Design of a Certifying Compiler Supporting Proof of Program Safety

Yiyun Chen Lin Ge Baojian Hua Zhaopeng Li Cheng Liu
Department of Computer Science and Technology
University of Science and Technology of China
Hefei, Anhui 230026, China
yiyun@ustc.edu.cn {*gelin, huabj, zpli, liuc5*}*@mail.ustc.edu.cn*

Abstract

Safety is an important property of high-assurance software, and one of the hot research topics on it is the verification method for software to meet its safety policies. In our previous work, we designed a pointer logic system and proposed a framework for developing and verifying safety-critical programs. And in this paper, we present the design and implementation of a certifying compiler based on that framework. Here we will mainly explain verification condition generation, generation of code and assertions, and proof generation for basic blocks. Our certifying compiler has the following novelties: 1) it supports a programming language equipped with both a type system and a logic system; 2) and it can produce safety proofs for programs with pointers.

1. Introduction

Nowadays, high-assurance software is more and more valued, and among its properties, safety and security are most important. Safety is the ability to guarantee that software execution does no harm to the system. And security is the ability to preserve the confidentiality, integrity, availability and authentication of data. There is a close relationship between safety and security. For example, hackers usually use low-grade safety errors, such as buffer overflows, out-of-bound array accesses and dangling pointer dereferences, to destroy a system or achieve unauthorized control of a system. Apparently, it is helpful to ensure software security by strengthening its safety, and it makes us focus on software safety in this paper.

To achieve software safety, all program errors should be discovered before the execution of the program or be gently captured during the execution. The research goal of software safety is to build a wholesome scientific and technological infrastructure for the management of software safety. And the verification method for software to meet its safety policies is one of the hot topics in this research field.

In the past decade, great progress has been made in the area of program verification. Many projects adopt type-based or logic-based approaches to reason about the properties of low-level or high-level programs. The TAL (Typed Assembly Language) [15] project and the theory of type refinements [13] are two typical projects using type-based approaches, while Proof-Carrying Code (PCC) [16], Foundational PCC (FPCC) [1], Certified Assembly Programming (CAP) [19] and Stack-based CAP (SCAP) [6] are typical projects on logic-based techniques. Type-based and logic-based techniques are complementary to each other, and some projects have tried to combine them. Hamid *et al.* adopted a syntactic approach to FPCC [7], and gave a translation from a typed assembly language into FPCC in the Flint project. The ATS (Applied Type System) [18] project proposed by Hongwei Xi *et al.* extends a type system with a notion of program states, so that invariants on states can be captured in stateful programming. By encoding Hoare logic in its type system, ATS can support Hoare-logic-like reasoning via the type system.

Based on the above work, we have proposed a new framework for developing and verifying safety-critical software:

1. Safety policies of a program are specified formally by programmers. These specifications as well as the corresponding program are submitted to a compiler.
2. The compiler produces verification conditions (VCs) for the submitted program. These VCs are required to verify whether the program satisfies its specifications. If all of them are proved, the program is believed to satisfy the specifications. Most VCs can be proved automatically by the embedded theorem prover in the compiler, and the rest are proved interactively by programmers using proof-assistant tools. All the proofs should be produced and preserved.
3. The compiler compiles the source code and specifications into assembly level counterparts while translating

the proofs obtained from step 2 into proofs at assembly level. The latter proofs are carried in the assembly code and they ensure that the code meets the corresponding specifications. These specifications must be equivalent to the counterparts at source level. Such a compiler is called a certifying compiler.

4. At assembly level, a proof checker checks the proofs carried in the assembly code automatically to ensure that the code satisfies its specifications.

The advantage of this framework is that it provides source-level approach for reasoning about program properties rather than the assembly-level one. Compared with the approaches for assembly code certification, this approach can improve the efficiency of safety-critical software development. Since the proofs are checked at assembly level, the compiler, including a VC generator and a prover, can be removed from the TCB (Trusted Computing Base). Therefore, the TCB of the system will be reduced remarkably.

In this framework, we choose a C-like programming language PointerC [9] as our source language and implement a certifying compiler prototype for it. Our main contribution is the design of a certifying compiler for a source language with pointers and explicit memory management (malloc/free). And a prototype of the certifying compiler has been implemented in our work. A distinct feature of our certifying compiler is that it supports languages equipped with both a type system and a logic system.

In this paper, we introduce the design and implementation of our certifying compiler. The rest of the paper is organized as follows. We introduce the source language—PointerC briefly in section 2. In section 3, we discuss how the compiler generates VCs at source level from the assertions given by programmers and the side conditions in the typing rules. Section 4 describes the scheme for generating corresponding assertions at some program points during intermediate code generation. This ensures that each basic block has proper pre- and postconditions. Section 5 explains how the compiler generates a proof for each basic block. Each proof ensures that the corresponding basic block satisfies its pre- and postconditions. Section 6 compares our work with related work and section 7 concludes.

2. Source language—PointerC

PointerC is a C-like programming language that we have designed as the source language in our work. In order to check more pointer programs statically, some pointer operations are restricted in PointerC [9]. These restrictions are based on the premise of not affecting the language functionality of PointerC. Variables with pointer type can only be used in assignment, equality comparison, dereference or as the parameters of functions; the address-of operator (&) and pointer arithmetic are forbidden. To guarantee that type-checked programs are well-typed, union types and coercion of types are also forbidden. malloc and free are used as predefined functions which meet the primary safety policies. For example, each invocation of malloc will allocate an area of memory successfully, and the region of the memory will never overlap those already in use.

Pointers are classified into **effective pointers** (those point to objects), null pointers and dangling pointers, and the latter two are also called **ineffective pointers**. Ineffective pointer dereferences, memory leaks or using an ineffective pointer as the actual parameter of function free are all considered as violations of the primary safety policies.

A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. It is mainly used to eliminate context-sensitive errors in programs. A traditional type system is not sufficient when the legality of a phrase depends not only on the context but also on some expression values in the phrase. Dependent types [18] is one possible solution to this problem. Another solution is not to treat the constraints on values as parts of the type system, and the constraints don't appear in the typing rules. The former leads to a complicated type system and the latter leads to an unsound one. We try to find a trade-off between these two solutions in the design of PointerC. To achieve simplicity in the type system and guarantee the safety of the language at the same time, side conditions are introduced into the typing rules to express the constraints on values. For example, the following two typing rules show that the constraints on subscript expressions and pointers are put in the side conditions.

$$\frac{\Gamma\vdash e:\texttt{int} \quad \Gamma\vdash a:\texttt{array(10, bool)}}{\Gamma\vdash a\texttt{[}e\texttt{]}:\texttt{bool}}\ 0\le e\wedge e\le 9$$

$$\frac{\Gamma\vdash p:\texttt{ptr(struct(...,}x\texttt{:int,...))}}{\Gamma\vdash p\texttt{->}x:\texttt{int}}\ p\in \mathit{effective\_ptrs}$$

These side conditions must be checked before the execution of the program to ensure that the execution does not violate the primary safety policies. And to check these side conditions statically, we have designed a pointer logic for PointerC. The pointer logic is an extension of Hoare logic and essentially is a pointer analysis tool. It collects pointer information in a forward manner. This information includes whether a pointer is null, dangling or effective, and the equality relation between effective pointers. The information can be used to prove that the program satisfies the side conditions in the typing rules and then to support value-sensitive static checking. For example, according to the pointer logic, the following two Hoare triples

$$\{\!|\{p,q\}\wedge\{p\texttt{->}\mathit{next}\}|\!\}\ p{=}\mathsf{malloc}(\cdots)$$
$$\{\!|\{p\}\wedge\{q\}\wedge\{q\texttt{->}\mathit{next}\}\wedge\{p\texttt{->}\mathit{next}\}_{\mathcal{D}}|\!\}$$

and

$$\{\!|\{p,q\} \wedge \{p\text{->}next\}_{\mathcal{N}}|\!\}\ \mathsf{free}(p)\ \{\!|\{p,q\}_{\mathcal{D}}|\!\}$$

hold, where the set $\{p\}$ represents an assertion, and its informal meaning is "p is an effective pointer and is not equal to any other pointers". Similarly, the set $\{p,q\}$ means "Effective pointers p and q are equal (but not aliases), and they are not equal to any other pointers"; $\{p,q\}_{\mathcal{D}}$ denotes p and q are dangling pointers; and $\{p\text{->}next\}_{\mathcal{N}}$ means that $p\text{->}next$ is a null pointer. According to the pointer logic, there are no postconditions Q_1 and Q_2 which make the following two Hoare triples hold. That is, the preconditions of malloc and free here can not satisfy the side conditions in the corresponding typing rules.

$$\{\!|\{p\} \wedge \{p\text{->}next\}_{\mathcal{N}}|\!\}\ p{=}\mathsf{malloc}(\cdots)\ \{\!|Q_1|\!\}$$

(The object previously pointed by p is lost.)

$$\{\!|\{p\} \wedge \{p\text{->}next\}|\!\}\ p{=}\mathsf{free}(p)\ \{\!|Q_2|\!\}$$

(The object previously pointed by $p\text{->}next$ is lost.)

In history, pointer analyses were mostly conservative estimates of the pointer status at runtime. And they mainly answered the question: what was the possible set of objects that a pointer may point to at runtime? Such pointer analyses can be used in many fields of static analysis and program optimization, such as liveness analysis needed by register allocation and constant propagation *etc.* To meet the safety requirements of software, we have restricted some undecidable pointer operations in PointerC, and thus obtained an accurate pointer analysis instead of an approximate one. This is the primary difference of our pointer analysis from others. Although accurate pointer analysis does not exist for all programs, our practical experience on the applications of some data structures, such as singly linked list, binary tree and circular doubly linked list *etc.*, shows that the pointer logic is feasible. Besides side conditions in the typing rules, the pointer logic can also be used to reason about other program properties to accommodate the requests of user-defined safety policies.

3. Verification condition generation

In order to check program safety statically, side conditions in the typing rules must be provable at the corresponding program points. This is achieved through the following steps:

1. Programmers annotate each function with a pair of pre- and postconditions and each while loop with a loop invariant. These annotations belong to the specifications of the source program.
2. A verification condition generator (VCGen) is embedded into the front end of the compiler. It can convert the task of proving a program satisfying its specifications into the task of proving a set of VCs. From the annotations mentioned in 1 and the side conditions in the typing rules, the VCGen generates a set of VCs using the pointer logic rules. And these VCs should be proved in order to guarantee program safety.
3. A simple theorem prover, which produces corresponding proofs for pointer-related VCs, is embedded into the compiler as well. Integer-related VCs are proved interactively in Coq [4] by the programmers. The VCs and their proofs show that the source program satisfies its specifications.

At first, we discuss how to generate VCs for functions with pointer-type data. Figure 1 shows the structure of a function chosen from PointerC syntax, some irrelevant details are omitted.

$$\begin{array}{rcl}
\mathit{FunDcl} & \rightarrow & \mathit{id}(\mathit{arg})\{\mathit{Body}\} \\
\mathit{Body} & \rightarrow & \mathit{VarDecList}\ \mathit{StmtList} \\
\mathit{StmtList} & \rightarrow & \mathit{Stmt}\ \mathit{StmtList} \mid \epsilon \\
\mathit{Stmt} & \rightarrow & \mathit{lval} = \mathit{Exp} \\
 & \mid & \mathsf{if}\ (\mathit{Bexp})\ \{\mathit{StmtList}\}\ \mathsf{else}\ \{\mathit{StmtList}\} \\
 & \mid & \mathsf{while}\ (\mathit{Bexp})\ \{\mathit{StmtList}\} \\
 & \mid & \mathit{lval} = \mathsf{alloc}(\mathit{Type}) \\
 & \mid & \mathsf{free}(\mathit{Exp}) \\
 & \mid & \mathit{lval} = \mathit{id}(\mathit{Exp}) \\
 & \mid & \mathsf{return}\ \mathit{res}
\end{array}$$

Figure 1. Structure of function

The pointer logic is fit for collecting pointer information in a forward manner, so that the VC generation is based on the strongest postcondition calculus. Figure 2 shows the main rules for the strongest postcondition calculus (function sp) in the pointer logic and the framework of VC generation (for data with pointer types). These rules are recursively defined according to the syntactic structures in a function. In Figure 2, Π, $\mathcal{N}$ and $\mathcal{D}$ denote the equivalent division of all effective pointers, the set of null pointers and the set of dangling pointers respectively [2]; Ψ is the short notation for $\Pi \wedge \mathcal{N} \wedge \mathcal{D}$; the first parameter of function sp is a syntactic structure, and the second one is the precondition of the syntactic structure. The VC generation described here has the following important characteristics:

1. Since the precondition Ψ and the postcondition Ψ' of a function are given, the VC $\mathsf{sp}(\mathit{StmtList}, \Psi) \supset \Psi'$ is generated at the exit of the function (note that *StmtList* forms the statement list of the function).

1. Function definition

   $\{\Psi\}$ $id(arg)\{Body\}$ $\{\Psi'\}$, that is $\{\Psi\}$ $StmtList$ $\{\Psi'\}$, where the $StmtList$ is the $StmtList$ in the $Body$ production.

2. Statement List

   - $\mathsf{sp}(Stmt\ StmtList,\ \Psi) = \mathsf{sp}(StmtList,\ \mathsf{sp}(Stmt,\ \Psi))$
   - $\mathsf{sp}(\epsilon,\ \Psi) = \Psi$. If ϵ forms the $StmtList$ of a function, then VC $= \Psi \supset \Psi'$ (see 1 for Ψ') will be generated.

3. Statement

   - assignment: $\mathsf{sp}(lval = Exp,\ \Psi) = \Psi'$, Ψ' can be calculated using Ψ according to the assignment rules in the pointer logic.
   - condition: $\mathsf{sp}(\mathsf{if}\ (Bexp)\ \{StmtList_1\}\ \mathsf{else}\ \{StmtList_2\},\ \Psi) = \mathsf{sp}(StmtList_1,\ Bexp \wedge \Psi) \vee \mathsf{sp}(StmtList_2,\ \neg Bexp \wedge \Psi)$
   - loop: $\mathsf{sp}(\mathsf{while}\ (Bexp)\ \{StmtList\},\ \Psi) = \neg Bexp \wedge I$, where I is the loop invariant for pointer-type data.
     VC1 $= \Psi \supset I$ and VC2 $= \mathsf{sp}(StmtList,\ Bexp \wedge I) \supset I$ should be generated at the entry point and the exit of $StmtList$ respectively.
   - allocation: $\mathsf{sp}(lval = \mathsf{alloc}(Type),\ \Psi) = \Psi'$, Ψ' can be calculated using Ψ according to the allocation rules in the pointer logic.
   - deallocation: $\mathsf{sp}(\mathsf{free}(Exp),\ \Psi) = \Psi'$, Ψ' can be calculated using Ψ according to the deallocation rule in the pointer logic.
   - function call: If the pre- and postconditions of function id are $\{arg\} \wedge \{lval\}_{\mathcal{N}} \wedge Q_{arg}$ and $\Psi' \wedge Q$ respectively, then $\mathsf{sp}(lval = id(Exp),\ \Psi) = (\Psi' \wedge Q)[arg \leftarrow Exp][res \leftarrow lval]$, where Q and Q_{arg} are assertions that have nothing to do with pointers, arg is the formal parameter, and res is the return value (see Figure 1).
     VC $= \Psi \supset (\{arg\} \wedge \{lval\}_{\mathcal{N}} \wedge Q_{arg})[arg \leftarrow Exp]$ should be generated at the entry point of this statement.
   - return: $\mathsf{sp}(\mathsf{return}\ res,\ \Psi) = \Psi$

Figure 2. The Strongest postcondition calculus and the VC generation of pointer-type data

2. One difficulty of strongest postcondition calculus is the need to find a fixpoint for a recursive equation in a loop statement. The solutions to such equations are usually undecidable, and it is also the primary reason why the correctness of a program can not be proved automatically. The loop invariant provided by programmers is used to avoid the difficulty. However, in order to prove the validity of the loop invariant, two VCs must be generated at the entry point and the exit of the loop respectively.

3. The pre- and postconditions of each function have also deeply simplified the computation of sp for function call statement. Briefly speaking, the Ψ before the call statement should imply the precondition of the callee, and the callee's postcondition should be used as the strongest postcondition after the call statement. Certainly, we also need to consider the substitution of actual parameters for formal parameters as well as the substitution of the variable *lval* for the variable *res* (see function call in Figure 2).

4. One remarkable distinction of our pointer logic from Hoare logic is that the pointer logic has no uniform assignment axiom. Instead, different assignment rules are used in different cases in our pointer logic. Since the pointer analysis is precise, it is easy to determine which rule to use in a certain case and it is clear how to compute Ψ' using Ψ in the sp rule for assignment. Please refer to [2] for some details skipped in Figure 2.

5. At the entry point of a function, the pointer logic should check the initial values of the pointer-type formal parameters and local variables. And at the exit of the function, their effectiveness should also be checked to avoid memory leaks. But for the space limit, the VC generation in Figure 2 does not reflect this.

For integer-type data, we adopt a complemented approach of Hoare logic—weakest precondition calculus (wp) [5] to generate assertions or VCs at each program point. And we only introduce its distinctions from the approach for pointer-type data as follows.

1. Since the assertions in a function are calculated backward using the postcondition Q of the function, a VC $P \supset \mathsf{wp}(StmtList, Q)$ is generated at the entry point of the function, where P is the precondition of the func-

tion and *StmtList* is the statement sequence of the function.

2. The side condition in a typing rule should be combined with the assertion which is at the entry point of the corresponding statement. For pointer-type data, such a problem does not exist. Because a pointer-related side condition is consistent with the premise of the corresponding inference rule in the pointer logic, and should have been checked before the rule is chosen.

3. When we compute pointer-related assertions, we need to decide which assignment rule to use according to the assertion before the assignment. However, there is no need to do that about integer-related assertions using wp.

In fact, the computations of assertions and VCs for integer-type data and pointer-type data interact with each other.

1. Both of them should be performed together with type checking, since they both need side conditions in the typing rules. But the backward calculus for integer-related assertions will reduce the efficiency of the performance, because the calculus direction does not consist with the direction of type checking.

2. The pre- and postconditions of functions, loop invariants and the *Bexp*s of while and if statements may contain both integers and pointers. Such an assertion should be divided into two parts, each of which concerns only one kind of data and participates one calculus.

3. The variable p->*data* makes sense only when p is an effective pointer, but in Hoare logic, there is no such a consideration. Therefore, rules in Hoare logic can not be used in the case where pointers are concerned. The pointer logic collects pointer equality information, and this information can help us check whether p->*data* makes sense. Moreover, when using assignment axiom, we can perform alias substitution [2] according to the equality information of pointers.

Using the pointer logic, it is easy to prove the correctness of the VC generation, *i.e.*, if all of the VCs are proved, the Hoare triple $\{P\}\ id(arg)\{Body\}\ \{Q\}$ holds. The VC generator is removed from the TCB, since there is a proof checker at assembly level in our framework. Specifications, VCs and the proofs of VCs at source level will be translated into equivalent assembly-level counterparts. And the translated counterparts will be used in the verification at assembly level.

4. Code and assertion generation

In our framework, specifications and the proof of code satisfying the specifications are carried in the assembly code. The assembly code is divided into basic blocks. Basic block, which is a concept in code optimization and generation, is a sequence of instruction; and in our design, the instruction sequence ends with a control transfer instruction such as `jmp` or `call`. Each basic block B has its precondition P, postcondition Q, and the proof or proof hint of $\{P\}\ B\ \{Q\}$. The proof can be checked by a proof checker. According to the principle that the postcondition of a basic block should imply the precondition of the succeeding basic block in the control flow, Q can be omitted since we can just take the precondition of the succeeding basic block as Q.

In order to make sure that each basic block has pertinent pre- and postconditions, assertions should be generated at proper program points during code generation. In this section, we introduce a scheme for generating intermediate code and the corresponding assertions. For the lack of space, we only consider pointer-related assertions. Using the calculi in section 3, we can get a proper assertion at each program point. Figure 3 shows the scheme for the main syntactic structures. The first parameter of the recursive function Code is a syntactic structure, and the second parameter is the precondition of the structure. The function Code generates intermediate code and assertions at some program points. The generated assertion is at the right side of the code. The assertion which follows "−−" should be inserted before the line of the code, and which follows "++" should be inserted after the line of the code; and the assertion led by "$**$" is the VC to be proved. The function Code calls function code which uses syntactic structure as the only parameter. The function code only generates intermediate code for the syntactic structure. The function Code does not generate assertions between the intermediate code which will obviously be translated into the same basic block, such as the code for assignment statements and Boolean expressions (Boolean expression calculation does not use short-circuit here). The description in Figure 3 actually covers some assertion and VC generations in Figure 2, hence all of these can be done via a one-pass inspection of source programs during compilation.

When considering the generation of integer-related assertions and VCs, it is difficult to do all the work about pointers and integers in one pass, because they are based on the calculi in different directions. The compiler can do type checking, pointer-related assertion generation, VC generation, integer side-condition annotation and intermediate code generation in the first pass and generate integer-related assertions and VCs using the annotation of integer side conditions in the second pass.

1. Statement list

• non-empty: Code(*Stmt StmtList*, Ψ) =

Code(*Stmt*, Ψ)	-- Ψ
Code(*StmtList*, sp(*Stmt*, Ψ))	

• empty: Code(ϵ, Ψ) = ϵ ** *If ϵ forms the StmtList of a function Body, $VC = \Psi \supset \Psi'$ will be generated. Ψ' is the postcondition of the function.*

2. Statement

• assignment: Code(*lval* = *Exp*, Ψ) =

code(*lval* = *Exp*)	++ sp($lval = Exp, \Psi$)

• condition: Code(if *Bexp* {$StmtList_1$} else {$StmtList_2$}, Ψ) =

code(*Bexp*)	++ Ψ
JUMP on false to L1	
Code($StmtList_1$, $Bexp \wedge \Psi$)	
GOTO L2	
L1: Code($StmtList_2$, $\neg Bexp \wedge \Psi$)	
L2:	

• loop: Code(while (*Bexp*) {*StmtList*}, Ψ) =

L1 : code(*Bexp*)	-- *I*, ++ *I*, ** $VC = \Psi \supset I$ (*I is the loop invariant.*)
JUMP on false to L2	
Code(*StmtList*, $Bexp \wedge I$)	
GOTO L1	** $VC = \text{sp}(StmtList, Bexp \wedge I) \supset I$
L2 :	-- $\neg Bexp \wedge I$

• function call: Code(*lval* = *id*(*Exp*), Ψ) =

temp := code(*Exp*)	-- $(\{arg\} \wedge \{lval\}_{\mathcal{N}} \wedge Q_{arg})[arg \leftarrow Exp]$
	** $VC = \Psi \supset (\{arg\} \wedge \{lval\}_{\mathcal{N}} \wedge Q_{arg})[arg \leftarrow Exp]$
param *temp*	++ $(\{arg\} \wedge \{lval\}_{\mathcal{N}} \wedge Q_{arg})[arg \leftarrow temp]$
call *id*	++ $(\Psi' \wedge Q)[arg \leftarrow Exp][res \leftarrow res\_temp]$
lval := *res_temp*	++ $(\Psi' \wedge Q)[arg \leftarrow Exp][res\_temp \leftarrow lval]$

Figure 3. Intermediate code generation and generation of pointer-related assertions

When generating assembly code, the pre- and postconditions of basic blocks should be adjusted as follows:

1. At source or intermediate level, variables are represented by names in assertions; but at assembly level, they are represented by memory addresses or registers. Also, an assertion at assembly level is parameterized by a machine state. So, assertions also need to be translated during code generation. It is lucky that this kind of translation is straightforward.

2. Registers are used to store temporary values in the assembly code, so the contents of some registers may equal the values of some variables at the exit of one basic block. Usually, code generation algorithm can collect such information. And this information makes it easy to adjust Ψs at the entry point and the exit of one basic block.

   The assertion generation in Figure 3 also faces this problem, because temporary variables may be introduced into the intermediate code for expressions.

3. At the entry point and the exit of each basic block, there are some relatively steady assertions such as "the return address saved in the current stack frame will not be overwritten during the execution of the basic block". All of these assertions depend on the target machine. Since they are almost the same for each basic block, there is no difficulty in generating them.

5. Proof generation for basic blocks

In the generated assembly code, each basic block *B* has a proof or proof hint for **{***P***}** *B* **{***Q***}**. The proof or proof hint is generated by the compiler. Besides the proofs or hints for basic blocks, the assembly code also carries assembly-level VCs and their proofs. These VCs and proofs can be achieved by translating the source-level VCs in Figure 2 and their proofs respectively. And they are usually used when a basic block's postcondition should be proved to imply its successor's precondition.

At assembly level, we adopt an approach that applies the method of Hoare logic directly to the operational semantics of Intel x86 ISA. This approach has the following characters. First, an assertion is parameterized by a machine state;

second, the primary inference rules are based on the operational semantics of the instructions, that is, the machine state before the execution of an instruction is used to calculate the state after it [12]. While in Hoare logic, the calculation of the precondition from the postcondition is directly based on syntactic substitution. Generally, machine state denotes a map from registers, stack locations and heap locations to values. The change of machine state is based on the operational semantics of the instructions. The formal description of our proof-carrying assembly code has referred to CAP and SCAP [19, 6], which are common frameworks supporting Hoare style reasoning for assembly code certification.

The instruction execution may change pointer information in Π, $\mathcal{N}$ and D, and the change can be deduced according to the pointer logic at assembly level [11]. Besides function call, the instructions which may change the pointer information include `movl`, `pushl` *etc.* (we call them assignment instructions in general). The rules for these instructions in the pointer logic at assembly level are almost the same as the corresponding assignment rules at source level, except that variables are replaced by memory addresses, registers, and register indirect addresses *etc.* Since there is no complex access path in assembly programs, aliasing calculation at assembly level is much simpler than that at source level.

To generate the proof of a basic block satisfying its pre- and postconditions, it is important to generate assertions between the instructions in the basic block. These assertions can be generated from the pre- and postconditions of the basic block. According to *P*, *Q* and the corresponding machine state, the verification framework at assembly level checks whether $\{P\}\ \iota\ \{Q\}$ holds for each instruction ι in the basic block. Pointer-related assertions are generated and inserted into the basic block in a forward manner, with integer-related assertions in a backward manner. Pointer-related assertions at assembly level are rather difficult to understand, so we take a basic block with integer-related assertions as an example to explain how to generate and insert assertions between instructions.

The source program in Figure 4 is a function to multiply integer `m` by `n`. The assertions in notation `{ }` are provided by programmers. The precondition of the function is represented as the assertion in the first line; in line 12 is the postcondition. The loop invariant for while is in line 6. The symbol "`result`" represents the return value of the function. The assembly code in the same figure corresponds to the while loop (line 6-9), where the addresses of `n`, `m`, `x` and `y` in the stack are represented by `12(%ebp)`, `16(%ebp)`, `-16(%ebp)` and `-20(%ebp)` respectively, and `ebp` is the base pointer of current stack frame. Then, we take the basic block `ltrue` (the basic block which are numbered in Figure 4) as an example to explain the generation of the proof.

The precondition *p* of the basic block is

$$\begin{aligned}\lambda\mathbb{S}.\mathbb{S}(\mathbb{S}(ebp)-16) = \mathbb{S}(\mathbb{S}(ebp)+16) \times \mathbb{S}(\mathbb{S}(ebp)-20) \wedge \\ \mathbb{S}(\mathbb{S}(ebp)-20) \leq \mathbb{S}(\mathbb{S}(ebp)+12) \wedge \\ \mathbb{S}(\mathbb{S}(ebp)-20) < \mathbb{S}(\mathbb{S}(ebp)+12).\end{aligned}$$

(Note that the corresponding assertion at source level is `{x == m * y ∧ y <= n ∧ y < n}`.)

The postcondition q (q is also the precondition of the succeeding basic block `lloop`) is

$$\begin{aligned}\lambda\mathbb{S}.\mathbb{S}(\mathbb{S}(ebp)-16) = \mathbb{S}(\mathbb{S}(ebp)+16) \times \mathbb{S}(\mathbb{S}(ebp)-20) \wedge \\ \mathbb{S}(\mathbb{S}(ebp)-20) \leq \mathbb{S}(\mathbb{S}(ebp)+12).\end{aligned}$$

(Similarly, the corresponding assertion at source level is `{x == m * y ∧ y <= n}`, *i.e.*, the loop invariant.)

In these assertions, $\mathbb{S}$ denotes the current machine state. If the postcondition of an instruction ι is *Q*, its precondition *P* can be achieved through the formula: (note that *P* and *Q* are parameterized by a machine state $\mathbb{S}$)

$$P = \mathsf{gp}(\iota, Q) = \lambda\mathbb{S}.Q\,(\mathsf{upd}(\mathbb{S}, \iota)),$$

where the function $\mathsf{upd}(\mathbb{S}, \iota)$ represents the operational semantics of the instruction ι, and the return value of $\mathsf{upd}(\mathbb{S}, \iota)$ is a machine state after the execution of the instruction ι in state $\mathbb{S}$.

Figure 5 shows the operational semantics and precondition generation formulas of some instructions. $\mathbb{S}[a \mapsto b]$ is a machine state, and it means:

$$\mathbb{S}[a \mapsto b](c) = \begin{cases} b & \text{if } c = a, \\ \mathbb{S}(c) & \text{if } c \neq a. \end{cases}$$

For example, suppose the postcondition for instruction "`movl  r1, r2`" is

$$\lambda\mathbb{S}.\mathbb{S}(\mathtt{r2}) = 3,$$

according to $\mathsf{gp}(($`movl  r1, r2`$), Q)$ in Figure 5, the precondition is

$$\lambda\mathbb{S}.((\lambda\mathbb{S}'.\mathbb{S}'(\mathtt{r2}) = 3)\ \mathbb{S}[\mathtt{r2} \mapsto \mathbb{S}(\mathtt{r1})]),$$

that is,

$$\lambda\mathbb{S}.\mathbb{S}[\mathtt{r2} \mapsto \mathbb{S}(\mathtt{r1})](\mathtt{r2}) = 3.$$

According to the definition of $\mathbb{S}[a \mapsto b]$, the precondition can be reduced to

$$\lambda\mathbb{S}.\mathbb{S}(\mathtt{r1}) = 3.$$

Using $\mathsf{gp}(\iota, Q)$, we can get the precondition of each instruction backward from the postcondition of the basic

```
1    {n >= 0}
2    int mult (m, n){
3      int x, y;
4      x = 0;
5      y = 0;
6      while (y < n) { {x == m * y ∧ y <= n}
7        x = x + m;
8        y = y + 1;
9      }
10     return x;
11   }
12   {result == m * n}
```

```
lloop:
          ...
          cmpl   %esi, %eax
          jg     ltrue
lfalse:
          movl   -16(%ebp), %eax
          jmp    lend
ltrue:
1         movl   16(%ebp), %eax
2         addl   -16(%ebp), %eax
3         movl   %eax, -16(%ebp)
4         movl   $1, %eax
5         addl   -20(%ebp), %eax
6         movl   %eax, -20(%ebp)
7         jmp    lloop
lend:
          ...
```

Figure 4. The source code and corresponding assembly code of mult (m, n)

Instruction: ι	**Operational semantics:** $\mathsf{upd}(\mathbb{S}, \iota)$	**Precondition:** $\mathsf{gp}(\iota, Q)$
`movl r1, r2`	$\mathbb{S}[\mathtt{r2} \mapsto \mathbb{S}(\mathtt{r1})]$	$\lambda\mathbb{S}.Q\ (\mathbb{S}[\mathtt{r2} \mapsto \mathbb{S}(\mathtt{r1})])$
`movl r1, z(r2)`	$\mathbb{S}[\mathtt{z} + \mathbb{S}(\mathtt{r2}) \mapsto \mathbb{S}(\mathtt{r1})]$, if $(\mathtt{z} + \mathbb{S}(\mathtt{r2})) \in \mathsf{Dom}(\mathbb{S})$	$\lambda\mathbb{S}.Q\ (\mathbb{S}[\mathtt{z} + \mathbb{S}(\mathtt{r2}) \mapsto \mathbb{S}(\mathtt{r1})])$
`movl z, r`	$\mathbb{S}[\mathtt{r} \mapsto \mathtt{z}]$,	$\lambda\mathbb{S}.Q\ (\mathbb{S}[\mathtt{r} \mapsto \mathtt{z}])$
`jmp l`	$\mathbb{S}$	Q
`addl z(r1), r2`	$\mathbb{S}[\mathtt{r2} \mapsto \mathbb{S}(\mathtt{z} + \mathbb{S}(\mathtt{r1})) + \mathbb{S}(\mathtt{r2})]$, if $(\mathtt{z} + \mathbb{S}(\mathtt{r1})) \in \mathsf{Dom}(\mathbb{S})$	$\lambda\mathbb{S}.Q\ (\mathbb{S}[\mathtt{r2} \mapsto \mathbb{S}(\mathtt{z} + \mathbb{S}(\mathtt{r1})) + \mathbb{S}(\mathtt{r2})])$
`movl z(r1), r2`	$\mathbb{S}[\mathtt{r2} \mapsto \mathbb{S}(\mathtt{z} + \mathbb{S}(\mathtt{r1}))]$, if $(\mathtt{z} + \mathbb{S}(\mathtt{r1})) \in \mathsf{Dom}(\mathbb{S})$	$\lambda\mathbb{S}.Q\ (\mathbb{S}[\mathtt{r2} \mapsto \mathbb{S}(\mathtt{z} + \mathbb{S}(\mathtt{r1}))])$

Figure 5. Operational semantics and assertion generation of some instructions

block. For example, from postcondition q, we can get the precondition of "`jmp lloop`" (the instruction labeled with 7):

$$\lambda\mathbb{S}.\mathbb{S}(\mathbb{S}(ebp) - 16) = \mathbb{S}(\mathbb{S}(ebp) + 16) \times \mathbb{S}(\mathbb{S}(ebp) - 20) \wedge \mathbb{S}(\mathbb{S}(ebp) - 20) \leq \mathbb{S}(\mathbb{S}(ebp) + 12).$$

The preconditions of the rest instructions can be generated in a similar way. The precondition of the first instruction is:

$$\lambda\mathbb{S}.\mathbb{S}(\mathbb{S}(ebp) - 16) + \mathbb{S}(\mathbb{S}(ebp) + 16) = \mathbb{S}(\mathbb{S}(ebp) + 16) \times (\mathbb{S}(\mathbb{S}(ebp) - 20) + 1) \wedge \mathbb{S}(\mathbb{S}(ebp) - 20) + 1 \leq \mathbb{S}(\mathbb{S}(ebp) + 12).$$

And the VC at the entry point of the basic block is:

$$\begin{aligned}\lambda\mathbb{S}.\mathbb{S}(\mathbb{S}(ebp) - 16) = \mathbb{S}(\mathbb{S}(ebp) + 16) \times \mathbb{S}(\mathbb{S}(ebp) - 20) \wedge\\ \mathbb{S}(\mathbb{S}(ebp) - 20) \leq \mathbb{S}(\mathbb{S}(ebp) + 12) \wedge\\ \mathbb{S}(\mathbb{S}(ebp) - 20) < \mathbb{S}(\mathbb{S}(ebp) + 12) \supset\\ (\lambda\mathbb{S}.\mathbb{S}(\mathbb{S}(ebp) - 16) + \mathbb{S}(\mathbb{S}(ebp) + 16)\\ = \mathbb{S}(\mathbb{S}(ebp) + 16) \times (\mathbb{S}(\mathbb{S}(ebp) - 20) + 1) \wedge\\ \mathbb{S}(\mathbb{S}(ebp) - 20) + 1 \leq \mathbb{S}(\mathbb{S}(ebp) + 12)).\end{aligned}$$

That is, the precondition of the basic block should imply the precondition of the first instruction in the basic block. And this VC is easy to prove obviously.

When we get all the assertions between the instructions, there is no difficulty to generate the proof of $\{P\}\ \iota\ \{Q\}$ for each instruction ι in a basic block. And it is also easy to generate a proof for the basic block if we have the proofs for all the instructions in the basic block and the proof for the VC of the basic block.

6. Related work

Verifying compiler [8] is a research topic proposed in recent years. It uses mathematical and logical reasoning to check the correctness of the program that it compiles before the program executes. The criterion of correctness is specified by types, assertions, and other redundant annotations associated with the program. Now it can be classified into two research directions on this topic. One is to prove the correctness of the compiler formally, that is, the compiler carries a formal proof which confirms the correctness of the generated objective code. Such a compiler is called a certified compiler. The other is to prove the correctness of the compiled code formally, *i.e.*, the code carries a formal proof to guarantee its correctness, safety or some other properties and the proof can be checked by a code consumer independently. This kind of compiler is called a certifying compiler. Obviously, this paper presents the design of parts of a certifying compiler.

In the research area of certified compiler, Moore was one of the first to mechanically verify semantic preservation for a compiler [14], although for a custom language and a custom processor that were not commonly used. After that, more compilers were verified, including a compiler for a subset of Common Lisp, a byte-code compiler for a subset of Java, and a compiler for a tiny subset of C. The most recent typical work is the certification of a lightly-optimizing back end that generates PowerPC assembly code from a simple imperative intermediate language called Cminor [10] by Leroy. A front end translating a subset of C to Cminor is under development and certification. One of the novel features of his work is to emphasize the certification of a complete compilation chain instead of parts of a compiler. Another novelty is that most of the compiler is written directly in the Coq specification language, in a purely functional style.

Generally, it is much easier to prove the correctness of a calculation result than to prove the correctness of the calculation itself, so that certifying compiler has more possibility to come into use than certified compiler. Necula first proposed the concept of Proof-Carrying Code, and implemented a certifying compiler called Touchstone [17]. This compiler was composed of a traditional optimizing compiler for a type-safe subset of C and a certifier that automatically produced a proof of type safety and memory safety for each assembly program. A proof checker could be used to check the generated proofs automatically. Since the source programs compiled by Touchstone were written in a very small safe subset of C, their type safety and memory safety were easy to be checked. The main limitations of the source language in Touchstone were in the aspects of pointer types and memory deallocation, which made the language only used in writing programs with simple data structures. Later, Colby *et al.* implemented a certifying compiler called Special J [3] for a large subset of Java. It compiled Java byte code into target code for Intel IA32 architecture. The major advance of Special J over Touchstone was the scope of the source language compiled, which in turn necessitated the handling of non-trivial run-time mechanisms such as object representation, dynamic method dispatch and exception handling. Moreover, Special J was freely able to apply many standard local and global optimizations.

Our design has the following significant differences from Touchstone and Special J:

1. PointerC has more pointer types and operations, and also provides dynamic storage allocation and deallocation. These features make it suitable for writing system-level programs. Both Touchstone and Special J have no dynamic storage deallocation.

2. We use some new techniques to handle the features of the language equipped with both a type system and a logic system. For example, our VC generator can perform both forward and backward VC generations, and the theorem prover embedded in the compiler and the Coq proof assistant can be used to prove VCs for different types of data respectively.

3. To certify the proof-carrying assembly code, we adopt an approach that applies the method of Hoare logic directly to the operational semantics of Intel x86 ISA. We use machine state as the parameter of assertions, so as to describe program properties which may change with the machine state. These properties are usually difficult to be expressed in common type systems. Touchstone and Special J mainly focus on type properties of programs, and rarely concern complicated program properties.

4. Due to the simplicity of the source language, loop invariants which only concern types can be generated automatically in Touchstone and Special J. In our certifying compiler, loop invariants may contain more information than types, and it should be provided by programmers explicitly.

7. Conclusion

This paper presents the design and implementation of a certifying compiler for PointerC (a programming language equipped with both a type system and a logic system). Although the implemented certifying compiler prototype only enforces the primary safety policies such as type safety and memory safety, the whole framework can fit in the situation where user-defined safety policies are allowed. This paper only concentrates on the main work of the certifying compiler and skips the translation of assertions, VCs and proofs

of VCs. The design of the theorem prover, which is embedded in the compiler to prove pointer-related VCs, is also skipped in this paper.

We are improving PointerC and its certifying compiler: relaxing the restrictions on pointer arithmetic operations and allowing calloc which is often used in programs. We are also studying an embedded theorem prover for integer-related VCs, and exploring the possibilities to replace the current proof-assistant tool. The influences of proof-carrying compilation on code optimizations are also under consideration.

8. Acknowledgements

We thank Professor Zhong Shao in Yale University and anonymous referees for suggestions and comments on an earlier version of this paper. This research is based on work supported in part by grants from Intel China Research Center and National Natural Science Foundation of China under Grant No.60673126. Any opinions, findings and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

[1] A. W. Appel. Foundational proof-carrying code. In *LICS '01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–258, Washington, DC, USA, 2001. IEEE Computer Society.

[2] Y. Chen, B. Hua, L. Ge, and W. Zhifang. A pointer logic for safety verification of pointer programs. http://ssg.ustcsz.edu.cn/lss/papers/index.html.

[3] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 95–107, New York, NY, USA, 2000. ACM Press.

[4] Coq Development Team. The Coq proof assistant reference manual. Coq release v8.0, Oct. 2005.

[5] E. W. Dijkstra. *A discipline of programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[6] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular verification of assembly code with stack-based control abstractions. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 401–414, New York, NY, USA, 2006. ACM Press.

[7] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. In *LICS'02: Proceedings of the Seventeenth Annual IEEE Symposium on Logic In Computer Science*, pages 89–100, Copenhagen, Denmark, July 2002. IEEE.

[8] T. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.

[9] B. Hua and L. Ge. The definition of pointerc programming language. http://ssg.ustcsz.edu.cn/lss/doc/index.html.

[10] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 42–54, New York, NY, USA, 2006. ACM Press.

[11] Z. Li. Coq implementation of the soundness proof of fcap (description). http://ssg.ustcsz.edu.cn/lss/software/index.html.

[12] Z. Li. A framework of function-based certified assembly programming. http://ssg.ustcsz.edu.cn/lss/doc/index.html.

[13] Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *ICFP'03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 213–225, New York, NY, USA, 2003. ACM Press.

[14] J. S. Moore. *Piton: a mechanically verified assembly-language*. Kluwer Academic Publishers, Norwell, MA, 1996.

[15] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system f to typed assembly language. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 85–97, New York, NY, USA, 1998. ACM Press.

[16] G. C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM Press.

[17] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 333–344, New York, NY, USA, 1998. ACM Press.

[18] H. Xi. Applied type system (extended abstract). In *post-workshop Proceedings of TYPES 2003*, pages 394–408. Springer-Verlag LNCS 3085, 2004.

[19] D. Yu, N. A. Hamid, and Z. Shao. Building certified libraries for pcc: dynamic storage allocation. *Sci. Comput. Program.*, 50(1-3):101–127, 2004.