Available online at www.sciencedirect.com
SCIENCE DIRECT
ELSEVIER
Electronic Notes in Theoretical Computer Science 132 (2005) 19–35
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

A Program Result Checker for the Lexical Analysis of the GNU C Compiler

Sabine Glesner[a] Simone Forster[a] Matthias Jäger[a]

[a] *Fakultät für Informatik, Universität Karlsruhe, 76128 Karlsruhe, Germany*
Email: {glesner|simone|matthias}@ipd.info.uni-karlsruhe.de

Abstract

In theory, program result checking has been established as a well-suited method to construct formally correct compiler frontends but it has never proved its practicality for real-life compilers. Such a proof is necessary to establish result checking as the method of choice to implement compilers correctly. We show that the lexical analysis of the GNU C compiler can be formally specified and checked within the theorem prover Isabelle/HOL utilizing program checking. Thereby we demonstrate that formal specification and verification techniques are able to handle real-life compilers.

Keywords: Formal correctness, compiler implementation correctness, program result checking, Gnu C compiler, lexical analysis, Isabelle/HOL.

1 Introduction

Program result checking is a method to reduce the cost of formal verification. Instead of verifying a piece of software, one only verifies its result. In the area of compiler construction, especially for compiler frontends, result checking has been established as the appropriate method for ensuring compiler correctness in theory. Nevertheless, proofs for practicality are still missing. Our work strives for closing this gap. In this paper, we show how the lexical analysis of the GCC compiler can be formally verified with program checking. We have chosen the GCC compiler because it is one of the most widely used compilers worldwide. We require our solution to fulfill the following criteria: Of course, it should be able to deal with such a large system as the GCC. Moreover, the method should not only be applicable to the GCC but other comparable systems as well. In particular, it should be possible to implement the checking

1571-0661
doi:10.1016/j.entcs.2005.01.029

method in generators which is important since many compilers are not written by hand but generated by compiler generators. The involved generators should be extendable to generate not only the compiler phases but also the formally correct checkers for testing correctness of their produced results. As a sacrifice, we do not require the checking method to be able to decide the correctness of each result, as long as most practical cases are covered.

In our approach, we specify the lexical analysis within the theorem prover Isabelle/HOL. Then we use Isabelle's program extraction facility [3,1] to directly generate executable ML code which is formally correct by construction (assuming the correctness of the Isabelle theorem prover). This ML code is the checker for the lexical analysis of the GCC. It gets as input the program to be lexed. It recomputes the result of the lexical analysis and compares it with the result computed by the GCC. We have specified the lexical analysis in Isabelle/HOL in two variants which both implement it as a finite automaton. The first variant implements it as a primitive recursive function such that all state transitions are defined by nested if-then-else conditions. The second variant implements the automaton also as a primitive recursive function but the state transitions are stored in a table (implemented as a list) and looked up during the scanning process. In our experiments, we have compared both versions wrt. their readability and efficiency.

This paper is structured as follows: In section 2, we introduce program result checking and its use in compiler verification. In particular, we discuss previous checking approaches for the lexical analysis and compare them to our approach taken in the work presented here. In section 3, we describe the architecture of the GCC compiler and its lexical analysis as far as necessary for presenting our work. Section 4 describes the two specification variants for the lexical analysis in Isabelle/HOL. To keep the presentation simple, we discuss a very elementary lexical analysis as example and subsequently point out highlights of the specification of the GCC lexical analysis. In section 5, we describe our overall checking architecture, in particular the connection between the Isabelle-generated checker and the GCC compiler. The results of our experiments are summarized in section 6. We close this paper with a discussion of related work in section 7 and the conclusions in section 8.

2 Program Checking in Compiler Verification

2.1 The Principle of Program Checking

Program checking [2] has been introduced to improve the reliability of programs. It assumes that there exists a black box implementation P computing a function f. A *checker* for f checks for a particular input x if $P(x) = f(x)$.

Assume that $f : X \to Y$ maps from X to Y. Then the checker *checker* has two inputs, x and y, whereby x is the same input as the input of the implementation P and y is its result on input x. The checker has an auxiliary function f_ok that takes x and y as inputs and checks if $y = f(x)$ holds. If the checker is formally verified, then we get a formally correct result of program P in case that the checker returns **True**. Since the checker does not depend on the implementation P, it can be used for any program P' implementing f.

```
proc compiler_checker(source : X,
     target : Y) : BOOL
   if translate_ok(source, target)
   then return True
   else return False
end proc
```

The original intention of program checking was to build checkers which are simpler and implement different algorithms than the implementation. This would imply that bugs in the implementation and in the checker were independent and unlikely to interact so that bugs in the program could be caught more likely. This view on program checking partly needs to be revised when applying it in the verification of compilers. In particular, we need to distinguish the different phases in the compilation process.

2.2 Program Checking for Compilers

Compilers consist of a frontend and of a backend. The frontend checks if a given input program belongs to the programming language and transforms it into an intermediate representation, cf. figure 1. The backend takes this intermediate representation, optimizes it, and generates machine code for it. The tasks to be performed by these phases differ significantly. While the

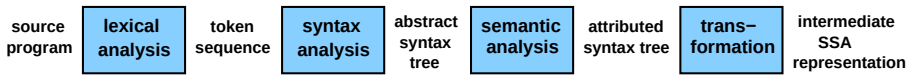


Fig. 1. Frontend Tasks

frontend tasks all have unique solutions, the backend phases with their optimization possibilities can produce several correct solutions. As long as results are unique, program checking in its form introduced above works well. Problems arise if several correct results exist. In [6], we proposed a method called *program checking with certificates* to deal with this situation. Thereby, the compiler is modified to not only output its result, but also a certificate how it has computed the result. The checker uses this certificate to recompute the result and compares its own result with the one produced by the compiler.

In this paper, we concentrate on checking frontend tasks, in particular lexical analysis, cf. also [7]. Programming languages are described and processed by a three-stage process: First, a regular language defines how to group indi-

vidual characters of the input program into tokens. For example, the character sequence 'e' 'n' 'd' would be combined into the token 'END', treated as an indivisible unit. Regular languages can be implemented easily by finite automata, constituting the *lexical analysis*, also called *scanner*. These tokens are the basis to define the context-free structure of programs by a deterministic context-free grammar. The corresponding compiler phase is the syntactic analysis. Its result is a unique syntax tree whose correctness is easily checked by a single top-down traversal. In the third stage, attributes are associated with the nodes in the abstract syntax tree, e.g. by specifying an attribute grammar. These attributes define the context-sensitive properties of programs. They are determined during the semantic analysis which therefore typically traverses the abstract syntax tree in sophisticated special orders. Nevertheless, the correctness of the attributes can be checked easily within the context of the corresponding production so that one traversal of the attributed syntax tree suffices to check its correctness. This characteristic that one traversal of the tree is enough to check correctness holds for both the syntactic and the semantic analysis but not for the lexical analysis which needs a bit of extra work as discussed in the following subsection.

Note that correctness in the frontend phases does not mean that transformations are semantics-preserving. Semantics is only defined for the attributed syntax tree but neither for the token sequence nor for the syntax tree. Hence, for the frontend phases, correctness means that the token sequence, the syntax tree, and its attributes have been computed correctly according to the specified finite automaton, context-free grammar, and attribute grammar, resp.

2.3 Program Checking for The Lexical Analysis

The lexical analysis combines subsequent input characters into tokens, thereby eliminating meaningless characters such as whitespaces, line breaks, or comments. Identifiers (e.g. variable names) need special treatment. They are always mapped to one specific token, e.g. 'IDENT'. The particular name of the identifier itself is stored in a separate table called symbol table. A similar approach is taken for numbers whose token is also unique, e.g. 'REAL' or 'INT', and whose value is stored in the symbol table as well. The lexical analysis is specified by regular languages (implemented as finite automata) together with the rule of the longest pattern (prefer the longest possible character sequence as possible) and priorities to avoid indeterminism. E.g. the sequence 'w' 'h' 'i' 'l' 'e' is mapped to the token 'WHILE', not to an identifier.

The checker for the scanner needs to make sure that the computed token sequence has been computed according to the transition rules of the automaton. In [7], a checking approach has been proposed which tries to recompute

the character sequence of the input program given the token sequence of the scanner. This backwards computation is rather complicated because it is not unique. Its ambiguity comes essentially from the whitespace, line breaks, and comments which separate the character sequences of tokens. There are pairs of subsequent tokens whose respective character sequences must be separated by whitespaces in the original program while the character sequences of other tokens may be, but do not need to be separated. For example, the character sequences for the tokens 'WHILE' and 'IDENT' must be separated. If they were not, then the rule of the longest pattern would be applied, recognizing only one identifier consisting of the concatenation of the character sequence of the 'WHILE' token and of the 'IDENT' token. In [6], we have argued that this checking approach is not any simpler than the original scanning algorithm and does not reduce the verification effort noticeably.

Therefore our approach is different. We completely recompute the lexical analysis. For this purpose we specify it within the interactive theorem prover Isabelle/HOL. Isabelle/HOL has a program extraction facility [3,1] which we use to directly generate executable ML code. In our setting, the checker for the lexical analysis has three inputs, the source program, the computed token sequence, and the specification of the finite automaton. It recomputes the token sequence and outputs **True** only if the recomputed sequence is identical with the original one.

3 Architecture of the GCC Compiler

The GNU Compiler Collection (GCC) [13] contains frontends for C, C++, Objective-C, Fortran, Java, and Ada, as well as libraries for these languages. These languages are transformed into an intermediate tree-based representation called *treelang*. There are a number of backends for *treelang* within GCC. GCC is also used as name when the emphasis is on compiling C programs (GCC = GNU C Compiler as former abbreviation). We focus on this use. GCC supports the standard for the C language [8]. Therefore options like '-ansi', '-std=c89', or '-std=c99' have been chosen.

The GNU C preprocessor [14,4], abbreviated by *cpp*, is a macro processor that is used automatically by the C compiler to transform input programs before compilation. As the C compiler, it can be used strictly according to the ISO Standard C, e.g. with options like '-std=c99' or '-std=c89'. *cpp* converts the input file into a sequence of *preprocessing tokens* which are also the tokens used by the compiler. The compiler does not re-tokenize the preprocessor's output so that each preprocessing token becomes one compiler token. *cpp* is implemented as a library *cpplib* with the preprocessing token as fundamental

unit. (The preprocessor in previous versions of the GCC operated on text strings as fundamental unit.)

Tokens fall into five classes: identifiers, numbers, string literals, punctuators, and other. An *identifier* is any sequence of letters, digits, and underscores which begins with a letter or underscore. Keywords are treated as ordinary identifiers. A *number* begins with an optional period, a required decimal digit, and then continues with any sequence of letters, digits, underscores, periods, and exponents. Exponents are two-character sequences starting with 'e', 'E', 'p', or 'P' and are followed by '+' or '-'. (Exponents beginning with 'p' or 'P' are used for hexadecimal constants.) This rather unusual definition isolates the *cpp* from the full complexity of numeric constants. Correctness of numbers is only checked in the syntactic analysis. String literals are string constants, character constants, and header file names (the arguments of '#include'). String constants and character constants are straightforward: ″...″ or '...'. Header file names either look like string constants, ″...″, or are written with angle brackets instead, <...>. *Punctuators* are the usual one-, two- and three-character operators. Any other single character is considered *other*. The C compiler will almost certainly reject source code containing *other* tokens.

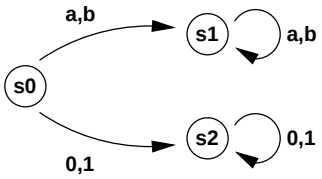
Within the library *cpplib*, the lexer is contained in the file 'cpplex.c'. It is hand-coded and not implemented as a state machine. cpplex.c contains the function `_cpp_lex_direct` which lexes individual tokens. For identifier, number, and string tokens, it does not only compute the token (e.g. CPP_NUMBER) but also its value, e.g. the concrete number. New lines are treated specially by this function, in particular context-sensitively. The C standard requires that directives for the preprocessor are terminated by the first unescaped newline character, even if it appears in the middle of a macro expansion. Therefore, if a certain variable called `in_directive` is set, the lexer returns a `CPP_EOF` token, which is normally used to indicate end-of-file, to indicate end-of-directive. In such a context, this token never means end-of-file. In section 5, we describe how we have handled this peculiarity when checking the correctness of the lexer. We also describe how we have modified the function `_cpp_lex_direct` so that it outputs the stream of tokens and their values to be checked for correctness.

4 Lexical Analysis in Isabelle/HOL

We check the lexical analysis of the GCC compiler by recomputing its result via a formally verified implementation and by comparing this correct result with the one produced by the GCC, cf. subsection 2.3. For this purpose, we specify the lexical analysis within the Isabelle/HOL theorem prover and use

Isabelle's program extraction facility to generate executable ML code [3,1]. In this section, we first demonstrate our method in principle by specifying a simple lexical analysis. Then we proceed by describing the specification for the lexical analysis of the GCC. We conclude with a discussion of our specification and the design decisions which we have made.

4.1 A Small Example



We start with a simple lexical analysis that recognizes identifiers and numbers. Identifiers consist only of characters 'a' and 'b', numbers only of '0' and '1'. This automaton is displayed in the figure to the left. In Isabelle/HOL we specify it as a scanner function that takes a list of characters as input and processes them one after the other, thereby possibly changing its state. The scanner function outputs the recognized token, in case of a sequence of characters the token 'IDENTIFIER', and in case of a sequence of numbers the token 'INTEGER'. In addition to these tasks, a scanner in the lexical analysis computes some more information. The succeeding phases in a compiler do not only need to know that a token 'IDENTIFIER' or 'INTEGER' has been recognized but also the value of that token, i.e. the actual identifier or number represented by the character sequence. To specify this functionality, we define the scanner as a primitive recursive function on four input parameters, namely on the sequence of characters that has not been read yet, the sequence of characters that represents the value of the current token, the current state, and a list of tokens and their values which have been scanned so far. Output is the possibly extended list of recognized tokens and their values.

We have two possibilities for the definition of the scanner. First we can specify it as a primitive recursive function by a case distinction such that each case describes the processing of one input symbol depending on the current configuration of the automaton. This results in many nested if-then-else case distinctions. Secondly, we can specify the scanner by a primitive recursive function which takes as additional input a table summarizing the state transitions and actions to be taken thereby. We have implemented both possibilities in Isabelle/HOL, cf. section 6 for a comparison between them.

In the remainder of this subsection, we discuss the most interesting parts of both specification possibilities for the above example automaton. The specification of the scanner recomputing the results of the GCC follows these lines directly, except that it is much larger, cf. subsection 4.2. In discussing the smaller example first, we can illustrate the underlying principles more clearly.

```
(*datatype for all characters that occur in our automaton*)
datatype isachar = l_a | l_b | n_0 | n_1 | s_nl | s_space | s_tab
(*a,b,0,1,space,newline,tabulator*)
```

Fig. 2. Definition for Isabelle-coded Characters

```
datatype token = IDENTIFIER | INTEGER | undef
datatype state = fail | s0 | s1 | s2
```

Fig. 3. Token Definition

```
(*IDENTIFIER:*)
constdefs
  ident :: "isachar set"
  "ident ≡ {l_a,l_b}"
(*a,b*)
```

```
(*INTEGER:*)
constdefs
  integer :: "isachar set"
  "integer ≡ {n_0,n_1}"
(*0,1*)
```

Fig. 4. Symbol Sets with Identical State Transitions

```
(*definition of the following state, starting in initial state*)
constdefs nextstate::"isachar ⇒ state"
  "nextstate x ≡ if x ∈ ident then s1
        else if x ∈ integer then s2
        else if x = s_space then s0
        else if x = s_nl then s0
        else if x = s_tab then s0
        else fail"
```

Fig. 5. Auxiliary Function "nextstate"

4.1.1 Direct Specification of the Scanner Function

The Isabelle/HOL scanner specification consists of two main parts, the definition of auxiliary data types and the definition of the scanning function implementing the automaton. Since certain characters as e.g. '0', '1', 'x', 'o' have a predefined meaning in Isabelle, we translate the entire input character sequence into a list of Isabelle-coded characters. The Isabelle-coded characters are introduced with the data type definitions in figure 2[1]. The translation of the input character sequence into the list of Isabelle-coded characters is implemented in ML, cf. also figure 9. It is a one-to-one mapping from the original input characters to their Isabelle-encoded correspondents, e.g. the input character '1' is mapped to n_1.

The states of the automaton as well as the recognizable tokens are also

[1] There is also a built-in data type "char" in Isabelle but since we have not found documentation for it, we decided to define this data type explicitly in our specification.

```
(*definition of the main function*)
consts   scanner :: "isachar list ⇒ (token × isachar list)list ⇒ state ⇒ isachar list ⇒
(token × isachar list)list"
(*input stream of characters × output stream of (token,word)tupels × current state
× list of already memorized characters ⇒ new output stream*)
primrec
  "scanner [] outlist currstate memlist = (
    if currstate = s0 then outlist
    else if currstate = s1 then outlist@[(IDENTIFIER, memlist)]
    else if currstate = s2 then outlist@[(INTEGER, memlist)]
    else if currstate = fail then outlist
    else outlist   )"

  "scanner (x#xs) outlist currstate memlist = (
    if currstate = s0 then( ... ) (* Details left out here for space reasons *)
    else if currstate = s1 then
      if x ∈ ident then scanner xs outlist s1 (memlist@[x])
      else if x =s_space
        then scanner xs (outlist @ [(IDENTIFIER, memlist)]) s0 []
      else if x = s_nl
        then scanner xs (outlist @ [(IDENTIFIER, memlist)]) s0 []
      else if x = s_tab
        then scanner xs (outlist @ [(IDENTIFIER, memlist)]) s0 []
      else if nextstate x=fail
        then scanner xs (outlist@[(IDENTIFIER, memlist),(undef,[x])]) fail []
        else scanner xs (outlist@[(IDENTIFIER, memlist)]) (nextstate x) [x]
    else if currstate = s2 then ( ... ) (* Details left out here for space reasons *)  )"
```

Fig. 6. Direct Implementation of the Scanner Function

defined by data type definitions, cf. figure 3. To simplify the specification of the scanner function, we unite symbols causing the same state transition each into one set, cf. figure 4. The specification of the scanner function in the direct implementation needs an auxiliary function "nextstate" that computes the following state when started in the initial state, cf. figure 5. This function is necessary when the sanner function reads a symbol which cannot be accepted in the current state. Since the symbol cannot be put back into the input stream (this would destroy the property of the scanner function of being primitive recursive), the scanner function computes instead the state which will be reached when processing this symbol in the initial state by calling the function "nextstate". The scanner function is specified as a primitive recursive function as listed in figure 6 [2]. For space reasons, we have displayed only that part of the state transitions which start in state s1 and left out the transitions starting in s0 and s2.

[2] In Isabelle, @ denotes concatenation of lists. The "cons" operation is denoted by '#'.

```
constdefs scannertab :: "(state × isachar set × state × token option × bool)list"
(*(current state × set of possible characters for state transition × poststate × token
  that will be output × if read character should be appended to memorizing list)list*)
  "scannertab ≡
    [ ...
      (   s1,   ident,      s1,   None,                 True   ),
      (   s1,   m_space,    s0,   Some IDENTIFIER,      False  ),
      (   s1,   m_nl,       s0,   Some IDENTIFIER,      False  ),
      (   s1,   m_tab,      s0,   Some IDENTIFIER,      False  ),
    ... ]"
```

Fig. 7. State Transition Table

4.1.2 Specification with a State Transition Table

Instead of the direct specification, we can also define the scanner function by passing it a table as additional parameter defining the state transitions of the scanner. For this variant, we need additional definitions for sets of symbols that each induce the same state transitions to be specified in the table. These definitions are analogous to the ones in figure 4 so that we only summarize them here: "emptyset = { }", "m_nl = { s_nl }", "m_space = { s_space }", "m_tab = { s_tab }". Furthermore, we need the specification of the state transition table. It is given in figure 7. For space reasons, we have only displayed state transitions starting from s1. For each current state and current input character (the first two columns), the table specifies the new state, the token to be output on that state transition, and a Boolean value indicating if the character should be memorized for the computation of the value of the currently scanned token. Also another table called "finstatetab" (final state table) is needed. It represents a list of triples (s, B, t), each consisting of a state s, a Boolean value b, and a token t. Its intended meaning is that whenever the automaton is in state s, cannot accept the current character, and is in a possible final state (b=true), then the output token is t.

The scanner function itself takes six arguments: the input list of characters, the already computed output list of tokens and their values, the current state, the list of memorized characters to be used when computing the value of the currently scanned token, the state transition table, and the final state table. Its output is the eventually extended output list of computed tokens and their values. The scanner function uses two auxiliary functions, "scanresult" and "finscanresult". Both functions are used to extract a specific tuple from the state transition table and the final state table, resp.

4.1.3 Discussion of the Specification Possibilities

Both specification variants, the direct one as primitive recursive function with many case distinctions and the one using a state transition table (also being

```
(*definition of scanner function with state transition table*)
consts
(*main function for the scanning process*)
scanner :: "isachar list ⇒ (token × isachar list) list ⇒ state ⇒ isachar list ⇒ (state ×
isachar set × state × token option × bool)list ⇒ (state × bool × token option)list ⇒
(token × isachar list)list"
(*input stream of characters × output stream of (token,word)tupels × current state
× list of actually memorized characters × state transition table × final state table ⇒
new output stream*)
primrec
"scanner [] outlist currstate memlist statetab finstatetab = (
   case scanresult None currstate statetab of (poststate,tok,append) ⇒ (
     if poststate = s0 then (
       case tok of None ⇒ outlist
         | Some cpptoken ⇒ outlist@[(cpptoken, memlist)] )
     else outlist@[(undef, memlist)])  )"

"scanner (x#xs) outlist currstate memlist statetab finstatetab = (
   case scanresult (Some x) currstate statetab of (poststate,tok,append) ⇒ (
     if poststate=fail then (
       case finscanresult currstate finstatetab of None
         ⇒ outlist@[(undef, (memlist@[x]))]
       | Some cpptoken
         ⇒ scanner xs (outlist@[(cpptoken, memlist)]) (nextstate x nextstatetab)
             [x] statetab finstatetab)
     else ( if append=True then
       scanner xs outlist poststate (memlist@[x]) statetab finstatetab
       else ( case tok of None ⇒
           scanner xs outlist poststate memlist statetab finstatetab
         | Some cpptoken ⇒
           scanner xs (outlist@[(cpptoken, memlist)]) poststate [] statetab finstatetab
   ))))"
```

Fig. 8. Table-Based Implementation of the Scanner Function

primitive recursive), arise naturally from the finite automaton to be specified and implemented, resp. They correspond directly with the two implementation possibilities for the lexical analysis in standard compiler construction. We have compared them with respect to the following criteria: Their readability, the efficiency with which Isabelle can read them in and generate ML code, and the efficiency when using them to scan input programs, i.e. to check the lexical analysis of input programs by recomputing the result of the lexical analysis of the GCC. Concerning the first criterion, readability, the version using the state transition table is clearly superior. In contrast, in our experiments discussed in detail in section 6, this version has turned out to be much more inefficient than the direct specification variant.

4.2 Lexical Analysis of the GCC

The specification of the lexical analysis of the GCC goes exactly along the lines described in subsection 4.1, except that it is much larger. We have set it up by a three-step process:

Determination of the Tokens In the first step, we determined the tokens to be recognized in the lexical analysis. The C language definition [8] (Annex A) defines the tokens and their regular syntax. It distinguishes between *preprocessing tokens* and *tokens*. In the GCC, this distinction does not exist because only the preprocessor computes tokens which are used directly in the succeeding compilation phases. Moreover, the GCC lexical analysis can recognize more tokens than provided in the language specification. Only if used with options like '-ansi', exclusively the standard tokens will be recognized. We decided to stick to the standard version. The tokens in the C standard and in the GCC code are named differently. The GCC defines tokens in a file called "cpplib.h" It defines a data structure `TTYPE_TABLE` which is a table containing pairs consisting of a token and the corresponding character sequence. Even though named differently, these tokens matched directly with the ones specified in the C standard.

Determination of the Finite Automaton Based on the descriptions of the tokens in the C standard and in the files "cpplib.h" and "cpplex.c", we determined the finite automaton for the lexical analysis by hand.

Specification in Isabelle/HOL By using the two specification possibilities described in subsection 4.1, we defined the lexical analysis in Isabelle/HOL in the two different versions.

In total, the automaton has 94 states (including a failure state) and 59 different tokens (including one for end-of-file and one failure token) can be recognized. If we include also the non-standard tokens, we get 9 extra states and 7 more tokens. The specification also includes definitions for 57 different sets of input characters used in the description of state transitions (as specified in figure 4) in the version using a state transition table; in the direct specification, only 18 such sets were necessary. Finally specifications for 98 different input characters to be coded as shown exemplarily in figure 2 were needed.

5 Checking the Lexical Analysis of the GCC

To check the lexical analysis of the GCC, we need to have access to the tokens and their values computed by it. In subsection 5.1, we describe the modifications of the GCC code necessary for that. In subsection 5.2 we explain our overall checking architecture, in particular how we have connected the Isabelle

specification with the lexical analysis of the GCC.

5.1 Modification of the Lexical Analysis of the GCC

The lexical analysis of the GNU C compiler is implemented in the library *cpplib*, in particular in the file cpplex.c and there in the function `_cpp_lex_direct`. We have modified this function so that it outputs two pieces of information for each token, its kind and its value, both separated by a whitespace and closed with a new line. Therefore we inserted the following code into the function `_cpp_lex_direct` directly before its final statement "`return result;`" [3]:

```
printf("%s",cpp_type2name(result->type));
if(TOKEN_NAME(result)!="CPP_EOF"){
     printf("%s\n",cpp_token_as_text(pfile, result));
} else { printf("EndOfFile\n"); }
```

The first line outputs the type of the token plus an additional whitespace. The subsequent lines output the value of the token. For uniformity reasons in the Isabelle specification, we also need a dummy value for the end-of-file token which is output in the else case.

The GNU C compiler introduces some standard inclusions at the beginning of each source program before compilation. This implies in turn that the stream of lexical tokens start with the tokens for these inclusions. We deal with them in the following way: We perform the lexical analysis for the empty source program and output the sequence of generated tokens and their values in a separate file. In doing so we get exactly the sequence of additionally generated tokens at the beginning of each generated token stream. When we check the correctness of the lexical analysis for a given program, we output the generated tokens and their values in a further separate file and cut off the additionally generated ones in the beginning. This resulting trimmed sequence is the one which we compare within Isabelle with the expected token sequence.

5.2 Connecting the Isabelle Checker with the Lexical Analysis of the GCC

Our overall checker architecture is shown in figure 9. We use Isabelle's program extraction facility to generate executable ML code [3,1] from our specification of the lexical analysis. We call the resulting ML code "Isabelle checker". To connect the Isabelle checker with the lexical analysis of the GCC, we need some auxiliary functions, cf. figure 9. The Isabelle checker expects its input as a list of Isabelle-coded characters, cf. subsection 4.1 and figure 2. The ML function `translate` computes this input list. The Isabelle checker outputs

[3] In our experiments, we have used the gcc-3.3 version of the GCC.

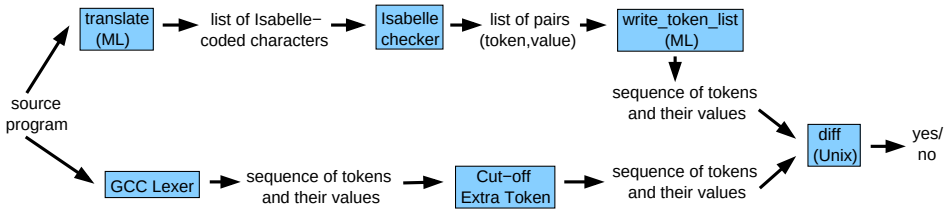


Fig. 9. Connecting the Isabelle Checker to the Lexical Analysis

the list of expected tokens and their values. This list is written back by the function `write_token_list` to the format in which also the modified GCC analysis outputs its tokens and their values. In this format, each token and its value are separated by a whitespace and closed with a new line.

The modified GCC lexer computes a sequence of tokens and their values. Thereby it computes extra EOF tokens at the end of lines containing directives for the preprocessor, cf. explanation in section 3. These tokens do not belong to the result of the lexical analysis but represent information needed in subsequent compilation phases. We deal with this situation by cutting off all EOF tokens except for the one appearing at the end of the token sequence. The resulting sequence is the one which we compare with the sequence computed with the Isabelle checker. Therefore we use the Unix program "diff".

To get a completely verified checker result, it would be necessary to formally verify the ML functions `translate` and `write_token_list` as well as the Unix function "diff" in addition to the Isabelle checker which is correct by construction. Because these functions are really simple, we have not verified them yet, a gap which can easily be closed.

Altogether, our checking architecture integrates well into the GCC system. We only need very simple modifications of the GCC code to extract the computed tokens and their values. Also, the Isabelle checker can easily be connected with the GCC code.

6 Experimental Results

In our experiments, we have specified the lexical analysis of the GCC within Isabelle/HOL in two variants as described in detail in section 4. The direct specification variant has a size of 1854 lines of specification code, the table-based variant 1856 lines. Then we have generated ML code with Isabelle's program extraction facility. The code generated from the direct specification has 1802 lines of code (loc), the code generated from the table-based variant 893 loc. Finally we have connected these generated Isabelle-checkers each with the GCC system as described in the previous section. Besides validating the

Criterion	direct spec.	table-based spec.
Readability	not really good	very clear
Efficiency wrt. Isabelle Processing	slow ($\sim$ 50 min)	fast ($\sim$ 1 min)
Efficiency of Generated ML Code	2'50 min	$>$ 20 min

Fig. 10. Experimental Results

checker by testing it for small example programs, we have also run it on a fairly large input program consisting of 2197 lines of code (one of the C files belonging to the GCC system). Our experimental results are summarized in figure 10 when run on an AMD Athlon XP 2400+.

The table-based specification variant is clearly superior concerning readability and the efficiency with which Isabelle processes it and generates machine code. But when it comes to the execution of the generated checkers, the direct specification variant is better by orders of magnitude. The scanning of the program with 2197 loc was done in less than three minutes by the direct specification variant while the table-based variant needed more than 20 minutes. We assume that the table-based version is so much more inefficient because accessing the state transition table which is implemented as a list takes so much time. In contrast, the if-then-else cascades in the direct version can be implemented much better (Isabelle's program extraction facility may replace inefficient functions by provably equivalent ones). Even though both variants are still less efficient than the original lexer of the GCC, we nevertheless think that efficiency can be increased by using Isabelle's built-in data types for characters and strings.

7 Related Work

Program checking has been used in the construction of correct compilers, most prominently in the Verifix project [5]. It has proposed program checking to ensure the correctness of compiler implementations. Program checking has been applied in the context of frontend verification [7], as already discussed in section 2. The program checking approach has also been used in further projects aiming to implement correct compilers. [9] shows how some backend optimizations of the GCC can be checked. In [12,11], the problem of constructing correct compilers is also addressed, but only for very limited applications. Only those programs consisting of a single loop with loop-free body are considered and translated without the usual optimizations of compiler construction. Those programs are translated correctly such that certain safety and liveness properties of reactive systems are sustained. In more recent work

[16], a theory for validating optimizing compilers is proposed similar to the method developed in the Verifix project. None of these projects has addressed to formally verify the frontend, in particular the lexical analysis, of a compiler widely used in practice.

Formal verification of compiler frontends has been investigated e.g. in [15] where proof fragments for the correctness of lexical and syntactic anlysis have been formalized in the mechanical theorem prover NQTHM. Formal verification of lexical analysis has also been addressed in [10]. This work shows the formal verification of a very simple lexical analyzer generator that takes a regular expression and yields a functional lexical analyzer. Emphasis in this work is placed on the formal verification of transforming a regular expression into an initially nondeterministic automaton and then into a deterministic one. In future work we plan to connect our checking architecture with the theory described in this work. In doing so, we would only need to extract the regular grammar for the lexical analysis and could construct the corresponding deterministic automaton as described in this related work.

8 Conclusions

With the results presented in this paper, we have set up an architecture for checking results of frontend computations, in particular for the lexical analysis. We have shown how to specify the task of scanning tokens within Isabelle/HOL. Furthermore, we have demonstrated how to access the intermediate results of the GCC system by minimally modifying its C source code. In doing so, we have shown that program checking can handle large non-academic compilers which are extensively used in practice. The presented checking method is not specific to the GCC nor to the C langugage and can be applied to other compilers as well. In future work, we plan to connect our checking architecture with the results of [10] which provides a formal proof in Isabelle/HOL for the generation of a functional lexical analyzer (i.e. a finite automaton) given a regular grammar. Since the regular grammar for the C tokens is directly given in the C language specification (as is the case for the specifications of many other programming languages as well), this will yield a completely verified result of the lexical analysis.

Acknowledgments: Thanks to Jan Olaf Blech for many helpful discussions. Also thanks to the anonymous reviewers for their supportive comments. This work was supported by a research grant within the “Eliteförderprogramm für Postdoktoranden der Landesstiftung Baden-Württemberg”.

References

[1] Stefan Berghofer. Program Extraction in simply-typed Higher Order Logic. In *Types for Proofs and Programs, International Workshop, (TYPES 2002)*. Springer Verlag, LNCS, Vol. 2646, 2003.

[2] Manuel Blum and Sampath Kannan. Designing Programs that Check Their Work. *Journal of the ACM*, 42(1):269–291, 1995. Preliminary version: *Proceedings of the 21st ACM Symposium on Theory of Computing (1989)*, pp. 86-97.

[3] Stefan Berghofer and Tobias Nipkow. Executing Higher Order Logic. In *Types for Proofs and Programs, International Workshop, (TYPES 2000)*. Springer Verlag, LNCS, Vol. 2277, 2002.

[4] Neil Booth. Cpplib Internals. Free Software Foundation, Inc., 2002.

[5] W. Goerigk, A. Dold, T. Gaul, G. Goos, A. Heberle, F.W. von Henke, U. Hoffmann, H. Langmaack, H. Pfeifer, H. Ruess, and W. Zimmermann. Compiler Correctness and Implementation Verification: The Verifix Approach. In P. Fritzson, editor, *Poster Session of CC'96*. IDA Technical Report LiTH-IDA-R-96-12, Linkoeping, Sweden, 1996.

[6] Sabine Glesner. Using Program Checking to Ensure the Correctness of Compiler Implementations. *Journal of Universal Computer Science (J.UCS)*, 9(3):191–222, March 2003.

[7] A. Heberle, T. Gaul, W. Goerigk, G. Goos, and W. Zimmermann. Construction of Verified Compiler Front-Ends with Program-Checking. In *Perspectives of System Informatics, Third Int'l Andrei Ershov Memorial Conference, PSI'99*, Russia, 1999. Springer LNCS, Vol. 1755.

[8] ISO/IEC. International Organization for Standardization, International Standard ISO/IEC 9899:1999, Programming languages – C. Reference number ISO/IEC 9899:1999(E), 1999. Second edition 1999-12-01.

[9] George C. Necula. Translation Validation for an Optimizing Compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*, 2000.

[10] Tobias Nipkow. Verified Lexical Analysis. In *Theorem Proving in Higher Order Logics*. Springer Verlag, LNCS Vol. 1479, 1998. Invited talk.

[11] A. Pnueli, O. Shtrichman, M. Siegel. The code validation tool (cvt.). *Int'l Journal Software Tools for Technology Transfer*, 2(2):192–201, 1998.

[12] A. Pnueli, M. Siegel, and E. Singermann. Translation validation. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, 1998. Springer Verlag, LNCS Vol. 1384.

[13] Richard M. Stallman. GNU Compiler Collection Internals. Free Software Foundation, Inc., 2002.

[14] Richard M. Stallman and Zachary Weinberg. The C Preprocessor. Free Software Foundation, Inc., 2001.

[15] D. Weber-Wulff. *Contributions to Mechanical Proofs of Correctness for Compiler Front-Ends*. PhD thesis, Christian-Albrecht-Universitäacrot zu Kiel, Germany, 1997. available as technical report "Bericht Nr. 9707".

[16] L. Zuck, A. Pnueli, and R. Leviathan. Validation of Optimizing Compilers. Technical Report MCS01-12, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, August 2001.