Interpreter Verification for a Functional Language

Manfred Broy, Ursula Hinkel, Tobias Nipkow[⋆], Christian Prehofer[⋆⋆], Birgit Schieder

Technische Universität München[⋆⋆⋆]

Abstract. Starting from a denotational and a term-rewriting based operational semantics (an interpreter) for a small functional language, we present a correctness proof of the interpreter w.r.t. the denotational semantics. The complete proof has been formalized in the logic LCF and checked with the theorem prover Isabelle. Based on this proof, conclusions for mechanical theorem proving in general are drawn.

1 Introduction

Compiler and interpreter verification is a key component in the correctness argument for any software system written in a high-level language. Otherwise the carefully verified high-level programs might be compiled or interpreted incorrectly. Proving the correctness of machine oriented programs [3] instead may be inevitable for some applications, but is methodologically a step backwards.

Verification of compilers and interpreters is also challenging from a theoretical point of view because complex semantical questions are involved [5, 6]. These comprise the formalization of semantical definitions and proof methods that are powerful enough to show the equivalence between quite different definitions of the semantics of programming languages.

When proving compilers correct, one of the difficulties is the treatment of recursion, which is handled by the fixpoint operator in the denotational semantics of the source language and by a stack discipline and gotos at the level of machine programs. This problem is dealt with in [4]. In the verification of interpreters similar problems have to be solved: here, recursive definitions are treated by unfolding at the level of syntax. Technically, the operational semantics defined by an interpreter is formalized by a recursive function transforming syntactic terms of the source language. This paper describes the correctness proof of an operational with respect to a denotational semantics for a small functional language. The main contributions are:

– The complete proof has been machine checked with the help of Isabelle [16], a generic theorem prover.

⋆ Research supported by ESPRIT BRA 6453, *TYPES*

⋆⋆ Research supported by DFG grant Br 887/4-2, *Deduktive Programmentwicklung*

⋆⋆⋆ Institut für Informatik, 80290 München, Germany.
E-mail: {broy,hinkel,nipkow,prehofer,schieder}@informatik.tu-muenchen.de

– Our notion of correctness of an interpreter is significantly stronger than most correctness conditions considered in other machine-checked compiler or interpreter proofs. We prove that whenever the outcome of a program is defined according to the denotational semantics, the operational semantics leads to a terminating computation with the same outcome. Thus we prove partial and total correctness by a single verification condition.

The paper is structured as follows. In Section 2 the syntax of the functional language, its denotational semantics, the definition of the interpreter and the basic correctness condition are given. In Section 3 the verification support system Isabelle/LCF is explained by describing its syntax, its basic type concept and its induction proof principle. In Section 4 the proof task is formalized in Isabelle/LCF. In Section 5 the structure and size of the proof is described. In Section 6 general aspects of machine support for large proofs are discussed.

2 The Interpreter and its Correctness

We define syntax and denotational semantics of *FOFL*, a first-order functional language, and an interpreter for it. Then we outline a correctness proof of the interpreter. FOFL is purposely kept small in order to focus on the main non-trivial aspect: proving the interpretation of recursively defined functions correct. Our interpreter is given as a recursively defined function, whereas in the literature, e.g. [19, 8], operational semantics is often given by means of inference rules. Therefore our verification task differs from those in the literature.

Syntax FOFL contains function application, conditional expressions, and recursive function definitions. Let Φ be a set of predefined function symbols with at least two constants $true, false \in \Phi$, let F a set of user-definable function symbols, and let V a set of variables. Each function symbol has a fixed finite arity.

The set T of terms is defined inductively:

– $x \in T$ for every variable $x \in V$.
– $\varphi(t_1, \ldots, t_n) \in T$ is an application of a predefined function symbol $\varphi \in \Phi$ of arity n to terms $t_1, \ldots, t_n \in T$.
– $f(t_1, \ldots, t_n) \in T$ is an application of a user-definable function symbol $f \in F$ of arity n to terms $t_1, \ldots, t_n \in T$.
– $\textbf{if}\, t_0 \,\textbf{then}\, t_1 \,\textbf{else}\, t_2 \,\textbf{fi} \in T$ for all terms $t_0, t_1, t_2 \in T$
– $(\textbf{fix}\, f(x_1, \ldots, x_n) = t_0)(t_1, \ldots, t_n) \in T$ is the application of a recursively defined function $f \in F$ (with formal parameters $x_1, \ldots, x_n \in V$ and body t_0) to terms $t_0, \ldots, t_n \in T$.

The set P of *programs* consists of all closed terms (that is, terms without free variables). FOFL is first-order because functions cannot have functions as arguments or return them as results.

Denotational Semantics We follow the standard theory of denotational semantics, see e.g. [12]. Let $\mathbf{D}$ be a set of data values equipped with a flat order.

The algebra A assigns a continuous function φ^A to each predefined function symbol $\varphi \in \Phi$. **Env** is the set of environments that assign data values to variables, and functions to user-defined function symbols. The functions **lookupvar** and **lookupfct** yield the values assigned to variables and functions, respectively, by an environment η. The operator FIX yields the least fixpoint of a functional. We write $\mathsf{IF}\ .\ \mathsf{THEN}\ .\ \mathsf{ELSE}\ .\ \mathsf{FI}$ for the conditional on the meta-level. The notation $[./.]$ is overloaded and denotes substitution as well as update of functions. It is used as a postfix operator.

The denotational semantics is defined by a function $\mathcal{T} : T \to \mathbf{Env} \to \mathbf{D}$ specified as follows:

$$\begin{aligned}
&\mathcal{T}[\![x]\!]\eta = \mathbf{lookupvar}\ x\ \eta\\
&\mathcal{T}[\![\varphi(t_1,\ldots,t_n)]\!]\eta = \varphi^A(\mathcal{T}[\![t_1]\!]\eta,\ldots,\mathcal{T}[\![t_n]\!]\eta)\\
&\mathcal{T}[\![f(t_1,\ldots,t_n)]\!]\eta = (\mathbf{lookupfct} f\ \eta)(\mathcal{T}[\![t_1]\!]\eta,\ldots,\mathcal{T}[\![t_n]\!]\eta)\\
&\mathcal{T}[\![\mathbf{if}\, t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2\ \mathbf{fi}]\!]\eta = \mathsf{IF}\ \mathcal{T}[\![t_0]\!]\eta\ \mathsf{THEN}\ \mathcal{T}[\![t_1]\!]\eta\ \mathsf{ELSE}\ \mathcal{T}[\![t_2]\!]\eta\ \mathsf{FI}\\
&\mathcal{T}[\![(\mathbf{fix}\ f(x_1,\ldots,x_n) = t_0)(t_1,\ldots,t_n)]\!]\eta = (\mathsf{FIX}\ \tau)(\mathcal{T}[\![t_1]\!]\eta,\ldots,\mathcal{T}[\![t_n]\!]\eta)\\
&\quad\text{where } \tau = \lambda g.\lambda d_1,\ldots,d_n.\mathcal{T}[\![t_0]\!]\eta[d_1/x_1,\ldots,d_n/x_n,g/f]
\end{aligned}$$

The Interpreter Let W be the set of all closed terms over Φ. Let a continuous boolean function *normal* be given, which yields true on a subset of W. The terms of this subset are called *normal forms*. Among them are *true* and *false*. The continuous function *eval* evaluates each term of W to its unique normal form. If t is a term in W, we write t^A for its interpretation in the algebra A.

The interpreter is based on the function *reduce*, which performs a single reduction step on a program $t \in P$:

$$\begin{aligned}
t \in W \Rightarrow{}& reduce[\![t]\!] = eval[\![t]\!]\\
\varphi(t_1,\ldots,t_n) \notin W \Rightarrow{}& reduce[\![\varphi(t_1,\ldots,t_n)]\!] = \varphi(reduce[\![t_1]\!],\ldots,reduce[\![t_n]\!])\\
\neg normal[\![t_0]\!] \Rightarrow{}& reduce[\![\mathbf{if}\, t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2\ \mathbf{fi}]\!] =\\
&\quad \mathbf{if}\ reduce[\![t_0]\!]\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2\ \mathbf{fi}\\
& reduce[\![\mathbf{if}\, true\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2\ \mathbf{fi}]\!] = t_1\\
& reduce[\![\mathbf{if}\, false\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2\ \mathbf{fi}]\!] = t_2\\
& reduce[\![(\mathbf{fix}\ f(x_1,\ldots,x_n) = t_0)(t_1,\ldots,t_n)]\!] =\\
&\quad t_0[(\mathbf{fix}\ f(x_1,\ldots,x_n) = t_0)/f, t_1/x_1,\ldots,t_n/x_n]
\end{aligned}$$

The interpreter is recursively defined by the function *val*, which applies *reduce* to a program t until a normal form is reached:

$$val[\![t]\!] = \mathsf{IF}\ normal[\![t]\!]\ \mathsf{THEN}\ t\ \mathsf{ELSE}\ val[\![reduce[\![t]\!]]\!]\ \mathsf{FI}$$

Interpreter Correctness Proof We call an interpreter *correct* with respect to a denotational semantics if the following property holds: whenever the denotational semantics assigns a defined value to a program, then the interpreter terminates with the same value. If a program has the undefined value under the denotational semantics, then its interpretation may terminate with any value or not terminate at all. Since the data domain $\mathbf{D}$ carries a flat order, we can state interpreter correctness formally as follows, where **void** denotes the empty environment:

$$\forall t \in P : \mathcal{T}[\![t]\!]\mathbf{void} \sqsubseteq val[\![t]\!]^A \tag{1}$$

A proof by structural induction over t looks promising, but there is a problem. In the case of function definitions the induction hypothesis is not applicable for two reasons: the function body need not be a closed term and the environment in which it is evaluated is not empty. Hence we face a typical problem of proofs by induction: the induction hypothesis is not strong enough and must therefore be generalized.

The main difficulty consists in finding a *suitable* generalization of the correctness condition. As we have seen, we need inequations of the form $\mathcal{T}[\![t]\!]\eta \sqsubseteq val[\![u]\!]^A$, where t is not necessarily closed, η not necessarily empty, and u a closed term. Since this inequation does certainly not hold for all such t, η, and u, we must find a relation $\sim$ between terms and environments on the one hand, and interpreted programs on the other hand, such that $\sim$ has the following properties:

1. The implication $(t, \eta) \sim u \Rightarrow \mathcal{T}[\![t]\!]\eta \sqsubseteq val[\![u]\!]^A$ is provable by structural induction over t.
2. For all closed terms t we have $(t, \mathbf{void}) \sim t$.

Proposition (1) follows directly from these conditions.

We will not give the exact definition of $\sim$, but describe it only informally. We say that $(t, \eta) \sim u$ holds if u is obtained from t by a substitution with the following two properties:

– For each free variable x of term t the following holds: the environment η assigns a value to x that is less or equal to the result of interpreting the term that is substituted for x in u. This property will be called WV (*weaker in variable*) in Sect. 5.
– For all user-definable function symbols f the following holds: if the environment η assigns a function to f, then this function is less or equal to the result of interpreting the declaration that is substituted for f in u. If η does not assign a function to f, then f is not substituted in u. This property will be called WF (*weaker in function*) in Sect. 5.

The generalized correctness condition reads:

$$\forall t \in T, \forall \eta \in \mathbf{Env}, \forall u \in P : (t, \eta) \sim u \Rightarrow \mathcal{T}[\![t]\!]\eta \sqsubseteq val[\![u]\!]^A \tag{2}$$

We prove this generalized correctness condition by structural induction over t. The most difficult case is application of a function definition $(\mathbf{fix}\ f(x_1, \ldots, x_n) = t_0)(t_1, \ldots, t_n)$. We must find and prove an auxiliary property: the least fixpoint of the functional associated with the declaration is less or equal to the result of interpreting the function definitions. This property is proved by fixpoint induction inside the structural induction.

The remaining cases of t are not difficult to prove, but require a lot of technical lemmata. These lemmata primarily concern invariance of $\sim$, and substitutions. The whole hand-written proof consists of about 70 pages. More details can be found in [18].

3 Isabelle/LCF

The proof described in the previous section does not make use of a specific logical system but relies on general notions from domain theory, e.g. $\bot$, $\sqsubseteq$ and fixpoints. The obvious choice for a machine-assisted version of the proof is LCF [7], a *Logic for Computable Functions*, which formalizes standard domain theory. Having fixed the precise logic, we still had a choice between two theorem provers supporting this logic: Cambridge LCF [15] and Isabelle/LCF. Cambridge LCF is dedicated solely to theorem proving in LCF whereas Isabelle [16] is a generic theorem prover which supports a host of other logics apart from LCF, e.g. First-Order Logic (FOL), Zermelo-Fraenkel set theory (ZF) and Higher-Order Logic (HOL). Isabelle can be instantiated with the syntax and proof rules of the object logic; Isabelle/LCF is the LCF instantiation of Isabelle.

Isabelle offers many principles for interactive theorem proving not present in Cambridge LCF: schematic variables ("logical variables" in Prolog parlance), higher-order unification and proof search via backtracking. These features give rise to powerful proof procedures which are a definite advance in automation over what Cambridge LCF has to offer. Thus we opted for Isabelle/LCF, which is an extension of Isabelle/FOL and follows the logic LCF as described by Paulson [15] as closely as possible. We will therefore concentrate on the differences between LCF and Isabelle/LCF.

Syntax Due to Isabelle's flexible front-end, the only syntactic difference is that curried application $f\ x\ y$, where $f : \tau_1 \rightarrow \tau_2 \rightarrow \tau_3$, is written $f(x, y)$. Correspondingly, $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ may be written $[\tau_1, \tau_2] \rightarrow \tau_3$.

Types Isabelle's type system is fairly close to that of LCF, namely ML-style polymorphism. However, LCF has both continuous functions, which are identified with Isabelle's built-in function type, and *predicates*. Predicates are formalized in Isabelle as functions with result type o, the type of formulae. Thus Isabelle/LCF needs to support two kinds of functions, because predicates need not be continuous. Otherwise one could define $X \equiv FIX(\lambda P.\neg P)$ and derive the contradiction $X \leftrightarrow \neg X$.

This problem can be solved using Isabelle's *type classes*, an overloading scheme similar to the one in the functional programming language Haskell [10, 13]. In Isabelle/LCF we simply declare a new class *cpo*, which is the class of all domains. We can now restrict certain constants to be available only at types which are of class *cpo*: $\bot : \forall\alpha{:}cpo.\alpha$, $\sqsubseteq : \forall\alpha{:}cpo.[\alpha, \alpha] \rightarrow o$, $FIX : \forall\alpha{:}cpo.(\alpha \rightarrow \alpha) \rightarrow \alpha$. In all three cases the type variable α ranges only over types of class *cpo*. The type o is *not* of class *cpo*, thus ruling out the term X above. If a type is declared to be in class *cpo*, e.g. $nat : cpo$, this means we can write formulae like $\bot \sqsubseteq 0$. Of course the behaviour of $\sqsubseteq$ on *nat* has to be axiomatized explicitly and does not follow automatically from *nat* : *cpo*.

Finally we need to say that $\tau_1 \rightarrow \tau_2$ is of class *cpo* provided both τ_1 and τ_2 are. However, this is only true if all terms of type $\tau_1 \rightarrow \tau_2$, where τ_1, τ_2 : *cpo*, are constructed from continuous functions by abstraction and application. Unfortunately one can construct $\lambda x.f(g(x))$, where $g : \tau_1 \rightarrow \sigma$, $f : \sigma \rightarrow \tau_2$ and σ is not a domain, and hence the composition of f and g need not be continuous. Fortunately, this situation can be ruled out quite easily:

1. All types except o are required to be domains, i.e. of class *cpo*.
2. There are no functions of type $o \rightarrow \tau$ where τ : *cpo*.

These restrictions correspond exactly to the ones in LCF where all types must be domains (there is no type of formulae), and there are no functions taking formulae as arguments.

The restriction that all types must be domains is a fairly severe one and causes many complications. Regensburger [17] solves this dilemma by a semantic embedding of LCF in Isabelle/HOL which allows to construct a separate space of continuous functions.

Induction The only induction principle for LCF is *fixpoint induction*:

$$\frac{P(\bot) \quad \forall x.P(x) \rightarrow P(f(x))}{\forall x.P(x)}$$

where $P(x)$ must be *admissible* [15]. In Cambridge LCF, the test for admissibility is an ML function which checks certain sufficient syntactic conditions. Most of these can be expressed as inference rules and have thus found their way into Isabelle/LCF. In this respect Isabelle/LCF is a little weaker than Cambridge LCF, which turns out not to be a problem in practice.

Paulson [15] shows how to derive structural induction from fixpoint induction and Cambridge LCF automates this derivation. Since Isabelle/LCF does not provide this automation, structural induction schemata were added explicitly.

4 The Specification

The abstract syntax of FOFL, its denotational and operational semantics, and the relation necessary for the correctness proof are all formalized as extensions of Isabelle/LCF.

The syntax of FOFL is represented by the type T. Its constructors correspond to the different syntactic forms of the language. Isabelle's mixfix notation enabled us to use the following readable syntax:

T ::= `var(`x`)` | `cst(`g`)` | `cstf` g `[`T`]` | `fun` f `[`T`]` | `cons[`T`,`T`]`
  | `if` T `then` T `else` T `fi` | `fix` f`(`x`) =` T `[`T`]`

This syntax is based upon the types V of variables, F of user-definable function symbols and Φ of predefined function symbols. These auxiliary types are not specified any further. Let us examine the different cases in detail.

`var(`x`)` is the variable $x : V$.

`cst(`g`)` is the predefined constant $g : \Phi$.

`cstf` g `[`t`]` is the application of the unary predefined function $g : \Phi$ to an argument t.

`fun` f `[`t`]` is the application of the user-defined function f:F to an argument t.

`cons[`t_1`,`t_2`]` is the pair (t_1, t_2).

`fix` $f(x)$ `=` t_0 `[`t_1`]` is the application of the recursive function f:F with formal parameter $x : V$ and body t_0 to an argument t_1.

Since we have constants, unary functions and pairs, we can express arbitrary n-ary functions. Isabelle/LCF provides no automation for the definition of data types like T. Hence the necessary induction and freeness axioms were asserted explicitly.

As an example of a FOFL program we present the recursively defined function `length` computing the length of a list and apply it to some argument list `cons[cst(a), cst(b)]`. The function symbols `0`, `succ`, `is_empty` and `tail` have their usual fixed interpretation.

```
fix length(xs) = if cstf is_empty [var(xs)] then cst(0)
                 else cstf succ [fun length [cstf tail [var(xs)]]]
[cons[cst(a), cst(b)]]
```

For the specification of the denotational semantics it was essential that Isabelle offers higher order constructs and λ-abstraction. We introduce the function den ($\mathcal{T}$ in Sect. 2) which evaluates a term relative to two environments:

$$den : [T, (V, D)map, (F, D \to D)map] \to D$$

The type constructor $(\alpha, \beta)map$ realizes finite functions from α to β. Its definition is not shown. The two environments provide semantic values for free variables: $(V, D)map$ maps first-order variables V to data values D and $(F, D \to D)map$ maps user-defined function symbols F to functions from D to D.

The translation of the different clauses for $\mathcal{T}$ in Sect. 2 is fairly straightforward. As an example we look at the case of **fix**. First we define a functional

$$tau : [F, V, T, (V, D)map, (F, D)map] \to (D \to D) \to (D \to D)$$

which corresponds to the term τ in Sect. 2 and is parameterized by the name of the recursive function, the name of its formal parameter, its body, and the two environments:

```
tau(f, x, t0, envV, envF) = (λg d. den(t0, envV[d/x], envF[g/f]))
```

The functional tau realizes one step in the approximation of the recursive function. LCF's fixpoint operator FIX is used to define the denotational semantics of recursive functions in FOFL:

```
t1 ≠ ⊥ ==> den(fix f(x) = t0 [t1], envV, envF) =
             FIX (tau(f, x, t0, envV, envF)) (den(t1, envV, envF))
```

The premise $t_1 \neq \bot$ is necessary because we have chosen to define all constructors of T to be strict, i.e. (`fix` $f(x)$ `=` t_0 `[`$\bot$`]`) $= \bot$.

The remaining clauses for *den* and the definition of the operational semantics in Isabelle closely follow the original specification in Sect. 2.

The overall specification defines 45 functions with 138 axioms. Most of the complexity comes from the full formalization of substitution. Fortunately, Isabelle's type system offers parametric polymorphism, which enabled us to define finite sets and maps once (following Paulson [14]) and use them repeatedly. Both the denotational (*den* above) and the operational semantics rely heavily on environments, i.e. maps, of all kinds.

5 The Correctness Proof in Isabelle

Next we discuss the mechanical verification of the interpreter. Starting from the proof in Sect. 2, it took a student with no previous experience with proof assistants approximately four months [9] to redo the whole proof in Isabelle, including the time to formalize the specification.

The guiding principle was a top down development of the proof. We first studied the top level of the proof of the generalized correctness condition (2):

```
((t, envV, envF) REL u) --> den(t, envV, envF) ⊑ ipret(val(u))
```

The definition of `REL` (see below) corresponds to the definition of $\sim$ in Sect. 2, the function `ipret` denotes the interpretation of terms.

The proof of the main theorem is based on a large body of lemmata about substitution and the interpreter function *val*. Rather than developing theories for substitutions and *val* first, we isolated the required lemmata during the proof of the main theorem, asserted them as additional axioms, postponing their proof until later. Except for one, all theorems concerning the function *val* were proved by fixpoint induction. The lemmata can be divided into independent classes:

Purpose of Lemmata	Number of Lemmata
general purpose	156
substitution	46
free variables, closed terms	52
properties of the interpreter, i.e. val	33
main theorem	24

As very large formulae are employed in the proofs we introduced abbreviations in order to hide details. Such abbreviations are of the form $t == u$ and can be expanded and folded during a proof. Two important abbreviations used are

```
denot_less_oper(t) ==
  ALL envV envF u.
    ((t, envV, envF) REL u) --> den(t, envV, envF) ⊑ ipret(val(u))
```

and

```
((t, envV, envF) REL u) ==
(t ≠ ⊥ & u ≠ ⊥ & is_ct (u) &
(EX substV :: (V,T)map. EX substF :: (F,decl)map.
  (substV ≠ ⊥ & substF ≠ ⊥ & u = subst(t, substV, substF) &
   (ALL x.(x: FV_x(t) = TT) --> WV(substV, envV, x)) &
   (ALL f.(f: FV_f(t) = TT) --> substF def f = TT & is_cdecl(substF @ f))
   (ALL f. (envF def f = FF --> substF def f = FF) &
           (envF def f = TT --> WF(substF, envF, f) )))))
```

where `WF` and `WV` are further abbreviations not shown here. Not only can the main theorem can now be stated concisely as `ALL t. denot_less_oper(t)` its inductive proof is also greatly simplified because the induction hypothesis is still readable.

Tactics Theorem proving in Isabelle is an interactive process. The user states the desired theorem to Isabelle and guides the proof by choosing the proof techniques and envoking appropriate tactics. *Tactics* are user-definable proof strategies and can be anything from the application of a single inference rule (single-step) to full-blown decision procedures. Altogether our proof consists of approximately 2400 user interactions. These can be analyzed as shown below.

Tactic	Number of Applications
Simplification	680
Single-step	920
Classical prover	173
Other	555
Total	approx. 2400

The Isabelle tactics [16] in the above table, e.g. the classical prover, are explained and discussed in the following. The order in the above table roughly reflects the user effort involved. For instance, the simplifier is fast and easy to use, whereas the (automatic) classical prover is slow and its success is hard to predict.

Isabelle's simplifier goes beyond classical first-order term rewriting. Its many enhancements, such as local assumptions and conditional equations, together with its flexiblity explain its extensive use to some degree. Isabelle provides several commands for single-step forward or backward reasoning, all of them variations on resolution. For instance structural and fixpoint induction are envoked by a backward resolution step. In many cases Isabelle's higher-order unification finds the correct assumption automatically. In only 15% of all applications of resolution we had to provide explicit instantiations to guide the search. Isabelle's classical prover is an automatic tactic for predicate calculus.

The tactics for term rewriting and resolution are very fast — they normally return within seconds compared to the automatic tactics which sometimes take up to minutes. Executing the whole proof takes 40 minutes (on a Sun Sparc 10).

Isabelle encourages users to construct new tactics by composing existing tactics via tacticals, thus customizing the prover for their particular application.

Once found, they allow for shorter and more abstract proofs. This was particularly important for us, because our proofs were undergoing frequent change, and small proofs are easier to maintain than large ones.

6 Proving in the Large

In this section we discuss some general aspects of large verification tasks. Let us first recall some often stated properties of interactive proof:

- Proofs can grow to a huge size, and it is a serious problem to extract the important information from a proof (state).
- Interactive proofs are produced incrementally, which has implications for the kinds of proof procedures that are useful.
- Proofs have to meet a range of sometimes conflicting criteria, among them: efficiency, elegance, readability, robustness under change, reusability, etc.

With these observations in mind, we discuss the theorem prover support.

Structuring Proofs Ideally, one would like to structure a proof in many abstract definitions and small lemmata until the proofs are easy. This is typically done in math books. We believe, however, that this is very hard in software verification where the details are overwhelming. First of all, this divide and conquer approach usually takes many attempts, often by skilled people, to find the right structuring. Although structuring is essential, mechanical verification requires a much more detailed and careful decomposition than typical proofs on paper. For instance, mathematicians often achieve elegant proofs by simply leaving most things implicit and by changing the perspective, which is hard to model formally. We identified the following concepts to alleviate this problem:

- *Definitions and abbreviations* are essential for structuring and decomposing larger proofs. With large systems, properties (e.g. invariants) of systems easily grow to pages. As in mathematics finding the right definitions and notation is often essential.
  Usually, abbreviations are global, but they may be also local to a proof. For instance, theorems in math books often have local abbreviations. Definitions not only need expanding but also contracting. This is often ignored because it requires higher-order rewriting: the left-hand side of a definition is a first-order term, whereas the right-hand side can be considerably more complex. For instance the abbreviation `denot_less_oper` above contains quantifiers on the right-hand side, which means that matching modulo α-conversion is required.
- *Structuring large proofs vertically: divide and conquer.* Apart from abbreviations, the only effective tool for structuring large verifications is the division into subtheories and lemmata. Generally, a clear and systematic design is is essential for successful verification. Case studies with functional programs [1] suggest writing a program in small units, in the hope that properties of these are easier and more compact to state.

– *Structuring large proofs horizontally: intermediate lemmata.* It is frequently necessary to introduce intermediate lemmata which are superfluous from a human point of view but are necessary to convince the theorem prover of the correctness of a proof step. In contrast to vertical structuring, these lemmata are tailored only towards a particular theorem and/or theorem prover. Most of them are tedious to find and obscure. Theorem provers with a high degree of automation and a low degree of user control, e.g. Boyer-Moore [2] and Ontic [11], often require such intermediate lemmata to guide the search.

Of course vertical structuring is to be preferred over horizontal structuring, which turned out to be essential for this case study. Yet for several proofs horizontal structuring by intermediate lemmata was used, although a detailed and well structured proof on paper was available. One reason was that many seemingly simple lemmata required a large number of interactive steps, which made intermediate lemmata necessary.

Automated Proof Support For interactive verification, strong and incremental automated proof support is necessary. Ideally, the user only has to give very abstract input to the prover, such as "do rewriting", or "use decision procedures". For instance, the Boyer-Moore system [2] is designed for automatic proof without any input from the user, except for providing some "hints". However, in our case studies we found that finer control is frequently necessary. Now the problem is that in many proof systems there is a wide gap between the automatic facilities and the low-level stepwise facilities: for lack of a middle ground, the user is often forced to work at an unnaturally low level. The following different levels of user control seem very natural, but are rarely fully supported:

1. In the first refinement, the user gives the prover hints on *what* to use, *e.g.*, suggesting certain rewrite rules, lemmata or proof strategies.
2. The user sometimes wants to have more control over *where* to use a tactic. For instance, one might want to apply simplification only to a particular premise of the goal.
3. Even more control can be exercised by specifying *how* a step should be done, *e.g.*, by providing explicit substitutions for instantiating a lemma.

For instance, in our case study higher-order unification combined with backtracking was used as a schematic method to compute desired instantiations of logic variables. This often relieves the user of the burden to provide concrete substitutions. Thus tactics can be expressed more abstractly, e.g. a tactic may roughly express "apply rule x in such a way that rule y applies afterwards". This is useful to avoid low-level proofs in situations where fully automatic support fails.

Abstract high-level proof methods facilitate reuse, as shown in our work: we first completed the verification for one language. Changing the syntax of the language invalidated most proofs, but redoing the proofs was a matter of days. Similarly, we added new constructs to the language, while being able to reuse most of the proof successfully.

Acknowledgments. We thank Konrad Slind for his contributions to Section 6.

References

1. M. Aagaard and M. Leeser. Verifying a logic synthesis tool in Nuprl: A case study in software verification. In K. G. Larsen, editor, *Proc. 4th Workshop Computer Aided Verification*, volume 663 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1992.
2. R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, 1988.
3. R. S. Boyer and Y. Yu. Automated correctness proofs of machine code programs for a commercial microprocessor. In D. Kapur, editor, *Proc. 11th Int. Conf. Automated Deduction*, volume 607 of *Lect. Notes in Comp. Sci.*, pages 416–430. Springer-Verlag, 1992.
4. M. Broy. Experiences with software specification and verification using LP, the Larch proof assistant. Technical Report SRC 93, DIGITAL Systems Research Center, 1992.
5. B. Buth, K.-H. Buth, A. Fränzle, B. v. Karger, Y. Lakhmeche, H. Langmaack, and M. Müller-Olm. Provably correct compiler developement and implementation. In U. Kastens and P. Pfahler, editors, *Compiler Construction*, volume 641 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1992.
6. P. Curzon. A verified compiler for a structured assembly language. In M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, editors, *Proc. 1991 Int. Workshop on the HOL Theorem Proving System and its Applications*. IEEE Computer Society Press, 1992.
7. M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: a Mechanised Logic of Computation*, volume 78 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1979.
8. C. A. Gunter. *Semantics of Programming Languages*. MIT Press, 1992.
9. U. Hinkel. Maschineller Beweis der Korrektheit eines Interpreters. Master's thesis, Institut für Informatik, TU München, 1993. In German.
10. P. Hudak, S. P. Jones, and P. Wadler. Report on the programming language Haskell: A non-strict, purely functional language. *ACM SIGPLAN Notices*, 27(5), May 1992. Version 1.2.
11. D. A. McAllester. *Ontic: A Knowledge Representation System for Mathematics*. MIT Press, 1989.
12. P. D. Mosses. Denotational semantics. In J. v. Leeuwen, editor, *Formal Models and Semantics, Handbook of Theoretical Computer Science*, volume B. Elsevier, 1990.
13. T. Nipkow and C. Prehofer. Type checking type classes. In *Proc. 20th ACM Symp. Principles of Programming Languages*, pages 409–418. ACM Press, 1993. Revised version to appear in *J. Functional Programming*.
14. L. C. Paulson. Deriving structural induction in LCF. In G. Kahn, D. B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lect. Notes in Comp. Sci.*, pages 197–214. Springer-Verlag, 1984.
15. L. C. Paulson. *Logic and Computation*. Cambridge University Press, 1987.
16. L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1994.
17. F. Regensburger. *HOLCF: Eine konservative Erweiterung von HOL um LCF*. PhD thesis, Technische Universität München, 1994. To appear.
18. B. Schieder. *Logic and Proof Method of Recursion*. PhD thesis, Institut für Informatik, TU München, 1994. To appear.
19. G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.