr : Seq      -> Nat

             #   : Nat, Seq -> Nat
..
assert
         es^s -> s
         s^es -> s
         (m@s)^r -> (m@( s^r))

         ((m@s) = (m1@s1)) = (m = m1 & s = s1)
         (es = (m@s)) = false

         (s^r)^s1 = s^(r^s1)
         (s^s1) = (s^s2) == (s1 = s2)
         (s1^s = s2^s) == (s1 = s2)

         nbr(es) = 0
         nbr(m@s) = succ(nbr(s))

         (i#es) = 0
         (i#(j@s)) = if(i = j, succ(i#s), i#s)
..
assert   seq generated by es, @
```

Based on this specification the following theorems were proved using LP.

```
prove   (nbr(s) < succ(0)) == (s = es)
prove   nbr(s) < nbr(m@s)
prove   nbr(r^s) < nbr(r^(m@s))
prove   nbr(m@s) < nbr((m1@s1)) = nbr(s) < nbr(s1)
prove   (i#(s^r)) = ((i#s) + (i#r))
```

These theorems can be proved mainly by induction in a straightforward way.

A.3 Functions Used in the Safety and Liveness Proof

To be able to prove the theorems in the example of the implementation of an interactive queue by a network we introduced a bunch of auxiliary functions and prove some basic properties about them as listed in the following.

```
prove   (nbr(s) < succ(0)) == (s = es)
prove   nbr(s) < nbr(m^s)
```

```
prove   nbr(r^s) = nbr(r)+nbr(s)
prove   nbr(r^s) < nbr(r^(m^s))
prove   nbr(m^ s) < nbr((m1^s1)) = nbr(s) < nbr(s1)
prove   nbr(r^s) < nbr(r^(m^s))

prove   nbrrq(r^s) = nbrrq(r)+nbrrq(s)
prove   nbrrq(r^s) < nbrrq(r^(rq^s))
prove   nbrrq(s) < succ(nbr(s))
prove   nbrrq(r^s) < nbr(r^(rq^s))
prove   nbrrq(r^(dt(i)^s)) = nbrrq(r^s)..

declare op   rt, dhd, rtl, fdm, rdm : stream -> stream
declare op   ft : stream -> message
declare op   fdi : stream -> nat
assert   dhd(es)       = es
         dhd(rq^s)     = es
         dhd(dt(i)^s) = (dt(i)^dhd(s))

         rtl(es)       = es
         rtl(rq^s)     = s
         rtl(dt(i)^s) = rtl(s)

         ft(m^s) = m
         rt(m^s) = s

         fdm(es)       = es
         fdm(dt(i)^s) = (dt(i)^es)
         fdm(rq^s)     = es

         rdm(es)       = es
         rdm(dt(i)^s) = s
         rdm(rq^s)     = (rq^s)

         fdi(dt(i)^s) = i
..
```

For these auxiliary functions the following theorems have been proved. They are used in the correctness proof for the safety properties of the network implementation of queues.

```
prove   s = es | s = (rq^rt(s)) | s = (dt(fdi(s))^rt(s))
prove   (nbrrq(s) < succ(0)) = norq(s)
prove   ((dhd(s) = es) & not(norq(s))) =>
                                   (dt(i)^s) = (dt(i)^(rq^rtl(s)))
```

```
prove   when  ((dhd(s) = es) & not(norq(s)))
        yield (dt(i)^s) = (dt(i)^(rq^rtl(s)))
..
prove   nbr(rtl(s)) < nbr(s) | nbr(rtl(s)) = nbr(s)
prove   (nbr(s) < succ(i)) => nbr(rtl(s)) < succ(i)
prove    br(s) < i => (nbr(dhd(s))+nbr(rtl(s))) < i
prove   ((not(dhd(s) = es) & not(norq(s)) &
        (nbr(s) < succ(i)))) =>
              ((nbr(dt(j)^(dhd(rt(s))^rtl(s)))) < succ(i) = true)
assert  norq(dhd(s)) = true
assert  when   ((not(dhd(s) = es) & not(norq(s))))
        yield  (dt(i)^s) = dt(i)^(dt(j)^(dhd(rt(s))^(rq^rtl(s))))
assert  not(norq(s)) =>
              (dt(i)^s) = (dt(i)^(dhd(s)^(rq^rtl(s))))
assert  when norq(r) yield dhd(r) = r
assert  when not(norq(s)) yield nbr(rtl(s)) < nbr(s)
prove   s = es | nbr(rt(s)) < nbr(s)
```

These theorems can be proved in LP mainly by induction in a straightforward way.

Bibliography

[deBakker et al. 90] J. W. de Bakker, W.-P. de Roever, G. Rozenberg (eds): Stepwise Refinement of Distributed Systems. Lecture Notes in Computer Science 430, Springer 1990

[Boyer, Moore] B. Boyer, J. Moore: MJRTY: A fast majority algorithm. Unpublished

[Bevier 89] W. R. Bevier: Kit and the Short Stack. Journal of Automated Reasoning 5, 1989, 519-530

[Broy 88] M. Broy: Views of Queues. Science of Computer Programming 11, 1988, 65-88

[Broy 90] M. Broy: Functional Specification of Time Sensitive Communicating Systems. REX Workshop. In: [deBakker et al. 90], 153-179

[Cardelli 87] L. Cardelli: Basic Polymorphic Typechecking. Science of Computer Programming 8, 1987, 147-172

[Cardelli, Wegner 85] L. Cardelli, P. Wegner: On Understanding Types, Data Abstraction, and Polymorphism. ACM Computing Survey 1985, 471-523

[CIP 84] M. Broy: Algebraic methods for program construction: The project CIP. SOFSEM 82, also in: P. Pepper (ed.): Program Transformation and Programming Environments. NATO ASI Series. Series F: 8. Berlin-Heidelberg-New York-Tokyo: Springer 1984, 199-222

[Fraus, Hußmann 91] U. Fraus, H. Hußmann: A Narrowing-Based Theorem Prover (Extended Abstract). In: 4th International Conference on Rewriting Techniques and Applications (RTA), Como, Italy, Lecture Notes in Computer Science 488, Springer-Verlag 1991, 435-436

[Garland, Guttag 86] S.J. Garland, J.V. Guttag: An Overview of LP, The Larch Prover. Lecture Notes in Computer Science 355, 1986, 137-151

[Garland, Guttag 90] S.J. Garland, J.V. Guttag, J.J. Horning: Debugging Larch Shared Language Specifications. Digital Systems Research Center, SRC Report 60, July 1990

[Garland, Guttag 91] S.J. Garland, J.V. Guttag: A Guide to LP, The LARCH Prover. Digital Systems Research Center, SRC Report 82, December 1991

[Geser, Hußmann 91] A. Geser, H. Hußmann: Experiences with the RAP System - A Specification Interpreter Combining Term Rewriting and Resolution. In: B. Robinet, R. Wilhelm (Eds.): ESOP 86 Conference Proceedings, Lecture Notes in Computer Science 213, Springer 1986, 339-350

[Guttag, Horning 86a] J.V. Guttag, J.J. Horning: Report on the Larch Shared Language. Science of Computer Programming 6:2, 1986, 103-134

[Guttag, Horning 86b] J.V. Guttag, J.J. Horning: A Larch Shared Language Handbook. Science of Computer Programming 6:2, 1986, 135-157

[Guttag, Horning 93] J.V. Guttag and J.J. Horning (eds.), with S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing: Larch: Languages and Tools for Formal Specification. Springer-Verlag, New York, Texts and Monographs in Computer Science 1993

[Guttag et al. 85] J.V. Guttag, J.J. Horning, J. Wing: An Overview of the Larch Family of Specification Languages. IEEE Software 2:5, 1985, 24-36

[Jones 86] C.B. Jones: Systematic Program Development Using VDM. Prentice Hall 1986

[Hoare 62] C. A. R. Hoare: Quicksort. Computer Journal 5, 1962, 10-15

[Hoare, Foley 71] C. A. R. Hoare, M. Foley: Proof of a Recursive Program: Quicksort. Computing Journal 14:4, 1971, 391-395

[Hußmann 91] H. Hußmann: A Case Study Towards Algebraic Verification of Code Generation. In: T. Rus, C. Rattray (eds.), 2nd Conference on Algebraic Methodology and Software Technology (AMAST), Iowa City, Iowa, USA, May 1991, to appear in Springer Workshops in Computer Science

[London 71] R. London: Correctness of Two Compilers for a LISP Subset. Stanford University, Computer Science Department, CS 240, October 1971

[MacKenzie 91] D. MacKenzie: The Fangs of the VIPER. Nature Vol. 352, 1991, 467-468

[Manasse, Nelson 84] M.S. Manasse, G. Nelson: Correct Compilation of Control Structures.

[Misra, Gries 82] J. Misra, D. Gries: Finding Repeated Elements. Science of Computer Programming 2, 1982, 143-152

[Nipkow 89] T. Nipkow: Term Rewriting and Beyond - Theorem Proving in Isabelle. Formal Aspects of Computing 1, 320-338

[Reeves 91] A. C. Reeves: Towards a Sketch Based Model of Self-Interpreters. University of Stirling, Ph. D. Thesis, Submitted September 1991

[Rus 90] T. Rus: Algebraic Construction of a Compiler. The University of Iowa, Department of Computer Science, Technical Report 90-01, February 1990

[Schieder 92] B. Schieder: Private Communication.

[SPECTRUM 91] The Munich SPECTRUM Group: M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, K. Stølen: The Requirement and Design Specification Language SPECTRUM: An Informal Introduction. Institut für Informatik, Technische Universität München, TUM-I9140, October 1991

[Wirsing 91] M. Wirsing – Algebraic Specification. Handbook of Theoretical Computer Science 1990