Certified Merger for C Programs Using a Theorem Prover: A First Step

Yuki Goto
School of Science & Technology
Kwansei Gakuin University
2-1, Gakuen, Sanda, 669-1337, JAPAN

Kazuko Takahashi
School of Science & Technology
Kwansei Gakuin University
2-1, Gakuen, Sanda, 669-1337, JAPAN

Abstract

This paper proposes a method of using theorem provers for supporting system development. As a case study, we construct a certified merger for C programs. The merger is implemented and verified by a theorem prover Isabelle/HOL. It also has a front-end and back-end which connect to C programs. It provides a useful tool for users who are not familiar with theorem provers. It can be applied to merging test suits used in the verification stage.

Keywords: theorem prover, merger, system verification, software development

1 Introduction

Recently, the size of software becomes larger and more complicated controls are contained in it. Some of them are operated in parallel or distributed environment. Moreover, hardware and software should be totally handled in case of embedded systems. These facts cause to raise both importance and difficulty in developing a reliable software.

It is proposed that formal methods [5] such as model checking or theorem proving should be introduced which can provide a formally specified and mechanically proved software. In such a sophisticated approach, validity for all possible cases can be guaranteed based on logic or mathematics, which cannot be covered by a usual test-based approach. However, formal methods are not so popular in an actual development scene although the cases they are partly introduced are increasing in number. The main reason is that most software developers are not familiar with such a formalization, and high expertized knowledge and long-term training are required to manage these tools. Above all, theorem proving technique is stronger to handle infinite data, but it is harder to get used to it. In addition, users other than the developer cannot analyze the detail of software in some cases because of its inner complicated structure and a huge size. For example, we cannot represent full specification of a commercial C compiler such as *gcc*, of which several bugs have been reported so far [17]. It means that system test is still valid and frequently used in the verification stage.

In a test stage, not only a simple data but also structured data such as a program is used as input data. These data are sometimes combined or merged into one program automatically to increase efficiency in the test stage [4]. The correctness of such a merger is more important than that of the target software to be verified, since if the test program is incorrect, when an error occurred in a test, it cannot be decided whether it is due to the target software or the test program.

In this paper, we take the verification of a merger of test programs for C compiler as a case study. We implement it and prove its correctness using a theorem prover Isabelle/HOL [14], and construct a certified merge system for C programs.

Merger combines a set of programs consistently. It is not a simple combination, since the duplication of names of variables and functions should be eliminated. As the main job of the merger is the renaming, that is an operation on characters, the target of the formalization is program syntax rather than semantics. In this paper, we take such a syntactical correctness of a merger. In the proof procedure, we clarify the condition to be satisfied. If a sufficient condition is not specified, the proof does not succeed. It makes the user to reconsider the condition to be added. This cannot be done in a usual unproved merger.

Our merger consists of three parts. First, given pair of C programs are converted to the representation of Isabelle/HOL in the front-end process. Second, the converted programs are merged by the certified merger on Isabelle/HOL in the main part. Third, the result is converted to C code and written onto the file in the back-end process.

Since Isabelle/HOL is based on a functional language, programming style is different from popular C language. However, our merger system has the interface with C program, even the users familiar only with

978-1-880843-90-1 / ISCA CATA / March 2013

C language feel less burden to use it. It can contribute to increase efficiency of system development.

The rest of this paper is organized as follows. In Section 2, we describe a theorem prover Isabelle/HOL. In Section 3, we describe our certified merger in Isabelle/HOL. In Section 4, we explain the merger system. In Section 5 we show the related works. Finally, in Section 6, we present our conclusions.

2 Isabelle/HOL

Theorem proving is a technique for formal methods and several tools have been developed such as Isabelle/HOL [14], Coq [3], ACL2 [7] and PVS [15]. They are sometimes called proof assistants, since the proof usually goes not full-automatically, but needs a navigation of a user. There are many applications for theorem proving, from pure mathematics to practical hardware verification or security protocols, and many successful results have been reported [6].

Here, we use Isabelle/HOL, since it has a strong simplification mechanism, which helps to make a proof semi-automatic. It is a theorem-proving environment that is part of the HOL system. Data types and functions are defined in recursive form. If a function is not well-defined, it is not accepted. The proof goes backwardly. In other words, the prover first tries to verify the goal shown by the user; if it cannot be proved directly, the prover generates subgoals using tactics such as resolution and assumption. The prover repeats this process for each subgoal, and if all the subgoals are proved to be true, the verification terminates.

3 Certified Merger in Isabelle/HOL

3.1 Overview of merger

For a pair of C programs, we set one of them as a primary program and the other is a secondary program. The merger merges them by adding statements of a secondary one to the structure of a primary one. It performs renaming for the secondary program as follows: it changes names of functions (including "main") and those of global variables by adding the file name of the secondary program as a pre-sequence; it makes the renamed "main" function of the secondary program to be called in the "main" function of the merged program; in addition, it changes all the names of the elements effected by these renaming.

Figure 1 shows an example. Let F be a file name of a secondary program. In Figure 1, both of a primary and a secondary program have a global variable V. The global variable name in the secondary program

```
% primary program
int V;
int main(void)
  { V = 1; }
```

```
% secondary program
int V;
int main(void)
  { V = 0; }
```

```
% merged program
int V;
int FV;
int main(void) {
  Fmain();
  V = 1;
}
int Fmain() {
  FV = 0;
}
```

Figure 1: An example of merging two programs

is changed to FV, and "main" in the secondary program is renamed to "Fmain". Note that if FV occurs in the primary program and V occurs in the secondary program, then duplication appears in the merged program.

3.2 Formalization

We represent formally a pair of C programs and the merger in Isabelle/HOL. If occurrence of variables and function types are inconsistent, the definition is not accepted. Since what merger does is a syntactical transformation, if a definition is accepted, then basically syntactical correctness is guaranteed.

Each C program consists of a quad-ruple of $FileName$, $GvarList$, $Funcs$ and $Main$. It is defined as s datatype $Prog$.

```
Prog = "(FileName * GvarList * Funcs * Main)"
```

$FileName$ is a file name the program is stored.

$GvarList$ is a list of global variables which are declared in the program.

$Funcs$ is a list of definitions of functions. Each definition consists of output type, a function name, a list of arguments with their types and a function body that is a list of the statements.

$Main$ is a body of the "main" function.

Since Isabelle/HOL is based on functional language, the computation model is based on reduction, that is, the input term is rewritten to the output term in the computation.

Hereafter, $V_1 + V_2$ denotes the connection of strings of variables V_1 and V_2. Let F be a file name of a secondary program.

$merge$ is composed of four functions $merge\_func$, $merge\_gvar$, add_change_main1 and add_change_main2. All these functions including $merge$ is a function from $Prog$ and $Prog$ to $Prog$.

```
fun merge :: "Prog ⇒ Prog ⇒ Prog" where "merge pr1 pr2 =
merge_func (merge_gvar (add_change_main1 (add_change_main2 pr1 pr2) pr2) pr2) pr2)
```

For *Main* of *pr*2, (*add_change_main*2 *pr*1 *pr*2) changes the name CF of function in each function call to F+CF, changes the name GV of each global variable to F+GV, sets the result as the function body of a new function; sets F+"main" as the function name of a new function; adds the definition of this new function to *Funcs* of *pr*1 and outputs the resulting program.

(*add_change_main*1 *pr*1 *pr*2) adds the function call F+"main" to "Main" of *pr*1, and outputs as the resulting program.

(*merge_gvar pr*1 *pr*2) changes the name VN of each element of *GvarList* of *pr*2 to F+V and outputs as the resulting program.

For each function of *Funcs* in *pr*2, (*merge_func pr*1 *pr*2) changes the name FN to F+FN, changes each function call CF to F+CF if there is any in its body, adds the result to *Funcs* of *pr*1 and outputs the resulting program.

```
fun change_funcs :: "Prog ⇒ Funcs ⇒ Funcs" where "change_funcs pr xs =
callfunc_changename2 pr (funcs_changename pr xs)"
```

This code shows the definition of the function according to the final step of *merge_func*. The function is given *Prog* and *Funcs* as the inputs, and outputs *Funcs*. It is a composition of functions *funcs_changename* and *callfunc_changename*2.

The main points of the formalization in Isabelle/HOL is as follows.

We define a new data type of string as a list of characters. Although string is already set in the library, it includes a lot of definitions such as "Nibble" that are unnecessary in our merger, which sometimes makes a proof procedure complicated. Thus, we define the minimum string as new model for easy treatment of a string.

We use list as a data type in lots of definitions. Usage of the higher-order function helps much in the proof.

```
fun gvar_changename :: "Prog ⇒ GvarList ⇒ GvarList" where "gvar_changename pr gs =
map(λx. case x of (GV gvtype gvname) ⟹ (GV gvtype ((fst pr) @ gvname) )) gs"
```

In this example, an anonymous function adds the file name denoted by (*fst pr*) to an element *gs* in the form of (*GV gvtype gvname*) of *GvarList*. The function *map* takes this anonymous function as an argument and applies it on all the elements of *GvarList*.

3.3 Proof

The correctness of merger is specified in several points: renaming is consistent, no duplication appears in the variable names and function names in the resulting program, the relationship between caller and callee are preserved. We show some lemmas stating these points.

Let F be a file name of a secondary program.

If a primary program and a secondary program have no duplication in the names of global variables, respectively, there is no variable V that occurs in the secondary program and F+V occurs in the primary one, and the secondary program is not empty, the merged program has no duplication.

```
lemma no_duplication :
(distinct (Prog2GvStrl pr1) ∧ distinct (Prog2GvStrl pr2) ∧ ( ∀ a. ( (a ∈ set (Prog2GvStrl pr1)) ⟶ (hd a ≠ hd (fst pr2)))) ) & fst pr2 ≠ [ ] ⟹
distinct (Prog2GvStrl (merge_gvar pr1 pr2))
```

Here, ⟶ denotes the logical connection used in the higher-order logic, where ⟹ denotes an inference rule.

In our experience, first an error occurred in the proof since we missed the last condition. We succeeded in the proof after adding this condition. If a merger is not certified, this condition may be missing.

Renaming of global variables in the secondary program is correctly done. That is, the variable name is changed to be the one added by the file name on its head. Note that @ denotes the operation of list-append.

```
lemma correct_renaming_of_gvs :
Prog_GvarList (merge pr1 pr2) =
(Prog_GvarList pr1) @ (gvar_changename pr2 (Prog_GvarList pr2))
```

The head of the "main" function of the merged program is the statement that calls the function named F+"main".

```
lemma : correct_trans_of_main
(hd (Func_StatementList (Prog_Main (merge pr1 pr2))) ) =
(CallFunc ( (fst pr2) @ (Func_Name (Prog_Main pr2))) [ ] )
```

The body of the function named F+"main" is included in the list of definitions of the functions.

```
lemma : correct_trans_of_body
(Statement2CallFuncName (hd (Func_StatementList (Prog_Main (merge pr1 pr2)) ))) ∈ set (Prog2FuncNamel (merge pr1 pr2))
```

The merger in Isabelle/HOL consists of about 535 lines including all definitions and lemmas, and the number of proved lemmas is 11.

4 The System

4.1 System configuration

The certified merger system for C programs consists of three parts: front-end, main part and back-end (Figure 2).

Front-end is a converter from a pair of C programs to Isabelle/HOL representation, main part is a certified merger written in Isabelle/HOL, and back-end is a converter from the result of merging represented in Isabelle/HOL to the corresponding C program. Front-end and back-end are implemented in C.

4.2 Representation of an object program

The target C program is a subset of the complete C program (Figure 3). It covers the essential elements such as assignment statements, control statements, function calls, but does not handle the data type of array, structure, pointer, and statements such as "goto" and "break." The one reason is that the first goal of this work is to show the validity of treatment of merger in Isabelle/HOL, and the other reason is that motivated application is merging test programs, each of which consists of these fundamental elements. The method shown here can be easily extended for the full language specification.

4.3 Conversion from C program to Isabelle/HOL

C program consists of three parts: declaration, "main" function and the list of functions. The front-end processor reads source code of C, extracts declaration of the global variables, definitions of functions and stores them in the corresponding arrays, respectively. Then, it transforms them to the program data in Isabelle/HOL.

Figure 4 shows the image of front-end process.

4.4 Conversion from Isabelle/HOL to C program

As Isabelle/HOL is a prover, it basically proves the given lemma and produces only its result, that is, *true* or $false$. We use "value" command which evaluates the computation to get the resulting values of variables corresponding to a list of global variables and

```
program ::= (defVar ";")* (decFun ";")*
            defMain (defFun)*
defVar ::= type varName
decFun ::= type funName "(" ( "void" |
           decVar ( "," decVar)* ) ")"
defFun ::= type funName "(" ( "void" |
           decVar ( "," decVar)* ) ")" funBody
defMain ::= "int" "main" "(void)" funBody
funBody ::= "{" stmt* "}"
type ::= "int" | "double"
stmt ::= ϵ | funCall ";" | assStmt | decVar
assStmt ::= expLhs "=" exp ";"
Lhs "=" exp ";"
exLhs ::= varName
varName ::= CHAR (CHAR)*
exp ::= INT | CHAR | funCall | refVar
refVar ::=  varName
funCall ::= funName "(" argList ")"
funName ::= CHAR (CHAR)*
argList ::= (ϵ | exp ("," exp))*
CHAR ::= "A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|
         "J"|"K"|"L"|"M"|"N"|"O"|"P"|"Q"|"R"|
         "S"|"T"|"U"|"V"|"W"|"X"|"Y"|"Z"
INT ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

Figure 3: Syntax of a target program

definitions of functions, respectively. As Isabelle/HOL is processed using the Emacs interface called "Proof General," we write this result onto the file using Emacs command. The data is stored in the corresponding arrays, and the resulting C code is generated from this data. Finally, we get the merged C program.

4.5 Application to test suits

In software development, a test suite (validation suite) is a collection of test cases. It is intended to be used to show that software program has some specified set of behaviors [4]. When we want to perform multiple test programs stored separately, we combine them to the single program to reduce the times of file opening/closing. In such a case, a merger is used.

We applied the certified merger to sample programs in test suits and confirmed that the merged program generates the same output with the one when each test program is executed separately.

5 Related Works

The merger we developed is expected to be used in the test process of a compiler verification.

The beginning of the application of theorem proving technique to the compiler verification is back to Moore's earlier work [13]. Afterwards, there have been several studies on the verification of compilers.

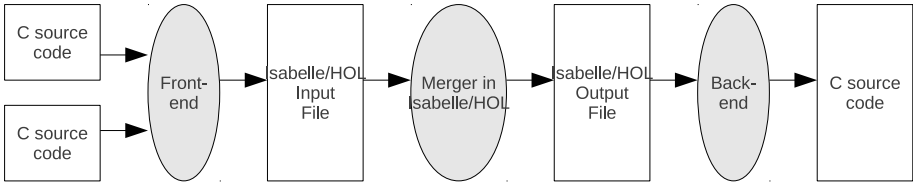

Figure 2: A certified merger: system configuration

Leinenbach shows the correctness of C0 compiler, as a subset of C, using Hoare logic in Isabelle/HOL [8]. They studied it in the Verisoft project aiming at pervasive verification of entire computer systems from hardware to software.

Leroy constructed certified C compiler using Coq [9]. It is an elaborating work that can be applied to practical use. They handle all the semantics including register allocation and data flow. In their system, an input Cminer program, a large subset of C, is transformed into machine language through several intermediate languages. They proved that the correct code is generated. Afterwards, they extend the specification of the target C programs and verification of front-end and back-end processors [1, 10, 11].

Strecker showed the verification of Java compiler written in Isabelle/HOL [16]. They also prove that the back-end processor generates the correct code [2].

Zimmerman presented a proof approach based on abstract state machines (ASM) for bottom-up rewriting system specifications for back-end generators [18]. These strategies are implemented in PVS to generate a correct compiler. They verified the correctness of transformation from an intermediate language to the machine language. Their main purpose is to provide a compiler that generates the code for a real-life processor. Although they use PVS, the entire proof contains manually proved parts.

Mansky showed the verification of the optimization of the compiler. They verified the correctness of the optimizer defined on a graph representation for the compiler using Isabelle/HOL [12].

Most of these works prove the correctness of the compiler from the semantic point of view. It is one possibility of applying theorem proving technique in system verification. On the other hand, we take another approach of proving the correctness of software used in the test stage. In this case, proof on the syntax level is appropriate. There have been no works on application of theorem provers to the syntax level proof, and no certified merger has been provided so far.

6 Conclusion

We have developed the certified merger for C programs in Isabelle/HOL. If a function is not well-defined, or some condition is missed, the error is reported, whereas a usual merger passes through such a case. That is the advantage of certified merger. Use of a theorem prover can clarify the conditions that input programs should satisfy.

Interface with C program enables users not familiar with theorem proving to use the system easily. This is a way of applying theorem provers to a support on system development.

There is a restriction on the target C program. We are considering to loosen them. Proving the front-end and back-end parts is more challenging. We have to make a different model of the merger since we have to consider semantics to prove them. We consider it as a second step of our work.

Although we showed the application to merger of test suits here, other mergers, for example, merging LaTeX programs, can be constructed and verified in the same way. We are considering to make the set of definitions and proofs of Isabelle/HOL in the sophisticated form and to publish it as a library.

In future, we will introduce an optimizing process, for example, elimination of redundant variables and the functions with the same definitions, to this certified merger.

References

[1] G. Barthe, D. Demange and D. Pichardie, A formally verified SSA-based middle-end Static Single Assignment meets CompCert, *ESOP12*, pp.47-66, 2012.

[2] S. Berghofer and M. Strecker, Extracting a formally verified, fully executable compiler from a proof assistant, *COCV03*, pp.33-50, 2003.

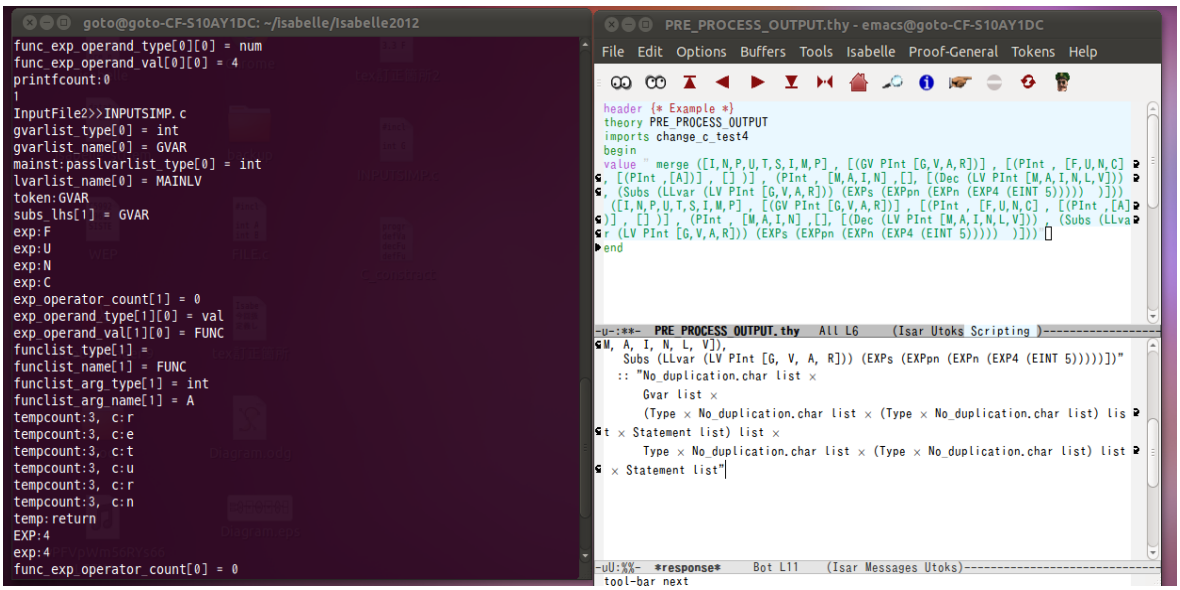

Figure 4: Screenshot of front-end process

[3] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development: Coq'art-The Calculus of Inductive Constructions*, Springer, 2004.

[4] C. Cheng, The test suite generation problem: Optimal instances and their implications, *Discrete Applied Mathematics*, Vol.155, No.15, pp.1943-1957, 2002.

[5] Formal methods: state of the art and future directions, E. Clarke and Jeannette. Wing *JACM Computing Surveys*, Vol.28, No.4, pp.626-643, 1996.

[6] H. Geuvers, Proof assistants: History, ideas and future, *Sadhana : Academy Proceedings in Engineering Sciences (Indian Academy of Sciences)*, Vol.34, No.1, pp.3-25, 2009.

[7] M. Kaufmann, P. Monolios and J. Moore, *Computer-Aided Reasoning: An Approach*, Kluwer Academic Publishers, 2000.

[8] D. Leinenbach, W. Paul and E. Petrova, Towards the formal verification of a C0 compiler: Code generation and implementation correctness, *SEFM05*, pp.2-12,2005 .

[9] X. Leroy, Formal certification of a compiler back-end or: programming a compiler with a proof assistant, *POPL06*, pp.42-54, 2006.

[10] X. Leroy, Formal verification of a realistic compiler, *Communications of the ACM*, Vol.52, No.7, pp.107-115, 2009.

[11] X. Leroy, A formally verified compiler back-end, *Journal of Automated Reasoning*, Vol.42, No.4, pp.363-446, 2009.

[12] W. Mansky and E. Gunter, A framework for formal verification of compiler optimizations, *ITP10*, pp.371-386, 2010.

[13] J. Moore, A mechanically verified language implementation, *Journal of Automated Reasoning*, Vol.5, pp.461-492, 2009.

[14] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL A Proof Assistant for Higher-Order Logic*, Springer, 2008.

[15] S. Owre, J. Rushby and M. Shankar, PVS: A prototype verification system, *CADE92*, pp.748-752, 1992.

[16] M. Strecker, Formal verification of a Java compiler in Isabelle, *CADE02*, pp.63-77, 2002.

[17] X. Yang, Y. Chen, E. Eide and J. Regehr, Finding and understanding bugs in C compilers, *PLDI11*, pp.283-294, 2011.

[18] W. Zimmermann and T. Gaul, On the construction of correct compiler back-ends: an ASM-approach, *Journal of Universal Computer Science*, Vol.3, No.5, pp.504-567, 1997.