Provably Correct Compiler Development and Implementation

Bettina Buth, Karl-Heinz Buth, Martin Fränzle, Burghard v. Karger, Yassine Lakhneche, Hans Langmaack, Markus Müller-Olm *

Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität Kiel
Preußerstr. 1–9, D-2300 Kiel, FRG
E-Mail: procos@informatik.uni-kiel.dbp.de

Abstract. This paper reports on provably correct compiler implementation in the ESPRIT basic research action 3104 ProCoS (Provably Correct Systems). A sharp distinction is drawn between correctness of the specification of a compiler and correctness of the actual implementation. The first covers semantical correctness of the code to be generated, whereas the second concerns correctness of the compiler program with respect to the specification. The compiler construction framework presented aims at minimizing the amount of handcoding during implementation and at reusing specification correctness arguments for proving the implementation correct. The classical technique of bootstrapping compilers is revisited with respect to implementation correctness.

1 Introduction

For an increasing number of applications, software failures may be very costly in terms of economic loss or even human suffering. Examples include control systems for aircrafts, industrial plants, power plants, cars, railways, and weapon as well as for banking and commercial transactions. Consequently, large amounts of money are spent on software dependability. But even with correct source programs, defective development software (such as compilers, assemblers, linkers, and loaders) can result in incorrect machine code.

As such errors cannot be detected by source code inspection, certification institutions insist on examining machine code with disassemblers, decompilers and further analysis tools. As a result, software development and software validation become completely separated. In addition to the problems encountered in the verification of large machine programs, this has a direct effect on the quality of software development, as programmers get less motivated to supply good program documentation. As much more powerful tools for machine program analysis are not in sight, the only way out is the construction of very reliable development software, particularly compilers.

Traditionally, compilers are validated by running a test suite, i.e. by compiling a number of test programs. But clearly, this cannot give sufficient reliability, as e.g. test programs will normally exhibit comparatively simple behaviour in order to allow detection of successful compilation. Mathematical compiler verification would supply a much more rigorous correctness argument. Correctness proofs exist for various parts of compilers written in various formalisms (e.g. [Po81, Yo89]). But verification of a complete

* This work has been partially funded by the Commission of the European Communities under ESPRIT Basic Research Action 3104 ProCoS (Provably Correct Systems).

compiler for implementation of safety-critical software systems is a much more fundamental task: It has to show that some piece of binary code for a real machine is correctly transforming source programs into executable binary code. We are heading for this goal by providing specifications for scanners, parsers, and code generators accompanied by correctness proofs, and also supplying provably correct implementations of these specifications.

2 Separation of Compiler Specification and Implementation

The traditional view of a compiler is that of a program of some programming language transforming source programs to target programs. T-diagrams are a representation of this view. On the other hand, the classical manipulation of T-diagrams shows that there is another view of compilation abstracting from the particular programming language used for compiler implementation. According to this view, the different compilers obtained during a bootstrapping process are considered equivalent, as they do represent the same mapping of source to target programs. This mapping is the crucial aspect of a compiler, and thus also the essential factor of its correctness: A compiler is correct, if it maps source programs to semantically adequate target code.

When formally treating compiler correctness, we are not free to choose which view to take: In the end we have to deliver a concrete compiler program. But in an intermediate step we may take advantage of the more abstract view of compilation, as it permits modularization of compiler development and verification. In a first step, we will describe a relation between source and target programs by purely mathematical means, without recurrence to any compiler implementation language. This process is called *compiler specification*. Its correctness can be stated without considering an implementation: A compiler specification is correct, if and only if the target programs related to a source program adequately reflect the source program's semantics. We will elaborate on the concept of semantical adequacy in section 3.1.

It is only after this compiler specification and specification verification effort that we will go for a *compiler implementation* in some programming language. This division of compiler development in a specification and an implementation task does also modularize our verification efforts: During specification verification we are free from implementation issues. During implementation verification we are no longer concerned with source and target language properties, we only have to show that the semantics of the compiler program matches the relation defined by the compiler specification. We will discuss this in section 2.3.

2.1 Structuring the Specification

We will now sketch the compiler specification method applied in our project. We are heading for a relation between the source and target language that associates semantically adequate target programs to source programs. Thus, our compiler specification defines a relation $Comp \subseteq S_{concr} \times T_{concr}$, where S_{concr} is the set of concrete source programs and T_{concr} the set of concrete target programs, respectively. But there is a problem in defining the correctness of such a specification: Formal semantics is usually associated

with abstract programs, not concrete ones[2]. Consequently, semantical correctness is directly defined for a relation between abstract languages, but not for *Comp*. We solve this by defining $Comp \subseteq S_{concr} \times T_{concr}$ from components $FrontEnd \subseteq S_{concr} \times S_{abstr}$, $CodeGen \subseteq S_{abstr} \times T_{abstr}$, and $Printer \subseteq T_{abstr} \times T_{concr}$, where each part has its own correctness property. In particular, *CodeGen* has a semantically based one.

In our framework, the definition of a language L consists of two parts. First, the abstract language L_{abstr} is defined by a system of recursive equations defining the set of abstract programs, usually tree structures (the equations can be given in e.g. the VDM style of abstract syntax definitions [BJ78]). Concrete programs are strings over an alphabet A, i.e. $L_{concr} \subseteq A^*$. Both the set L_{concr} of concrete L-programs and the relationship between concrete and abstract programs is provided by a total relation $Printer_L \subseteq L_{abstr} \times L_{concr}$ whose converse is a function. $Printer_L$ associates with an abstract L-program all concrete L-programs representing it (and since the converse is a function, no concrete program represents two different abstract ones). The set of concrete programs is implicitely defined by $Printer_L$ as $L_{concr} \stackrel{\text{def}}{=} \{l \in A^* \mid \exists \tilde{l} \in L_{abstr}.(\tilde{l}, l) \in Printer_L\}$.

For definition of $Comp \subseteq S_{concr} \times T_{concr}$ we start from definitions of S and T, i.e. from definitions of S_{abstr}, T_{abstr}, $Printer_S$, and $Printer_T$. This does already fix the specifications of *FrontEnd* and *Printer*: *FrontEnd* is the converse of $Printer_S$ and *Printer* is $Printer_T$. As S_{abstr} is usually tree-structured, it is quite natural to define $CodeGen \subseteq S_{abstr} \times T_{abstr}$ by induction on the structure of S_{abstr}. Although general relations in $S_{abstr} \times T_{abstr}$ can be defined by structural induction in S_{abstr}, we restrict ourselves to inductive function definitions in order to be able to use functional programming languages as a means of prototyping the code generator.

2.2 Correctness of the Specification

The specification of *Comp* is built from the three components *FrontEnd*, *CodeGen*, and *Printer*; two of these (*FrontEnd* and *Printer*) are directly taken from the language definitions. No proof obligation is connected to these two components. The correctness of the code generator specification depends on the semantics of S and T.

Semantics for a language L is given by a mapping $[\![\cdot]\!]_L : L_{abstr} \longrightarrow Sem_L$, where Sem_L is the semantic domain used for describing the meaning of L-programs. In addition to the semantics of S and T we have to supply a relation $implements \subseteq Sem_S \times Sem_T$ telling us wether an element of Sem_T has all the properties of an element of Sem_S that are relevant to us. The relevance of different semantical properties of programs and hence the definition of *implements* depends on the field of application (compare e.g. real-time systems to more time-insensitive applications). Therefore, we will not give an example until we describe the concrete structure of the ProCoS project languages in section 3.

Based on the semantics of both the source and the target language and on the *implements* relation, we say that $CodeGen \subseteq S_{abstr} \times T_{abstr}$ is correct, if and only if $([\![s]\!]_S, [\![t]\!]_T) \in implements$ for each $(s,t) \in CodeGen$. *Compiler specification verification* is the task of verifying this property.

[2] In our terminology, *concrete programs* are strings over some alphabet, whereas *abstract programs* are trees as defined by the abstract syntax of the language.

2.3 Compiler Implementation

The compiling specification itself is just a mathematical relation. In principle, we may use it for translating programs by manually applying its definition to source code. In practice, however, we need an executable version on a real machine for successful application. I.e. we have to provide a compiler program written in a compiler implementation language CIL that implements the compiler specification.

What does it mean for a CIL program to implement the compiler specification? First of all, CIL has to supply the means to input concrete S programs and to output concrete T programs. As no further input or output is required, the semantic domain of CIL is a subset of the partial functions from *Input* to *Output*, where $\mathit{Input} \supseteq S_{concr}$ and $\mathit{Output} \supseteq T_{concr}$. Furthermore, we assume that source programs given to our compiler will be verified, as we are dealing with safety-critical applications, and thus source programs are syntactically correct. This implies that we need not deal with input strings that are not concrete S-programs. We then say that a CIL program c implements *Comp*, if we have $(s, [\![c]\!]_{\mathrm{CIL}}(s)) \in \mathit{Comp}$ for any $s \in S_{concr}$ with $[\![c]\!]_{\mathrm{CIL}}(s)$ being defined. I.e. c maps source programs to target programs that are permitted by the compiler specification *Comp*. This is the only proof obligation of *compiler implementation verification*; the semantical adequacy of the resulting target code is already guaranteed by compiler specification verification, i.e. by correctness of the code generator part of *Comp*.

2.4 Compiler Execution

A compiler implementation in CIL is still not sufficient for compiling programs. Applying the compiler to source programs requires an execution mechanism for CIL. This mechanism is not taken into account in the compiler implementation verification, instead, implementation correctness is defined with respect to the abstract semantics of CIL. But in absence of verified compilers and interpreters, the execution mechanism for CIL may not agree with the formal semantics and thus, executing the compiler on the CIL execution mechanism may yield erroneous results although the compiler implementation has been verified. This brings us back at our starting point: In order to obtain a reliable S-to-T compiler, we need a proven CIL compiler or interpreter. Redoing our work for CIL would push the problem ahead, leaving us with the need for a verified implementation of yet another compiler implementation language. But we may implement a CIL compiler in CIL itself and bootstrap it. To get things going, we have to have some initial execution mechanism for CIL. If we do not want to verify it, we have to control the influence of this mechanism on the correctness of the bootstrapped compilers. Different initial execution mechanisms and means to control their results will be discussed in section 7.

3 ProCoS Project Languages

The objectives of the ESPRIT basic research action ProCoS are to advance the state-of-the-art of systematic design of complex heterogenous systems, including both software and hardware; in particular, to reduce the risk of errors in the specification, design and implementation of embedded safety-critical systems. A development method is built around a system of three languages: A language for specification of reactive systems, an occam-like programming language PL, and a transputer-like machine language ML.

In this article we are dealing with the problem of how to build a proven correct compiler for the above mentioned programming language PL. The compiler is to run on the transputer and will generate ML programs, namely transputer machine code. The reason for running the compiler on its own target machinery is reduction of hardware verification effort: With only one machine architecture involved, it is sufficient to verify only that architecture for guaranteeing the correctness of both the execution of the compiler and of the generated target code.

The programming language PL is a small subset of occam [i88b]. It features integers and arrays of integers as data types. Programs are built from the processes `SKIP`, `STOP`, assignment, communication via two synchronous external channels connecting the program to its environment, sequential composition, conditional execution, and loops. As compiler implementation directly in machine language would be tiresome and hardly verifiable, PL is extended beyond occam by adding recursive procedures without parameters. This makes compiler implementation in PL feasible, and compilers written in PL can be transferred to the machine with the PL-to-ML compiler specification.

The target language ML is a subset of the 32 bit transputer's binary code [i88a]. Again, it features synchronous communication with the environment via two external channels.

As PL is a rather small language with simple data types, compiler writing remains tedious even with PL as a compiler implementation language. To ease this process, we supply a language which matches the inductive function definition style of our compiler specifications more directly. This language, called SubLISP, is a simple first-order functional language. SubLISP programs are systems of mutually recursive, non-nested function definitions treating a subset of LISP s-expressions as data. SubLISP programs can be executed in a Common-LISP environment with only minor changes, so there is a means of rapid prototyping for compiler implementations in SubLISP. A similar link exists to Boyer-Moore logic making mechanical verification of SubLISP programs in the Boyer-Moore theorem prover feasible [BM88].

In order to obtain a compiler implementation running on the transputer from an implementation in SubLISP, we have to translate SubLISP to ML. To ease specification and verification of that translation, we will provide a compiler specification and its verification from SubLISP to PL, and will reuse the compiler specification of PL to ML to get down to machine level.

3.1 Semantics and Correctness Concepts

Compiling PL. PL, as an occam-like language, provides a clear distinction between aspects of program execution relevant to the environment and internal aspects. Externally visible and thus relevant to the environment is only the behaviour of external communication channels. All other aspects including the state of the program variables and the point of execution are internal and thus invisible. Hence a program can be replaced by another with a completely different internal structure, if both share the relevant aspects of the behaviour on external channels.

The semantics of PL as well as of ML can be adequately described by an operational formalism that shares the distinction between internal and globally visible aspects of execution. This formalism, called *labelled transition systems*, distinguishes between configurations which represent the internal, invisible aspects of computation and communications over a communication alphabet which are the externally visible aspects.

Definition 3.1 [Labelled Transition System]
A labelled transition system S is a quadrupel $S = (\Gamma_S, T_S, \Lambda_S, \rightarrow_S)$, where
- Γ_S is the set of configurations,
- Λ_S is the set of labels, with $\tau \in \Lambda_S$,
- $\rightarrow_S \subseteq \Gamma_S \times \Lambda_S \times \Gamma_S$ is the transition relation, and
- $T_S \subseteq \Gamma_S$ is the set of terminal configurations, satisfying $(\gamma, \lambda, \gamma') \in \rightarrow_S \Rightarrow \gamma \notin T_S$.

We will write $\gamma \xrightarrow{\lambda}_S \gamma'$ to denote $(\gamma, \lambda, \gamma') \in \rightarrow_S$.

The interpretation of a labelled transition system S is that the system described by S in a configuration γ is allowed to go to configuration γ' under an externally visible communication $\lambda \neq \tau$, iff $\gamma \xrightarrow{\lambda}_S \gamma'$, or silently, iff $\gamma \xrightarrow{\tau}_S \gamma'$. If more than one transition is permitted by this rule, the system may select any of these. It has to perform a transition within some unspecified, but finite amount of time, if one is possible (i.e. it must not refuse selection of a transition).

For the semantics of both PL and ML, the set of communication labels is $\Lambda = \{\tau\} \cup \{\mathtt{in}.v \mid v \text{ a 32 bit integer}\} \cup \{\mathtt{out}.v \mid v \text{ a 32 bit integer}\}$ which can be interpreted as values sent along two channels `in` and `out` capable of transmitting 32 bit words. A slightly simplified example of a rule defining the operational semantics of PL is

$$< \mathtt{INPUT?}name, \sigma > \xrightarrow{\mathtt{in}.v}_{SeqProc} < \mathtt{terminated}, \sigma[name \mapsto v] >$$

which means that a process of the form `INPUT?`*name* can in state σ of the program variables evolve under any communication of the form `in`.v, where v is a 32 bit integer, into a regularly terminated process with the state being modified such that *name* now takes the value v. Another example is

$$\frac{< exp, \sigma > \longrightarrow_{Expr} \mathtt{true}}{< \mathtt{WHILE}[exp, sproc], \sigma > \xrightarrow{\tau}_{SeqProc} < \mathtt{SEQ}[sproc, \mathtt{WHILE}[exp, sproc]], \sigma >}$$

which means that the while loop will invisibly unfold to the sequential composition of its body and the loop itself without a state change if the guarding expression evaluates to true.

It should be clear from these examples that labelled transition systems give a very fine-grain description of program execution together with a basic distinction between externally visible and invisible aspects of computation. But still, not all the externally visible aspects need to be relevant to our problem. Following the tradition of CSP [Ho85], we say that a program $\tilde{p}$ can safely replace a program p, i.e. preserves the relevant properties of p, if

1. $\tilde{p}$ has less traces than p, i.e. p can engage in any communication sequence that $\tilde{p}$ can exhibit,
2. $\tilde{p}$ has less failures than p, i.e. if $\tilde{p}$ may fail to engage in a communication λ after exhibiting a certain communication sequence, then p may also do so, and
3. $\tilde{p}$ has less divergences than p, i.e. if $\tilde{p}$ may engage in infinite internal behaviour after exhibiting a certain communication history, then p may do so either.

In that case we say $\tilde{p}$ *is a CSP-refinement of* p and denote this by $p \sqsubseteq \tilde{p}$.

Ideally, we would require the PL-to-ML compiler to map source programs to CSP-refinements thereof, as these can safely replace the source programs. Thus, the compiler

specification should satisfy $CodeGen_{\mathrm{PL}} \subseteq implements$ with $implements = \{(p, m) \in \mathrm{PL} \times \mathrm{ML} \mid p \sqsubseteq m\}$. Unfortunately, recursion in PL allows for programming of infinite state processes, whereas our target machinery is finite. In general, infinite state processes cannot be CSP-refined by finite state processes. Therefore, PL programs p exist that have no ML program m with $p \sqsubseteq m$. Such programs are not compilable under CSP-refinement as correctness notion of the compiler. In analogy to *partial correctness* of non-communicating programs, we defined *partial refinement* [Fr90] to circumvent the problem. Like partial correctness allows for irregular termination of implementations of semantically well-behaving programs, partial refinement allows for recovery actions upon stack overflow. This correctness notion is established for $CodeGen_{\mathrm{PL}}$ using *operational failure approximation* [Fr90] which is an operationally based simulation method developed for verification of CSP-refinement and related semantical properties.

Due to weaknesses in the treatment of visible communications in the CSP model when divergence is present, partial refinement is not completely sufficient for correctness arguments of compilers implemented in PL. We need to be sure that the code associated with a PL program by $Comp_{\mathrm{PL}}$ can only show sequences of external communications that the source program could exhibit. To be precise we define

Definition 3.2 [Operational communication sequences]
Let $S = (\Gamma_S, T_S, \Lambda_S, \rightarrow_S)$ be a labelled transition system, and $\gamma \in \Gamma_S$. The *operational communication sequences* of γ are

$$OpComSeq(\gamma) \stackrel{\text{def}}{=} \{tr \in (\Lambda_S \setminus \{\tau\})^* \mid \exists \gamma' \in \Gamma_S . \gamma \stackrel{tr}{\rightarrow}{}^*_S \gamma'\}$$

where $\gamma \stackrel{tr}{\rightarrow}{}^*_S \gamma'$ holds, if and only if there are $\lambda_1, \ldots, \lambda_n \in \Lambda_S$ and $\gamma_0, \ldots, \gamma_n \in \Gamma_S$ for some $n \in \mathbf{N}$ with $\gamma_0 = \gamma$, $\gamma_n = \gamma'$, $\gamma_{i-1} \stackrel{\lambda_i}{\rightarrow}_S \gamma_i$ for any $i \in \{1, \ldots, n\}$, and $\overline{\lambda_1 \ldots \lambda_n} = tr$, where $\overline{\lambda_1 \ldots \lambda_n}$ is the string over $\Lambda_S \setminus \{\tau\}$ obtained by removing every τ from $\lambda_1 \ldots \lambda_n$.

In addition to partial refinement, operational failure approximation proves

Theorem 3.3 [Correctness of $Comp_{\mathrm{PL}}$ wrt. communication sequences]
Let p be a PL and m be an ML program with $(p, m) \in CodeGen_{\mathrm{PL}}$. For any PL store σ we have $OpComSeq(< p, \sigma >) \supseteq OpComSeq(< m, \bar{\sigma} >)$, where $\bar{\sigma}$ is a representation of σ as an ML store.

Compiling SubLISP. The semantics of SubLISP is given as a direct denotational semantics ([St77]). This is natural since SubLISP is a simple functional language. The domain of denotable values is $D = Sexpr \uplus \{\bot\}$, where $Sexpr$ is the set of binary trees over integers and $\bot$ denotes the undefined value. The denotational semantics assigns strict functions from D^i to D to function definitions of arity i. As the main part of a SubLISP program is a call of a function of arbitrary arity, the semantic domain for programs is $\dot{+}_{i \in \mathbf{N}} (D^i \stackrel{strict}{\rightarrow} D)$.

Compilation of SubLISP to PL is defined via an intermediate language SIL (short for *stack intermediate language*) which is later removed from the compiler specification by composing the SubLISP-to-SIL and the SIL-to-PL compiler specification. The purpose

of introducing SIL is modularization of the compiler specification verification for SubLISP. SIL is a state-based language handling a state consisting of exactly one stack of s-expressions. A denotational semantics of SIL is given, and the compiler specification is verified by fixpoint induction. SIL-to-PL compilation is also verified by fixpoint induction against a denotational semantics of PL. Each of the fixpoint inductions requires a detailed investigation of each programming language construct and is therefore rather lengthy. Since argumentation appears to be rather systematic we hope that mechanical theorem provers can help to make proofs of this kind more tractable. Details of the SubLISP-to-PL compiler specification and its verification can be found in [MO90]. Here we only give a corollary from the correctness theorem that is sufficient for proven correct implementation of compilers with the aid of SubLISP. $\mathcal{P}$ denotes the semantic function for PL programs and $\mathcal{S}$ denotes the semantic function for SubLISP programs. For a PL program p, $Trace(\mathcal{P}[\![p]\!](\sigma, i))$ is the communication sequence exhibited by p when started with store σ and fed with input i. $CodeGen_{\mathrm{SL}}$ is the compiling function that specifies how SubLISP programs are translated to PL. Because a SubLISP program maps s-expressions to s-expressions whereas a PL program essentially communicates with its environment via channels, we must fix a convention how sequences of communication values are represented by s-expressions. $rep_{io} : io^* \rightarrow Sexpr$ is a function recording this convention, where io is a proper subset of the 32 bit integers and can thus be communicated along the PL and ML channels. rep_{io}, which is a injective function, is the first example of a representation relation that we meet. In general, a *representation relation* is a relation which has a surjective function as its converse. As a counterpart to representations, surjective functions are sometimes called *abstraction functions.*

Theorem 3.4 [Correctness of SubLISP-to-PL compilation]
Suppose s is a SubLISP-program of arity one, $a, b \in io^*$, σ an arbitrary PL store, and $tr = Trace(\mathcal{P}[\![CodeGen_{\mathrm{SL}}[\![s]\!]]\!](\sigma, a^\frown < eof >))$. If

1. $rep_{io}(b) = \mathcal{S}[\![s]\!](rep_{io}(a))$ and
2. the communication sequence tr contains $\mathbf{out}.eof$,

then $tr = \mathbf{in}.a^\frown < \mathbf{in}.eof >^\frown \mathbf{out}.b^\frown < \mathbf{out}.eof >$, where $\mathbf{in}.a$ is an abbreviation for $< \mathbf{in}.a_1, \ldots, \mathbf{in}.a_n >$ when $a =< a_1, \ldots, a_n >$, similarly $\mathbf{out}.b$.

The theorem contains a partial correctness statement for the generated code. Condition (1) says that we only guarantee a correct output of the target program, if the result of the source program represents a sequence of communication values. Other values cannot be printed out anyway. (1) does not restrict the usefulness of the result because we will apply SubLISP translation only to verified programs having this property for every input. In particular compiler programs always yield concrete target programs, i.e., representations of character sequences. The need for condition (2) comes from the fact that a PL program has only access to a finite amount of memory, whereas execution of a SubLISP program requires infinite memory in general, since recursion depth is not bounded. Therefore we must provide a way of telling the observer that the computation has been finished succesfully. This is done by emitting the special symbol $eof \notin io$ on the output channel when the calculation terminates and the result has been completely delivered.

In order to interface the correctness arguments of the SubLISP-to-PL compiler specification to those of the PL-to-ML compiler specification, we have to relate the different semantical descriptions of PL used in the different compiler specification verifications. It is only then that we can claim to have a verified compiler specification of SubLISP

down to ML. Thus, a denotational semantics of PL and the labelled transition system of PL have to be proved equivalent. The denotational meaning of a PL program p is a function $\mathcal{P}[\![p]\!]$ that takes a store σ recording the stored values and an input stream as arguments. It yields a store and a communication trace if the program terminates regularly. For non-terminating programs, it just yields a communication trace, which is finite if the program is divergent.

The equivalence proof of the operational and the denotational semantics [La91] shows that both semantics assign the same communication and termination behaviour to PL programs. For the verification of $CodeGen_{\mathrm{SL}}$ it is sufficient to claim that the communication behaviours assigned by the operational and the denotational semantics are adequate, i.e. for an arbitrary PL program p and store σ we have

$$(\exists \gamma' \in \Gamma_{\mathrm{PL}}. < p, \sigma > \stackrel{tr}{\longrightarrow}^*_{\mathrm{PL}} \gamma') \iff tr \prec Trace(\mathcal{P}[\![p]\!](\sigma, tr \downarrow \mathtt{in}))$$

where tr is a communication sequence, $tr \downarrow \mathtt{in}$ is its projection on the input channel, and $\prec$ denotes the prefix order on strings.

Defining the denotational communication sequences of a PL program p and a PL store σ as

$$DenComSeq(< p, \sigma >) \stackrel{\mathrm{def}}{=} \{tr \in (\Lambda \setminus \{\tau\})^* \mid tr \prec Trace(\mathcal{P}[\![p]\!](\sigma, tr \downarrow \mathtt{in}))\}$$

we can rephrase the former correctness property to

Theorem 3.5 [Operational and denotational communication sequences of PL]
For any PL program p and any PL store σ we have

$$OpComSeq(< p, \sigma >) = DenComSeq(< p, \sigma >)$$

4 Interpreting the Project Languages as Compiler Implementation Languages

All programming languages of ProCoS are defined such that the concrete languages are subsets of io^*. In order to interpret different programs written in SubLISP, PL, and ML as compilers, we have to interpret them as relations in $io^* \times io^*$. Therefore, we will semantically embed SubLISP, PL, and ML in the domain of such relations. To enhance readability, we will use infix notation for binary relations (sometimes even for functions), i.e. instead of $(m, n) \in R$ we will write mRn, and we denote relational composition[3] by a semicolon.

We embed SubLISP programs of arity one by

Definition 4.1 [SubLISP programs as input/output relations]
For a SubLISP program s, its embedding $\lceil s \rceil \subseteq io^* \times io^*$ is the relation

$$\lceil s \rceil \stackrel{\mathrm{def}}{=} rep_{io}\,; \mathcal{S}[\![s]\!]\,; {rep_{io}}^{-1}$$

To obtain an embedding for PL and ML programs, we associate with any set of communication sequences an input/output relation encoded by the sequences:

[3] For two relations $R \subseteq M \times N$ and $S \subseteq N \times O$, the relational composition of R and S is $R;S \stackrel{\mathrm{def}}{=} \{(m, o) \in M \times O \mid \exists n \in N.\, mRn \wedge nSo\}$.

Definition 4.2
For any $T \subseteq (\Lambda \setminus \{\tau\})^*$, $R(T) \subseteq io^* \times io^*$ is the relation defined by $(a,b) \in R(T)$ if and only if there is $tr \in T$ with $tr \downarrow \mathbf{in} \prec a\hat{} < eof >$, and $tr \downarrow \mathbf{out} \succ b\hat{} < eof >$.

This enables us to assign relations in $io^* \times io^*$ to PL and ML configurations, with two different ways to embed PL based on its denotational and its operational semantics:

Definition 4.3 [PL and ML configurations as input/output relations]
Let p be a PL program, m an ML program, Σ the set of PL stores, and $\overline{\Sigma}$ the set of ML stores. We define $\lceil p \rceil, \lfloor p \rfloor, \lfloor m \rfloor \subseteq io^* \times io^*$ by

$$\lceil p \rceil \stackrel{\text{def}}{=} R(\bigcup_{\sigma \in \Sigma} DenComSeq(< p, \sigma >)),$$

$$\lfloor p \rfloor \stackrel{\text{def}}{=} R(\bigcup_{\sigma \in \Sigma} OpComSeq(< p, \sigma >)), \text{ and}$$

$$\lfloor m \rfloor \stackrel{\text{def}}{=} R(\bigcup_{\bar{\sigma} \in \overline{\Sigma}} OpComSeq(< m, \bar{\sigma} >)).$$

Theorem 3.5 shows that there is no need to distinguish $\lceil p \rceil$ and $\lfloor p \rfloor$ as they are always equal. And theorem 3.3 implies $\lfloor p \rfloor \supseteq \lfloor CodeGen_{\mathrm{PL}} [\![p]\!] \rfloor$ for any PL program p. With the embeddings, we can also give theorem 3.4 a new representation:

Corollary 4.4
Let s be a SubLISP program and σ be an arbitrary PL store, let $a, b, b' \in io^*$. If $a \lceil s \rceil b$ and $a \lceil CodeGen_{\mathrm{SL}} [\![s]\!] \rceil b'$, then $b = b'$.

5 Implementation Relations

In the previous section, we have already met three different relationships between relations in $io^* \times io^*$: Inclusion (between $\lfloor p \rfloor$ and $\lfloor CodeGen_{\mathrm{PL}} [\![p]\!] \rfloor$), equality (between $\lceil p \rceil$ and $\lfloor p \rfloor$), and the more complicated relationship of corollary 4.4. We will now start a more systematic investigation of these relationships. Let $r, \bar{r} \subseteq io^* \times io^*$. We say

- r *refines* $\bar{r}$, abbreviated $r \subseteq_{io} \bar{r}$, if and only if $r \subseteq \bar{r}$,
- r *completes* $\bar{r}$, abbreviated $r \supseteq_{io} \bar{r}$, if and only if $r \supseteq \bar{r}$ and r restricted to the domain of $\bar{r}$ is a function[4] (note that $r \supseteq_{io} \bar{r}$ is not equivalent to $\bar{r} \subseteq_{io} r$), and
- r *weakly equals* $\bar{r}$, abbreviated $r \mathrel{\bigcirc\!\!\!=}_{io} \bar{r}$, if and only if $a\, r\, b$ and $a\, \bar{r}\, \bar{b}$ implies $b = \bar{b}$.

Obviously, we have

Lemma 5.1
1. $h \mathrel{\bigcirc\!\!\!=}_{io} g \supseteq_{io} f \Rightarrow h \mathrel{\bigcirc\!\!\!=}_{io} f$, i.e. a weak equivalent of a completement is still a weak equivalent, and
2. $h \subseteq_{io} g \mathrel{\bigcirc\!\!\!=}_{io} f \Rightarrow h \mathrel{\bigcirc\!\!\!=}_{io} f$, i.e. a refinement of a weak equivalent is still a weak equivalent.

With the new relations, we can reformulate corollary 4.4, theorem 3.5, and theorem 3.3 to

[4] r restricted to the domain of $\bar{r}$ is the relation $\{(a,b) \in r \mid \exists b' \in io^*.a\, \bar{r}\, b'\}$.

Theorem 5.2

Let s be a SubLISP program, p a PL program.

1. $\lceil CodeGen_{\mathrm{SL}}\,[\![s]\!]\rceil \supseteq_{io} \lceil s\rceil$, i.e. $\lceil CodeGen_{\mathrm{SL}}\,[\![s]\!]\rceil$ weakly equals $\lceil s\rceil$,
2. $\lfloor p\rfloor = \lceil p\rceil$, and
3. $\lfloor CodeGen_{\mathrm{PL}}\,[\![p]\!]\rfloor \subseteq_{io} \lfloor p\rfloor$, i.e. $\lfloor CodeGen_{\mathrm{PL}}\,[\![p]\!]\rfloor$ refines $\lfloor p\rfloor$,
4. and thus $\lfloor CodeGen_{\mathrm{PL}}\,[\![CodeGen_{\mathrm{SL}}\,[\![s]\!]]\!]\rfloor \supseteq_{io} \lceil s\rceil$.

Property (4.) states that SubLISP programs are mapped to weak equivalents by the complete SubLISP-to-ML translation. Spending a little more effort on SubLISP implementation we could gain refinement in (1.) and thus could also obtain refinement in (4.). Although this would make our reasoning for the compiler bootstrap simpler, it is not strictly necessary, as the following theorems demonstrate.

6 Implementing Compilers in SubLISP

As compilers are relations in $io^* \times io^*$, we have to define what it means for a SubLISP program to implement such a relation: If f is a relation in $io^* \times io^*$ and s a SubLISP program, then we say *s is an implementation of f*, if and only if $\lceil s\rceil \supseteq_{io} f$. Note that this requires f to be a function.

With this definition of implementation we get as a direct consequence of lemma 5.1 and theorem 5.2:

Theorem 6.1 [Compilation of implementations]

Let $f \subseteq io^* \times io^*$, s a SubLISP implementation of f (i.e. $\lceil s\rceil \supseteq_{io} f$), and $m = CodeGen_{\mathrm{PL}}\,[\![CodeGen_{\mathrm{SL}}\,[\![s]\!]]\!]$. Then $\lfloor m\rfloor \supseteq_{io} f$ holds, i.e. completements are transformed to weak implementations by the complete SubLISP-to-ML compilation.

By now, we have only defined how to implement relations in $io^* \times io^*$ in SubLISP. Although this is sufficient to implement complete compilers for a language with a concrete syntax over the alphabet io, we would prefer to do stepwise implementation of the components *FrontEnd*, *CodeGen*, and *Printer*. As these are not normally relations in $io^* \times io^*$, their implementation is not covered by our formal notion of implementation. In order to deal formally with their implementation in SubLISP, we have to treat abstract syntax trees as abstract data types and have to consider their implementation by s-expressions. But as this is standard data reification, we will not go into the details of that and refer to e.g. [Jo90]. More details may be found in [Bj92].

7 Bootstrapping Compilers

Starting from compiler specifications and their implementations in SubLISP, theorem 6.1 gives us a possibility to obtain compiler implementations in ML:

Theorem 7.1 [Manual bootstrap]
Suppose that
1. S and T are languages with $S_{concr}, T_{concr} \subseteq io^*$,
2. $Comp \subseteq S_{concr} \times T_{concr}$ is a total relation (the compiler specification),
3. c is a SubLISP implementation of $Comp$,
4. $m = CodeGen_{\mathrm{PL}}\,[\![CodeGen_{\mathrm{SL}}\,[\![c]\!]\,]\!]$, and
5. $sp \in S_{concr}$.

If $sp\ \lfloor m \rfloor\ tp$ then $sp\ Comp\ tp$.

This means we can obtain a correct compiler implementation in ML, i.e. running on the transputer, by implementing the compiler in SubLISP and manually applying the code generator specifications for SubLISP and PL to it. But, obviously, manual application of these specifications to a compiler implementation is intractable due to the size of both the specifications and the implementation.

Thus, we may decide to take advantage of some host machine supplying an execution mechanism for a language sufficiently close to SubLISP to allow for simple conversion of the compiler implementation, e.g. a Common-LISP system. Abstractly, the host language system consists of a host language HL, an input/output alphabet IO (normally different from io), a representation $rep_{HL} \subseteq Sexpr \times IO^*$ that prescribes the way s-expressions should be represented as strings, and a semantic mapping $\mathcal{H}$ which associates with every program $p \in HL$ a relation $\mathcal{H}\,[\![p]\!]$ in $IO^* \times IO^*$. To relate HL programs to specifications or SubLisp programs, we must embed them in $io^* \times io^*$. We do this by letting $\lceil p \rceil = rep_{io}; rep_{HL}; \mathcal{H}\,[\![p]\!]; rep_{HL}^{-1}; rep_{io}^{-1}$ for $p \in HL$. An embedding of SubLISP into the host language is a function E from SubLISP in HL. The host system is called *correct* wrt. E, if and only if $\lceil E\,[\![s]\!]\,\rceil \supseteq_{io} \lceil s \rceil$ for any $s \in \mathrm{dom}\,E$. The analogue to the manual boostrap theorem is

Theorem 7.2 [Bootstrap on a host system]
Suppose that
1. the host system is correct wrt. E,
2. s is a SubLISP implementation of $Comp_{\mathrm{SL}}; Comp_{\mathrm{PL}}$, and $s \in \mathrm{dom}(E)$ (where $Comp_{\mathrm{SL}}$ is $Printer_{\mathrm{SL}}^{-1}; CodeGen_{\mathrm{SL}}; Printer_{\mathrm{PL}}$ and similarly $Comp_{\mathrm{PL}}$ is $Printer_{\mathrm{PL}}^{-1}; CodeGen_{\mathrm{PL}}; Printer_{\mathrm{ML}}$),
3. $Comp \subseteq S_{concr} \times T_{concr}$ is a total relation (the compiler specification),
4. c is a SubLISP implementation of $Comp$,
5. $m \in \mathrm{ML}_{abstr}$ with $c(Printer_{\mathrm{SL}}; \lceil E\,[\![s]\!]\,\rceil; Printer_{\mathrm{ML}}^{-1})m$, and
6. $sp \in S_{concr}$.

If $sp\ \lfloor m \rfloor\ tp$ then $sp\ Comp\ tp$.

That means that we can obtain a correct compiler implementation on the transputer by bootstrapping c with the SubLISP compiler on a correct host system. But in practice, we do not know wether the last theorem is applicable, as we cannot be sure of the correctness of the host system. We propose to use one that might contain errors. Still it is interesting to determine what will happen, if it is correct. Even if we cannot this way achieve a proof for the resulting compiler implementation, we get at least a chance to detect errors. A powerful test we can perform on a result obtained with a host interpreter is described by the next theorem:

Theorem 7.3 [Bootstrap self-test]

1. If the host system is correct (wrt. E),
2. s is a SubLISP implementation of $Comp_{\mathrm{SL}}$; $Comp_{\mathrm{PL}}$, and $s \in \mathrm{dom}(E)$,
3. $s\ Printer_{\mathrm{SL}}\ i$,
4. $m \in io^*$ with $i\ \lceil E\,[\![s]\!]\ \rceil\ m$, and
5. $i\ \lfloor Printer_{\mathrm{ML}}^{-1}(m) \rfloor\ \bar{m}$,

then $m = \bar{m}$, i.e. m and $\bar{m}$ are the same strings in io^*.

Let us now turn to the question that a safety engineer would pose: How likely is it that the compiler implementation is correct, given that the host interpreter may be faulty but the bootstrap self-test has succeeded? This can be answered with the aid of Bayes' formula for the probability of cause:

Denote by X the event that the host interpreter correctly performs the bootstrap and by $\neg X$ its complement. The a priori probability of X is the chance that a program will be interpreted correctly. Let $\varepsilon \stackrel{\mathrm{def}}{=} P(\neg X)$. Denote by T the event that the bootstrap self-test succeeds. We have the following conditional probabilities:

- $P(T|X) = 1$. If the host interpreter makes no error, then the bootstrap test will surely succeed (cf. theorem 7.3) because the compiler specification and its implementation in SubLISP have been verified.
- $P(T|\neg X) \stackrel{\mathrm{def}}{=} \delta$. It is possible that the bootstrap test succeeds even though the host interpreter produces a wrong result.

What we are interested in is $P(X|T)$, the a posteriori probability that the host interpreter has produced the correct result, given that we have seen success of the bootstrap test. It is given by

$$P(X|T) = \frac{1-\delta}{1-\delta+\delta\varepsilon} \approx 1 - \delta\varepsilon.$$

(The approximation is valid if both δ and ε are small).

What can we say about δ? Suppose that the host interpreter fails to produce a correct compiler. Most likely, the result will be incorrect in an obvious way (the output is not an ML program, or one that exhibits random behaviour). But there is a slight possibility that the output "looks" correct and correctly translates most programs (otherwise we will spot the mistake quickly). Then it may correctly translate its own source code. In this case the comparison will still fail, since the first version is incorrect, but not the second. Alternatively, it may make some arbitrary error when fed with its own source. There is no reason to suppose that this will be the same one the host interpreter made in the first place and if it is not, the bootstrap test will still fail.

The chance that the test succeeds under these circumstances is analogous to the probability that a randomly chosen Turing machine reproduces its own description. The chances against this must be astronomical. Thus δ is very small.

On the other hand, ε is the chance that a program will not be interpreted correctly and can be established by testing the host interpreter. Notice that the reliability of our bootstrapped compiler is significantly increased over the reliability of the host system. The likelihood ε of erroneous program execution on the host system is comparatively large, as no more than some thousands of tests can be done when developing such a system. But the chance of an error in our compiler implementation obtained by bootstrapping is only $\delta\varepsilon$ with δ being a very small factor. If this is still not sufficient, intermediate results

of the bootstrap (such as abstract program trees, compiletime environments, and further compiler internals) can be printed and manually inspected during the bootstrap for further reduction of the probability of errors. Consequently, there is a very good chance that a compiler implementation obtained with the aid of a host interpreter is correct.

More Bootstrap Tests. Clearly, we are not limited to bootstrapping compilers. As well, we could starting from a verified interpreter specification for SubLISP implement a SubLISP interpreter in SubLISP, and can then apply boostrapping on a host system with a SubLISP implementation of $Comp_{\mathrm{SL}}; Comp_{\mathrm{PL}}$ to obtain a SubLISP interpreter in ML. This interpreter can then be used on the transputer instead of the host interpreter on some host machine to perform further bootstrap steps. Doing this, we can increase our confidence in the results obtained on the host system by doing a variety of bootstrap tests.

Suppose we possess SubLISP implementations c and i (in concrete SubLISP) of a SubLISP compiler and a SubLISP interpreter. Denote by $\bar{c}$ and $\bar{i}$ corresponding (concrete) ML implementations obtained by application of $Comp_{\mathrm{SL}}$, i.e.

$$c\ Comp_{\mathrm{SL}}\bar{c} \qquad \text{and} \qquad i\ Comp_{\mathrm{SL}}\bar{i}.$$

There are at least four ways to calculate $\bar{c}$ and $\bar{i}$ with the aid of a correct host interpreter. If an imperfect host interpreter is used and all four results agree, there are very strong grounds indeed for believing in the correctness of the result.

Four Methods to compile c and i

1. Use $E[\![c]\!]$ interpreted by the host interpreter for compilation of c and i. After decoding the output representation of the host system, this yields a compiler $\bar{c}_1$ and an interpreter $\bar{i}_1$ in concrete ML.
2. Run $\bar{c}_1$, i.e. the compiler produced by the first method, on the transputer to get $\bar{c}_2$ and $\bar{i}_2$.
3. Use $E[\![i]\!]$ interpreted by the host system to interpret c. Supplied with c as input, this gives (after decoding), a compiler $\bar{c}_3$ in concrete ML. Similarly, i can be compiled to $\bar{i}_3$.
4. Run c interpreted by $\bar{i}_1$ on the transputer to compile c and i to $\bar{c}_4$ and $\bar{i}_4$.

With four results we can make three tests for equality. These tests seem to be logically (though probably not stochastically) independent in the sense that the success of any two of them does not imply success of the third.

8 Discussion

Our goal was a complete correctness proof for the specification and implementation of a compiler. Although we have not been able to solve this task completely, we have been successful in designing a framework of subgoals, interfaces and proof obligations that permits such proofs to be done in principle. The most difficult proof obligations arising for our example language have been discharged, including a denotational correctness proof for the compilation from SubLisp to PL, an operational correctness proof for the compilation from PL to transputer code, and a proof that denotational and operational

semantics of PL are equivalent. Furthermore, we have shown how the transition from compiler specification to compiler implementation can benefit from the use of a functional implementation language. Now that we have constructed a provably correct compiler for the implementation language, we can carry out future compiler developments at a higher level of abstraction without giving up mathematical rigour.

To demonstrate the feasibility of our method we have developed prototype compilers for SubLisp and for PL and an interpreter for SubLisp. All of them are running under CommonLisp on Sun workstations. The bootstrap yielding transputer implementations of these is currently being undertaken.

In the current paper, we have only given an overview of our results in order to give an idea of the interdependency of the correctness of compiler specifications and their implementations in different formalisms. We left out examples of language definitions, semantics, compiler specifications, and particularly all the proofs. A much more detailed description of the project can be found in [Bj92]. Part 3 gives a rather complete account of the compiler development work, and does also comment on missing links. The universitary environment we are working in could only supply enough manpower for all the work connected to the compiler specifications, their verification, and rigorous development of the SubLISP implementations of the specifications, but not for verification of these implementations.

References

[Bj92] Dines Bjørner et al. *Final Deliverable of the ProCoS Project.* Computer Science Department, Technical University of Denmark, Lyngby, DK, 1992 (submitted to Springer Verlag for publication)

[BJ78] Dines Bjørner, Cliff B. Jones. *The Vienna Development Method: The Meta Language.* LNCS 61. Springer Verlag, 1978

[BM88] R.S. Boyer, J S. Moore. *A Computational Logic Handbook.* Academic Press, 1988

[Fr90] Martin Fränzle. *Verification of Compilers for Recursive occam-like Languages.* Master's Thesis, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, FRG, 1990

[Ho85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall Int., 1985

[Ho90] C.A.R. Hoare. *Refinement Algebra Proves Correctness of Compiling Specifications.* PRG-TR-6-90. Programming Research Group, Oxford University, UK, 1990

[i88a] inmos Limited. *Transputer instruction set: A compiler writers guide.* Prentice-Hall Int., 1988

[i88b] inmos Limited. *occam 2 Reference Manual.* Prentice-Hall Int., 1988

[Jo90] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall Int., 1990

[La91] Yassine Lakhneche. *Equivalence of Denotational and Structural Operational Semantics of PL.* Master's Thesis, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, FRG, 1991

[MO90] Markus Müller-Olm. *Correctness Proof of SubLISP to PL Translation.* Master's Thesis, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, FRG, 1990

[Po81] W. Polak. *Compiler Specification and Verification.* LNCS 124. Springer Verlag, 1981

[St77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977

[Yo89] W.D. Young. *A mechanically verified code generator.* Journal of Automated Reasoning, 5(4), December 1989