Chapter V. Conclusions

1. Summary

We have presented the theory, specification, implementation, and correctness proof of a compiler. The source and target languages *LS* and *LT* are realistic and useful languages rather than toys.

We have shown that a correct compiler can be systematically developed from a formal definition of source and target languages. The techniques employed in this thesis are very general and will also be applicable to other large software systems.

We have given a formal denotational definition of the source language *LS* on a low level of abstraction. This definition captures and formalizes standard compiling techniques. Future compilers will profit from these results whether or not they are formally verified.

In the course of the proofs we had to tackle various technical problems, such as the treatment of pointer operations, quantification in a quantifier free assertion language and the treatment of fixed points. Though apparently minor, these problems are of general importance and will most certainly arise in totally different applications.

We hope that this work is a convincing argument, that today's verification techniques are sufficiently powerful to be used in real life software development. At the same time this work reveals some of the weak points of the technology and points to future areas of research; we will discuss some possible improvements in the following sections.

Certainly there is no guarantee that our compiler will never fail. We have discussed some of the possible sources of errors earlier. Still, software developed from formal specifications and formally verified increases our confidence in its correctness to a point not achievable with conventional testing techniques alone. As the development of techniques for program verification will continue future proofs will leave less and less margin for errors.

Our compiler is not yet a "software product"; several features have to be added to make it a useful compiler. A suitable error handling and recovery mechanism as

well as output of the produced code in suitable format have to be added. As we pointed out earlier, these additions can be made without invalidating the program proofs given. Furthermore, it is desirable to include a minimum of code optimization in the compiler. We will discuss some of the possibilities for this as well as other useful extensions in the following section.

Numerous programs are written today for which correctness is imperative. We believe that in many cases verification is economically feasible and that these programs would greatly benefit from verification.

2. Extensions

It is desirable to be able to verify compilers that involve code optimizations. This is particularly important since a large number of the errors of present compiler are in the optimization part of the compiler.

Also, one may wish to apply our techniques to languages with features not present in *LS*. Both points are discussed in this section.

2.1. Optimization

Except for compile time evaluation of constant expressions our compiler does not perform optimizations of any kind. Let us briefly consider how possible optimizations would affect our proofs.

Let us classify optimizations in those which require data flow analysis and those which do not. In both cases we can distinguish optimizations performed on the source and the target level.

Optimizations that require data flow analysis are very hard to include in our compiler. They require extensive proofs that certain manipulations of the source or target text do not alter the meaning of a program. The correctness of these transformations may depend on an arbitrarily large context. The necessary theorems cannot be proven (or even formulated) in our assertion language. A reasonable approach would be to include an additional "optimization" step in the compiling process and develop a suitable logical basis for it. But it should be expected that a sophisticated optimization alone approaches or exceeds the complexity of the complete compiler presented here.

Optimizations that do not require flow analysis present a much brighter picture. For example, given the sequence of assignments $x \leftarrow a; x \leftarrow b$ where a and b are free of side effects (say simple variables) then we can omit the first assignment without changing the meaning of the program. Sequences of this nature can be detected easily and the proof that certain optimizations preserve the meaning of the program

is easy. Unfortunately, redundant operations are not very common in the source langauge.

By far the easiest optimization to include in our compiler is a technique called "peephole optimization" introduced by McKeeman [Mc65]. The idea is to find redundant operations in the produced target code or to find sequences of code that can be replaced by shorter code. The idea is very simple. One moves a window across the final code and matches the instructions in the window with a given set of patters for which a correctness preserving transformation is known.

The technique will be very successful since our code generation is very primitive. Code for components of the source program is produced irrespective of the context. For example, code sequences of the form $HOP\ n; LABSET\ n$ are very common. Also, our instruction set was chosen to be minimal. Most machines have instructions that are not strictly necessary but speed up certain frequent operations. LT could have an instruction $INCR$ which increments the top of the stack. Then the optimizer could search for the pattern $LIT\ 1; ADD$ and replace it by $INCR$.

The verification of a peephole optimizer is fairly straightforward. One can prove the validity of a set of transformations of the form $pattern_1 \rightarrow pattern_2$ and systematically apply them.

2.2. Register machines

Generating code for a register machine is substantially more complicated than producing code for a stack machine. Still, a sensible approach would be to first produce intermediate code for a stack machine and in a later translation step to convert it into code for a register machine. This later translation can be isolated from the rest of the compiler in which case a correctness proof becomes manageable. Sophisticated code for a register machine will require optimizations (e.g. register allocation) and the remarks of the previous sections apply.

2.3. New language features

Let us first discuss the addition of data structures of Pascal omitted in *LS*. Types *real* and *char* do not cause any problems at all. They merely require changes in the definition of Ty, the domain of types. Reals require some attention in the treatment of coercion operations but a solution is straightforward. Similarly, a simple minded implementation of variant records is easy. To model the (undesirable) semantics of Pascal we simply define the location corresponding to records to have several identifiers mapping to the same location. Even simpler is the implementation that assigns disjoint storage to all variants.

Packed data structures pose an interesting problem. Specifying an object to

be packed does not change the semantics in any way. The **packed** attribute is of pragmatic rather than semantic nature. Since a denotational definition is purely extensional an implementation ignoring the packed attribute is perfectly valid[1]. Conversely, a compiler could pack all data objects and would still be correct. It is unclear how attributes like **packed** can be captured by a formal definition or whether they are desirable at all.

We omitted *for* loops, *case* and *with* statements in *LS*. Including any of these poses no problem at all. The *case* statement and the for loop could both be implemented merely by changing the tree transformations to convert it into nested conditionals and while loops respectively. A simple implementation of the with statement would require to open a new scope in which identifiers denoting record components are declared as variables. In the target language storage would be set up for these variables and their values (addresses) determined on entry to the while statement. In either case a correctness proof is immediate.

Another restriction in *LS* is that mutual recursion is disallowd. As we mentioned earlier, the code generation translates mutually recursive procedures and functions correctly. The restriction lies in the static semantics part. An extension to cover mutual recursion could be implemented in either of two ways. First, we could have forward declarations as in Pascal. This merely extends the syntax and the tree transformation part of the definition. Changes to the static semantic definition are minor. Alternatively, mutually recursive procedures could be treated similarly to recursive types, i.e. we could allow a procedure or function identifier to occur before its declaration. This requires that in the static semantic definition a fixed point is used to describe the meaning of a set of declarations.

2.4. A stronger correctness statement

We proved that whenever the compiler terminates without printing an error message then the produced code is correct. We argued earlier that is a useful statement since it will never deceive the user of the compiler to believe that his program is correctly translated when it is not. The reader may ask how diffficult it is to prove a stronger statement.

Let us first consider termination. There are well known methods to prove termination in a weak programming logic by adding counters to the program which decrease with every loop and recursion and for which one proves that they always remain positive [LS75, MP73]. It poses no particular problem to include such counters. However, additional program documentation is necessary to prove that

1.) There may be additional restrictions in the static semantics for packed data objects. For example, the ISO Pascal standard disallows that components of packed data objects are passed as parameters.

they remain positive. For example, consider the implementation of the static semantics. To prove that the recursive procedures terminate requires to prove that the list structure representing the abstract syntax is free of cycles. This is not trivial and requires additional proofs of the tree transformation program to show that only cycle free list structures are produced.

Another possible extension is to prove that whenever the compiler prints an error message, then the input program is wrong. This is true for most error messages. For example, the static semantics defines exactly, when an error is called for. Similarly, from the LR–theory it is easily provable that an error message in the parser is only printed if the input program is in error. But at several places in the compiler error messages are printed in situations which are never expected to arise. For example, in the static semantics of expressions we consider all cases of expressions and if none of these cases obtains, an error message is printed. If the abstract syntax tree is build correctly, this situation will never arise and this error message will never occur. However, to prove that the abstract syntax is well formed is not immediate.

In some situations it will be impossible to guarantee, that no errors occur. For example, the stacks used in the parsing algorithm all have finite length. But for any fixed size of these stacks there will be an input program requiring a greater size.

Providing error messages for situations which should never occur or which indicate overflow conditions is standard practice in programming. Our approach shows that these "redundancies" have their place in the framework of verification as well.

3. Future research

This thesis raises several open questions and points to research areas to improve verification. In this section we discuss some of the important issues; but clearly, the list is open ended.

3.1. Structuring a compiler

Did we choose the best formal definition? Is our semantic definition the best suited for a compiler proof? Should the structure of the compiler be different? A definite answer to these questions can only be given if we have other verified compilers and can compare different approaches. But some minor points should be noted here.

In the static semantics we computed the modes of all expressions to check the program for validity. This information has been discarded afterwards. Later, the

code generation also requires information about modes of expressions; our compiler recomputes modes if necessary. But clearly, this is redundant as one could store the mode information in the abstract syntax tree. The main reason why we choose not to do so is a technical one. Changing the abstract syntax tree is an update operation on a pointer structure which has no obvious counterpart in the formal definition. Consequently it is not immediately clear how the correctness of such update operations can be stated.

Alternatively, static and dynamic semantics could be combined in one program. This also eliminates redundant computation of modes. In return the structure of the resulting compiler is less clean.

The idea of combining different parts into one can be carried a step further. The scanner and the parser could be combined into one program thus eliminating the representation of the program as sequence of tokens. A natural way towards this implementation are coroutines which unfortunately are not available in our implementation language. Any two programs that communicate by writing and reading one file can be combined using coroutines without any change in the program proof.

The size of programs that can be translated by our compiler is severely limited by the fact that the complete abstract syntax tree is stored in memory. But this is not strictly necessary. The semantics of LS is such that semantic ckecking and code generation can proceed from left to right requiring only a limited context. A suitable reorganisation of our program seems possible without significant changes in the formal theory. Again, here is a tradeoff between efficiency and structural clarity.

3.2. Improvements of verification systems

The research presented here may well be the first real test of any automatic verification system. Several weak points that call for improvement have been revealed.

Clearly, the assertion language used in a verifier is a compromise between expressive power and efficiency of the system's theorem prover. For example, allowing quantification in assertions will make theorem proving much harder and may render the whole system useless.

Still it seems desirable to allow for limited quantification. Can a proof rule corresponding to our weak $\forall$-introduction be included. Even if the theorem prover does not deal with quantification, can quantified formulas be allowed syntactically. For example, one might might want to write $\forall x.P(x, y)$ and have the theorem prover treat the formula as a predicate symbol with one free variable y. Of course,

this would not change anything in the capabilities of the system except for the readability of the documentation.

Another desirable feature is a typed assertion and rule language. Many proofs given here were not forthcoming because of a misspelling of a predicate or function symbol. Simple minded type checking could detect many of these errors.

The verifier should provide for virtual code. Not merely virtual variables but rather virtual types, virtual parameters, virtual procedures and so on. Such a concept is imperative if the verified program is ever to be compiled. Also, it greatly enhances the readability of programs and clarifies which parts of the code require further refinement.

The theorem proving part of the Stanford verifier performed extremely well in most cases. In particular the built–in complete decision procedures for frequently occuring theories proved to be a valuable asset. The main problems are in the area of rules supplied to the prover.

The semantic contents of a rule should be stated independently of a particular heuristics of how to use this rule. Currently, there are several different ways of stating the same fact. To get the prover to simplify a verification condition efficiently, rules have to be stated in the "right" format. This requires some experimenting and frequent rewriting of rules, a possible source of errors.

In some cases it is desirable to manually intervene in a proof and give hints to the system. Or, one may want to find out why exactly a verification condition does not simplify to true. But manual proofs should be the exception rather than the rule. Giving manual proof guidance to the system in all cases would make this research impossible. Rather what is needed is a general concepts of a "proof schema", a way of telling the system how to prove a verification condition. Consider the static semantics for expressions. The program consists of several branches each of which checks a particular syntactic class of expressions such as identifiers, unary operators, binary operators and so on. The structure of the proof for each of these cases is very similar. Therefore, given the proof of one case, all the other cases follow by analogy. The concept of a proof schema seems very useful here.

We have shown that refinement of a program leads to structurally similar verification conditions. An intelligent verification system might remember previous proofs, thus greatly reducing the amount of work necessary to verify a refined program.

3.3. Better verification techniques

Weak $\forall$–introduction proved a useful concept in this research. Is it generally applicable? Is it a special concepts or can it be generalized? For example, is there weak $\exists$–introduction and so on?

We used operationalization of fixed points to compute recursive types. Is this just a gimmick to circumvent limitations of the verifiers assertion language or is it a generally applicable technique. What is the most general situation in which operationalization can be employed?

Even more interesting is the opposite question. Clearly, any update operation on pointers can be expressed as a least fixed point. Is this a practical way to reason about pointer operations? It certainly is, if pointers are used to represent recursive domains and if the pointer operations have a natural counterpart (e.g. fixed point) in the underlying theory. But can it also be used to prove a tree traversal algorithm which manipulates link fields to avoid stacking operations?

Alternatively, it may be possible to develop high level concepts to reason about pointers similar to those proposed by Reynolds for arrays [Re79].

3.4. Program development systems

The ideas put forth sofar all assume that the current paradigm of program verification is the best possible. Maybe, it is not. What would the ideal verification system for our compiler proof look like? Ideally, of course, we want a program synthesis system, but let us be a little more realistic.

One of the main problems in carrying out the development of the compiler and its proof was consistency. We had to deal with an extremely large formal definition, theorems derived from this definition, machine readable versions of specifications and theorems, several versions (of different refinement level) of the program to be written, and programs communication with a given program. A system that keeps track of all these formal objects and insures notational consistency seems easy to conceive and would be of great assistence.

But, such a system could do even more. We argued previously, that many proofs of cases of our program follow by analogy. But the implementation of different clauses of the definition follow by analogy also. Thus, what is needed is an intelligent editor. For instance, the user should be able to explain to such a system, how a recursively defined function is to be transformed into an efficient program. Part of such a transformation is the specification of implementation details, for instance, how a particular abstract object is to be represented in the program. An even more advanced system might know about efficient default implementation of certain objects (automatic data structure selection).