A pointer logic and certifying compiler [*]

CHEN Yiyun, GE Lin[**], HUA Baojian, LI Zhaopeng,
LIU Cheng, and WANG Zhifang

Department of Computer Science and Technology,
University of Science and Technology of China,
Hefei 230027, China

Abstract. Proof-Carrying Code brings two big challenges to the research field of programming languages. One is to seek more expressive logics or type systems to specify or reason about the properties of low-level or high-level programs. The other is to study the technology of certifying compilation in which the compiler generates proofs for programs with annotations. This paper presents our progress in the above two aspects. A pointer logic was designed for PointerC (a C-like programming language) in our research. As an extension of Hoare logic, our pointer logic expresses the change of pointer information for each statement in its inference rules to support program verification. Meanwhile, based on the ideas from CAP (Certified Assembly Programming) and SCAP (Stack-based Certified Assembly Programming), a reasoning framework was built to verify the properties of object code in a Hoare style. And a certifying compiler prototype for PointerC was implemented based on this framework.
The main contribution of this paper is the design of the pointer logic and the implementation of the certifying compiler prototype. In our certifying compiler, the source language contains rich pointer types and operations and also supports dynamic storage allocation and deallocation.

Categories and Subject Descriptors. F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs-Specification Techniques; D.2.4 [*Software Engineering*]: Software/Program Verification—safety proofs, formal methods;

Keywords. software safety, Hoare logic, pointer logic, Proof-Carrying Code, certifying compiler

1 Introduction

Proof-Carrying Code (PCC) [1] brings two big challenges to the research field of programming languages. One is to explore more expressive logics or type systems, so that the properties of low-level or high-level programs will be easily

[*] Received February 12, 2007; accepted ... .
[**] Email: gelin@mail.ustc.edu.cn

specified or reasoned about. The other is the research on certifying compilation, which explores how the compiler generates proofs for the compiled programs.

For the first challenge, the TAL (Typed Assembly Language) project [2] and the theory of type refinements [3] are two typical projects in type-based approaches, while PCC [1] and FPCC (Foundational Proof-Carrying Code) [4] are typical projects on logic-based techniques. Type-based and logic-based techniques are complementary to each other, and in recent years, some researchers have tried to combine those techniques. Shao *et al.* adopted a syntactic approach to FPCC, and proposed a simple and flexible framework to support Hoare-style reasoning for assembly code certification in their projects CAP (Certified Assembly Programming) [5] and SCAP (Stack-based CAP) [6] *etc.* The ATS (Applied Type System) project [7] proposed by Xi *et al.* extends the type system with a notion of program states, so that invariants on states could be captured in stateful programming. By encoding Hoare logic in its type system, ATS can support Hoare-logic-like reasoning via the type system.

For the second challenge, Necula implemented a certifying compiler [8] called Touchstone. It contains a traditional compiler for a small but type-safe subset of C and a certifier that automatically produces a proof of type safety for each assembly program produced by the compiler. Later, Colby *et al.* implemented Special J [9], a certifying compiler for a large subset of Java. It compiles Java byte code into target code for Intel's IA32 architecture.

We have adopted PCC technology and certifying compilation to a programming language with dynamic storage allocation and deallocation—PointerC[1] (a safe C-like language defined in our work). And we have also implemented a certifying compiler prototype for PointerC. Pointer operations in PointerC are restricted: pointer variables can only be used in assignment, equality comparison, dereference or as the parameters of functions; the address-of operator (&) and pointer arithmetic are forbidden. We adopt an approach combining both type-based and logic-based techniques to reason about program properties. To make the type system simple 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. In order 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. Such 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.

The pointer logic is the main contribution of this paper. As extensions of Hoare logic, separation logic [10] and our pointer logic are both to reason about properties of pointer programs for shared mutable data structures. But there is a great difference between them. Our pointer logic concerns pointer aliasing. In the pointer logic, different access paths are assumed to be bound to different storage locations, unless it can be proved that they are bound to the same location

[1] Hua B J and Ge L. The definition of PointerC programming language. http://ssg.ustcsz.edu.cn/lss/doc/index.html. (in Chinese)

(those bound to the same location are aliases). Therefore, equality information of effective pointers is needed to deduce the access paths that are bound to the same location. And this is also the reason why the pointer logic need not introduce new connectives. Separation logic can only handle programming languages in which all access paths are simple, so it only concerns where a pointer points to. In separation logic, pointers are assumed to potentially point to the same location, unless they are explicitly expressed to point to different locations. Therefore, it needs to introduce new logical connectives such as separating conjunction "*" *etc.*

While designing inference rules for the pointer logic, we extends Hoare logic in the viewpoint of pointer analysis. The inference rules are designed to be suitable for the collection of pointer information in a forward way. Such information can be used to check the legality and other properties of a program. Separation logic extends Hoare logic in the viewpoint of program verification, and its inference rules are designed to fit backward reasoning.

In addition, in source programs, it is common to use expressions with the form `id1->id2->id3->...->idn`, such as `s->next->next`, but this kind of expressions could not be used directly in the assertions of separation logic because they involve several separated portions of addressable storage. This makes it difficult for separation logic assertions to be expressed directly using program expressions. In assertions of separation logic, accessibility is explicitly represented by "$\mapsto$" . This makes the inference rules simple and clear, but when several access paths are bound to the same location, which is common in source programs, the explicit representation will increase the steps of deduction. For example, considering the program fragment:

```
p = malloc(...); *p = a;
{*p == a}
q = p; *q = b;
{*p == b}
```

where expressions in notation **{ }** are assertions. When using separation logic, the deduction from assertion `*p == a` to assertion `*p == b` has more steps than that using the pointer logic.

Another contribution of this paper is the implementation of a certifying compiler prototype. Our certifying compiler supports a source language (PointerC) equipped with both a type system and a logic system. And compared with the source languages compiled in Touchstone and Special J, PointerC has more pointer types and operations, and provides dynamic storage allocation and deallocation as well. These features make PointerC suitable for writing system-level programs.

In this paper, we present the pointer logic designed for PointerC and the certifying compiler prototype that has been implemented. The rest of the paper is organized as follows. In section 2, we explain the design of the pointer logic; section 3 describes the assembly-level pointer logic system added in an assembly code certification framework; section 4 discusses the design of the certifying compiler prototype; section 5 gives an example; section 7 compares our work with related work and section 8 concludes.

2 The pointer logic

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 enough when the legality of a phrase depends not only on the context but also on some expression values in the phrase. One solution is to use dependent types [7]. Another solution is not to deem the constraints on values as parts of the type system, and the constraints don't appear in the typing rules. The former makes the type system complicated and the latter makes it an unsound type system.

We try to investigate a trade-off between these two solutions in the design of PointerC. To make the type system simple 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, in the following two typing rules, side conditions contain the constraints on subscript expressions and pointers.

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

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

These side conditions must be checked. And to check these side conditions statically, we have designed a pointer logic for PointerC. The design of the pointer logic is inspired by the following facts and speculations:

1. At a program point, if we know whether a pointer is an effective pointer (a pointer which points to an object), a null pointer or a dangling pointer and the equality relation between all the effective pointers, we can deduce whether a pointer operation is safe or not. And we can also capture the change of the pointer information after this operation.
2. A logic system may not able to prove the safety of all programs. But it is still feasible if it can prove the safety of most programs.
3. The pointer information collected by Hoare-style reasoning can be used to reason about program properties. It amounts to combining program analysis with program verification and this combination has a brilliant future.

To present the pointer logic, we make clear some pointer-related terms and notations first, and then explain some elementary operations used in the pointer logic. After that, we introduce the inference rules in our pointer logic.

2.1 Conventions, terminologies and notations

Dynamic variables can only be accessed through declared pointer variables. For example, `s->data`, `s->next->pre`, `*s` and `*s[5]` *etc.* Such expressions are called *symbolic access paths* of variables, *access paths* for short. Note that `s->next` is not an access path if `s` is a null or a dangling pointer. For convenience, duplicated parts in an access path are abbreviated in the pointer logic. For example, access path `s->next->next->`...`->next` ("`->next`" is repeated i times) is abbreviated to `s(->next)`$^{\mathtt{i}}$, and if i=0, `s(->next)`$^{\mathtt{i}}$ just represents `s`. Pointer variables (including dynamic pointer variables) are called pointers for short, and they are classified into *effective pointers*, null pointers and dangling pointers, and the latter two are also called *ineffective pointers*. In the rest of this paper, two different concepts—*equality* and *aliasing* of pointers—are often mentioned. When two pointers are *aliases*, it means that their l-values are equal; and when they are *equal*, it means that their r-values are equal. Because of the limitations on pointer operations and the call-by-value calling convention in PointerC, a declared variable can not be aliased by other variables (in this paper, aliasing of array elements is not considered). Only if two pointers are equal, the access paths, which are formed by appending a common suffix to the two pointers, are aliases. For example, in Fig. 1, `u` and `v` are equal pointers, but not aliases. `u->next` and `v->next` are aliases. Obviously, if we have the information of pointer equality, we can deduce whether two access paths are aliases. In the rest of the paper, we use metavariables p, q and r to represent access paths.

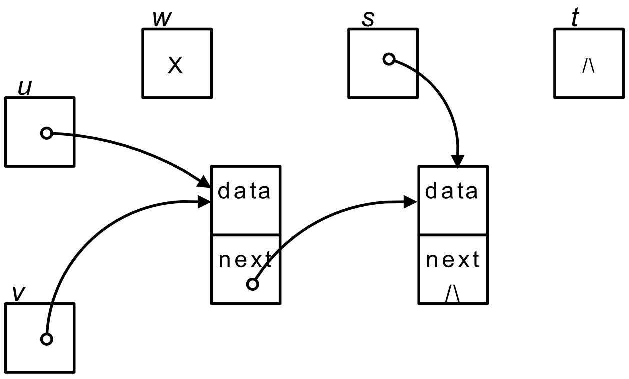

Fig. 1. Pointers in Π, $\mathcal{N}$ and $\mathcal{D}$

In the pointer logic, we use "$\mathcal{N}$" to denote null pointer set, "$\mathcal{D}$" for dangling pointer set and "Π" for effective pointer set at any program point. Π is divided into *equivalent sets* according to the equality information of effective pointers. Pointers in one equivalent set are equal but not aliases. At any point of a program, all pointers (or their aliases) can be found in Π, $\mathcal{N}$ or $\mathcal{D}$. If a pointer has several aliases at this point, only one of them is in Π, $\mathcal{N}$ or $\mathcal{D}$. For example, in Fig. 1, Π `=` {{`u, v`}, {`s, u->next`}}, $\mathcal{N}$ `=` {`t, s->next`}, $\mathcal{D}$ `=` {`w`}. (In this figure, the cross in `w` represents meaningless location, and "/\" represents null.) Note that the number of access paths in Π is just the number of arrows in the figure, and the arrows which point to the same object correspond to the pointers in the same equivalent set. Although Π, $\mathcal{N}$ and $\mathcal{D}$ are sets, they are essentially logical expressions connected by conjunction "$\wedge$", and can appear in the precondition or postcondition of a Hoare triple. "Ψ" is a short notation for $\Pi \wedge \mathcal{N} \wedge \mathcal{D}$. For example, in Fig. 1, the equivalent set $\{$`u, v`$\}$ is actually a logical expression, and it means that `u` and `v` are equal but unequal to any of other pointers; all the pointer sets can be expressed as $\{$`u, v`$\}\wedge\{$`s,u->next`$\}\wedge\{$`t, s->next`$\}_{\mathcal{N}}\wedge\{$`w`$\}_{\mathcal{D}}$, where the sets without subscript represent equivalent pointer sets, and the sets with subscript $\mathcal{N}$ and $\mathcal{D}$ represent null pointer set and dangling pointer set respectively. The null pointer set $\{$`t, s->next`$\}_{\mathcal{N}}$ can also be written as $\{$`t`$\}_{\mathcal{N}}\wedge\{$`s->next`$\}_{\mathcal{N}}$, and the same for the dangling pointer set $\mathcal{D}$. If, at a program point, Π is an empty set, the assertion at this point contains no equivalent set; if $\{\ldots\}_{\mathcal{N}}$ or $\{\ldots\}_{\mathcal{D}}$ does not appear in the assertion, it means that null pointer set or dangling pointer set is empty.

2.2 Elementary operations

Access paths are a special kind of strings that satisfies certain syntactical requirements, and in this paper, all strings represent the strings or substrings that form access paths. If an access path p is a prefix of q (but they are not the same), the value of the predicate $\mathsf{prefix}(p, q)$ is `true`, otherwise is `false`. The symbol "$||$" represents the concatenation operation of two strings or a string set and a single string. If the operands are a string set $\mathcal{S}$ and a single string s, it makes every element in the set $\mathcal{S}$ concatenated by the string s:

$$\mathcal{S}||s \triangleq \mathcal{S}' \ \ \textbf{where} \ \ (s'||s) \in \mathcal{S}' \ \ \textbf{iff} \ \ s' \in \mathcal{S}.$$

If the values of two access paths $s_1||s_2$ and s_1 are equal (note that neither of s_1 and s_2 are empty strings), then the access path s_2 is called a *cycle* in $s_1||s_2||s_3$ (s_3 is also a non-empty string). The symbol "$\equiv$" is used to represent that two strings are syntactically identical.

Next, we define some functions operated on access paths. The functions have Ψ or Π as their parameters. But such parameters are all omitted for simplicity.

1) Function $\mathsf{closure}(p)$ calculates alias set for the access path p, and the aliases in this set must have the simplest form. This set is called the closure of p, and it contains and only contains all acyclic aliases of p, including p itself.

```
$\mathsf{closure}(p) \triangleq$
  $\textbf{if}\ \mathsf{length}(p) = 1\ \textbf{then}\ \{p\}$
  $\textbf{else let}\ (s_1||\ldots||s_{n-1}||s_n) \equiv p$
        $\textbf{in}\ \mathsf{compression}(\mathsf{expansion}(\mathsf{closure}(s_1||\ldots||s_{n-1}))||s_n),$
```

where function $\mathsf{length}(p)$ calculates the length of the access path p. The length represents the number of meaningful units in p, rather than the number of characters in p. For example, the length of the access path `t->next->data` is 3.

Function $\mathsf{expansion}(\mathcal{S})$ expands the alias set $\mathcal{S}$ with the access paths whose values are equal to the elements in $\mathcal{S}$, and its formal definition is:

```
$\mathsf{expansion}(\mathcal{S}) \triangleq$
  if $\exists \mathcal{S}' : (\mathit{\Pi} \cup \{\mathcal{N}\} \cup \{\mathcal{D}\}).(\mathcal{S} \cap \mathcal{S}' \,!\!= \emptyset)$
  then let $\{p_1,\ \ldots,\ p_n\} = \ \mathcal{S}' - \mathcal{S}$
           where $\mathcal{S}' \in (\mathit{\Pi} \cup \{\mathcal{N}\} \cup \{\mathcal{D}\}) \wedge \mathcal{S} \cap \mathcal{S}' \,!\!= \emptyset$
       in $\mathcal{S} \cup \mathsf{closure}(p_1) \cup \ldots \cup \mathsf{closure}(p_n)$
  else $\emptyset$.
```

Function $\mathsf{compression}(\mathcal{S})$ deletes all the cyclic access paths from the alias set $\mathcal{S}$, and its formal definition is:

```
$\mathsf{compression}(\mathcal{S}) \triangleq \mathcal{S} - \mathcal{S}'$
  where $(\mathcal{S}' \subset \mathcal{S}) \wedge ((s_1||s_2||s_3) \in \mathcal{S}'$
        iff $(s_1 \,!\!= \epsilon) \wedge (s_2 \,!\!= \epsilon) \wedge (s_3 \,!\!= \epsilon) \wedge ((s_1||s_3) \in \mathcal{S}) \wedge (s_1||s_2 = s_1))$
```

For brevity, the above function closure is a *definition*, not a computing *algorithm*. For example, it does not consider the termination of the calculation concerning cyclic data structures, such as doubly-linked cyclic list. But, in the implementation of closure, it is easy to solve the problem of termination. And given the closure function, it is also simple to delete cycles from the access paths. So for convenience, it is assumed that all access paths in programs have the simplest form.

2) Function $\mathsf{alias}(p,\ q)$ gets an alias of p from the closure of access path p. The alias it gets must not use any alias of effective pointer q as a prefix. If there is no such an alias, it returns p.

```
$\mathsf{alias}(p, q) \triangleq$
  let $\mathcal{S} = \{p' : \mathsf{closure}(p) \mid \forall q' : \mathsf{closure}(q).\neg\mathsf{prefix}(q', p')\}$
  in if $\mathcal{S} == \emptyset$ then $p$ else $p'$ where $p' \in \mathcal{S}$
```

3) Function $\mathsf{equals}(p)$ gets the equivalent set of p. If one of the aliases of p appears in an equivalent set, the function returns the set, or else it returns empty set.

```
$\mathsf{equals}(p) \triangleq$
  if $\exists \mathcal{S} : \mathit{\Pi}.(\mathcal{S} \cap \mathsf{closure}(p) \,!\!= \emptyset)$
  then $\mathcal{S}$ where $\mathcal{S} \in \mathit{\Pi} \wedge \mathcal{S} \cap \mathsf{closure}(p) \,!\!= \emptyset$ else $\emptyset$
```

Next, we introduce the operations and predicates directly used in inference rules. These operations show how to get the $\mathit{\Psi}$ in a postcondition based on the $\mathit{\Psi}$ in the corresponding precondition.

4) Suppose $\mathcal{S}$ is an equivalent set in $\mathit{\Pi}$, and p is an effective pointer. For each pointer q in $\mathcal{S}$ which uses one of p's aliases as its prefix, $\mathcal{S}/p$ finds an alias of q using $\mathsf{alias}(q,p)$, substitutes the alias for q in $\mathcal{S}$, and then deletes p's aliases and any other pointers which use p's aliases as their prefixes from $\mathcal{S}$.

```
$\mathcal{S}/p \triangleq$
  let $\mathcal{S}' = \{q : \mathcal{S} \mid \forall p' : \mathsf{closure}(p).\neg\mathsf{prefix}(p', q)\} \cup$
      $\{q' \mid \exists q : \mathcal{S}.\exists p' : \mathsf{closure}(p).(\mathsf{prefix}(p', q) \wedge q' \equiv \mathsf{alias}(q, p))\}$
  in $\{q : \mathcal{S}' \mid \neg(q \in \mathsf{closure}(p)) \wedge \forall p' : \mathsf{closure}(p).\neg\mathsf{prefix}(p', q)\}$
```

We use Π/p to denote performing $\mathcal{S}/p$ for each $\mathcal{S}$ in Π.

When an effective pointer q is assigned a value which is not equal to q, Π/q should be performed. For example, in Fig. 2 where $\Pi = \{\{\texttt{u, v->pre}\}, \{\texttt{v, u->next}\}\}$, $\Pi/\texttt{v} = \{\{\texttt{u, u->next->pre}\}, \{\texttt{u->next}\}\}$. ($\{\texttt{u->next}\}$ means that `u->next` is an effective pointer and is not equal to any of other pointers.)

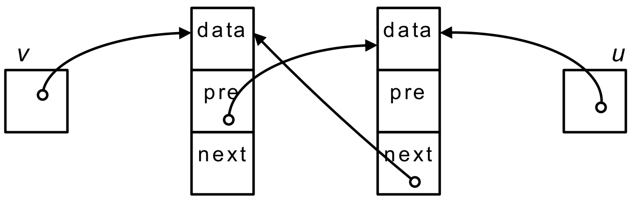

Fig. 2. Pointers in a doubly-linked cyclic list

5) $\mathcal{N}\backslash p$ or $\mathcal{D}\backslash p$ replaces the pointers in $\mathcal{N}$ or $\mathcal{D}$ which use p's aliases as their prefixes with their other aliases. $\mathcal{N}/p$ or $\mathcal{D}/p$ deletes the aliases of p from $\mathcal{N}$ or $\mathcal{D}$.

$$\begin{aligned}&\mathcal{N}\backslash p \triangleq\\ &\quad \{q : \mathcal{N} \mid \forall p' : \mathsf{closure}(p).\neg\mathsf{prefix}(p', q)\} \cup\\ &\quad \{q' \mid \exists q : \mathcal{N}.\exists p' : \mathsf{closure}(p).(\mathsf{prefix}(p', q) \wedge q' \equiv \mathsf{alias}(q, p))\}\end{aligned}$$

$$\mathcal{N}/p \triangleq \{q : \mathcal{N} \mid \neg(q \in \mathsf{closure}(p))\}$$

$$\mathcal{N}/\{p_1 \ldots p_n\} \triangleq ((\mathcal{N}/p_1) \ldots /p_n)$$

The formal definitions of $\mathcal{D}\backslash p$ and $\mathcal{D}/p$ are similar.

6) We use the set operator $\cup$ to represent the union of two pointer sets. Usually, this operator is used to add a pointer to a pointer set. And we also use set operators $\cup$ and $-$ and their combinations to represent the addition, deletion and substitution of equivalent sets.

7) Function $\mathsf{no\_leak}(p)$ tests the result of $\mathcal{S}/p$ for $\mathcal{S}$ containing p (just testing, not really deleting). This testing is to avoid memory leaks caused by an assignment to p.

$$\mathsf{no\_leak}(p) \triangleq \textbf{if } \mathsf{equals}(p)/p \,\texttt{!=}\, \emptyset \textbf{ then } \texttt{true} \textbf{ else } \texttt{false}$$

2.3 Axioms and inference rules

In this section, we define axioms and inference rules in our pointer logic. In the following rules, p and q are access paths; $p.\mathsf{eq}$ is p's equivalent set, and

$$p.\mathsf{eq} = \mathsf{equals}(p).$$

In each of the following rules, the premises of the rule should be computed based on the Ψ before the statement.

1) Pointer assignment: p=q.

Here different inference rules are used in different cases.

a) Both p and q are effective pointers or null pointers, and they are equal (including the case where p is a null pointer and q is constant `NULL`).

$$\frac{(p.\mathsf{eq}\,\texttt{!=}\emptyset \wedge p\texttt{==}q) \vee (p\texttt{==NULL} \wedge q\texttt{==NULL})}{\{\Pi \wedge \mathcal{N} \wedge \mathcal{D}\}\ p{=}q\ \{\Pi \wedge \mathcal{N} \wedge \mathcal{D}\}} \qquad \text{(RULE 1)}$$

b) Both p and q are effective pointers, and they are not equal.

$$\frac{(p.\mathsf{eq}\,\texttt{!=}\emptyset \wedge q.\mathsf{eq}\,\texttt{!=}\emptyset \wedge p\,\texttt{!=}q) \wedge \mathsf{no\_leak}(p)}{\{\Pi \wedge \mathcal{N} \wedge \mathcal{D}\}\ p{=}q\ \{((\Pi - \{q.\mathsf{eq}\})/p \cup \{q.\mathsf{eq}/p \cup \{p\}\}) \wedge \mathcal{N}\backslash p \wedge \mathcal{D}\backslash p\}} \qquad \text{(RULE 2)}$$

c) p is a null pointer and q is an effective pointer.

$$\frac{p\texttt{==NULL} \wedge q.\mathsf{eq}\,\texttt{!=}\emptyset}{\{\Pi \wedge \mathcal{N} \wedge \mathcal{D}\}\ p{=}q\ \{((\Pi - \{q.\mathsf{eq}\}) \cup \{q.\mathsf{eq} \cup \{p\}\}) \wedge \mathcal{N}/p \wedge \mathcal{D}\}} \qquad \text{(RULE 3)}$$

d) p is a dangling pointer and q is an effective pointer.

The rule for this case is similar to (RULE 3).

e) p is an effective pointer and q equals `NULL` (including the case where q is constant `NULL`).

$$\frac{p.\mathsf{eq}\,\texttt{!=}\emptyset \wedge q\texttt{==NULL} \wedge \mathsf{no\_leak}(p)}{\{\Pi \wedge \mathcal{N} \wedge \mathcal{D}\}\ p{=}q\ \{\Pi/p \wedge (\mathcal{N}\backslash p \cup \{p\}) \wedge \mathcal{D}\backslash p\}} \qquad \text{(RULE 4)}$$

f) p is a dangling pointer and q equals `NULL` (including the case where q is constant `NULL`).

$$\frac{p : \mathcal{D} \wedge q\texttt{==NULL}}{\{\Pi \wedge \mathcal{N} \wedge \mathcal{D}\}\ p{=}q\ \{\Pi \wedge (\mathcal{N} \cup \{p\}) \wedge \mathcal{D}/p\}} \qquad \text{(RULE 5)}$$

where $p : \mathcal{D}$ represents that an alias of p is in $\mathcal{D}$

2) The assignment axiom for non-pointer-type data is as follows. It is an extension of Hoare logic assignment axiom, and aliasing has been considered in this axiom. The axiom does not interfere with Ψ.

$$\{(\Psi \wedge Q)[y_1 \leftarrow x] \ldots [y_n \leftarrow x][x \leftarrow e]\}\ x{=}e\ \{\Psi \wedge Q\} \quad \text{(AXIOM 6-1)}$$

where $y_1, \ldots, y_n$ represent all the members of $\mathsf{closure}(x)$.

For assignment of pointer-type data, if besides $\mathit{\Psi}$, there is another Q which contains pointers (including pointers as prefixes), then another assignment axiom may also be useful:

$$\{\mathit{\Psi} \wedge Q\}\ p{=}q\ \{\mathit{\Psi}' \wedge Q[r \leftarrow p]\} \qquad \text{(AXIOM 6-2)}$$

where r is a member of $\mathsf{closure}(p)$ (not p itself), and $\mathit{\Psi}'$ can be derived from $\mathit{\Psi}$ using RULE 1 to 5. In another word, when assigning a pointer to p, the alias substitution of p should also occur in assertions besides $\mathit{\Psi}$.

Assertion Q in the above two axioms contains no pointer sets. And if there are any access paths in Q, they must be valid access paths according to current $\mathit{\Pi}$. All Qs in the following rules have this restriction.

3) *Composition*, *if-then-else*, *while* and *Consequence* rules are the same as the counterparts in Hoare logic. But note that $\mathit{\Psi}$ should not be strengthened or weakened in *Consequence* rule. For example, $\{p, q, r\}$ does not imply $\{p, q\}$, and $\{p, q\} \wedge \{r\}$ should not be weakened to $\{p, q\}$.

4) $\mathit{\Psi}$-modification axioms.

$$\mathit{\Psi} \wedge \neg(p : \mathit{\Psi}) \wedge (p\texttt{!=NULL}) \supset (\mathit{\Pi} \cup \{\{p\}\}) \wedge \mathcal{N} \wedge \mathcal{D} \qquad \text{(AXIOM 7)}$$

$$\mathit{\Psi} \wedge \neg(p : \mathit{\Psi}) \wedge (p\texttt{==NULL}) \supset \mathit{\Pi} \wedge (\mathcal{N} \cup \{p\}) \wedge \mathcal{D} \qquad \text{(AXIOM 8)}$$

where $p : \mathit{\Psi}$ indicates that an alias of p is in $\mathit{\Psi}$, and "$\supset$" denotes implication.

Some well-formed data structures , such as List and Binary Tree, contain no dangling pointer. So a pointer is either an effective pointer or a null pointer in such data structures. In a program, the effectiveness of such a pointer is usually determined through null-pointer test. Therefore, such a pointer can be added into $\mathit{\Pi}$ or $\mathcal{N}$ according to the different test results. These two axioms just modify $\mathit{\Pi}$ or $\mathcal{N}$ according to null-pointer test.

5) Allocation statement: p=$\mathsf{malloc}(t)$, where t is the type of the allocated storage. If t is a structure type, $r_1, \ldots, r_n$ represent pointer-type fields in t.

a) p is a null pointer.

$$\frac{p\texttt{==NULL}}{\{\mathit{\Pi} \wedge \mathcal{N} \wedge \mathcal{D}\}\ p{=}\mathsf{malloc}(t)\ \{(\mathit{\Pi} \cup \{\{p\}\}) \wedge \mathcal{N}/p \wedge (\mathcal{D} \cup \{p\texttt{->}r_1, \ldots, p\texttt{->}r_n\})\}} \qquad \text{(RULE 9)}$$

b) p is a dangling pointer. The inference rule for this case is similar to (RULE 9).

c) p is an effective pointer. In this case, the allocation statement can be regarded as a statement sequence p`=NULL;` p=malloc`(`t`);`. (RULE 4) and (RULE 9) can be used to handle this case.

6) Deallocation statement: $\mathsf{free}(p)$.

To simplify the following rule, we suppose the object pointed by p does not contain effective pointers. If the object pointed by p contains effective pointers, such as p->r_1,...,p->r_n, the deallocation statement can be regarded as p->r_1=`NULL; ...;` p->r_n=`NULL;` `free(`p`);`, and then handled by several rules.

$$\frac{p.\mathsf{eq}\,!\!=\emptyset}{\{\Pi \wedge \mathcal{N} \wedge \mathcal{D}\}\ \mathsf{free}(p)\ \{(\Pi - \{p.\mathsf{eq}\}) \wedge \mathcal{N}/\{p\text{->}r_1, \ldots, p\text{->}r_n\} \wedge (\mathcal{D} \cup p.\mathsf{eq})\}}$$

(RULE 10)

7) Function

Functions handled by our inference rules have the following restrictions:

– Pointer-type formal parameters must be read-only, and should not be used as the actual parameter of free.
– If the formal parameter is an effective pointer, the equivalent set of the corresponding actual parameter should only contain the actual parameter itself. This restriction is helpful for local reasoning. If a call statement does not obey this restriction, it can be easily transformed into a statement sequence which does obey the restriction.
– In function call ret=f(...), if the return type of the function f is pointer type, ret should not be an effective pointer at the call point. A call statement can be easily transformed to satisfy the restriction if necessary.

In the following rules, the function has only one formal parameter, and rules for multiple-parameters cases are similar. *arg* represents the formal parameter, and *act* represents the corresponding actual parameter. To simplify the inference rule for return, we assume that in the function there is a local variable `res` whose name is different from any declared local variables, and the statement `return` e; is always transformed to the statement sequence `res =` e`;` `return res;` when being reasoned about.

We only show the inference rule for function call ret=f(act), where both the parameter type and the return type of function f are pointer types. The rules for other cases are simpler. Variables v_1, ..., v_k denote all the local variables with pointer types.

a) Actual parameter *act* is an effective pointer, and *ret* is a null pointer. (Q_{arg} is the assertion which only concerns *arg*, and *f_body* is the body of the function f.)

$$\frac{\{\{arg\} \wedge \{v_1, \ldots, v_k, res\}_{\mathcal{D}} \wedge Q_{arg}\}\ f\_body\ \{\Psi \wedge Q\}}{\{(\{arg\} \wedge \{ret\}_{\mathcal{N}} \wedge Q_{arg})[arg \leftarrow act]\}\ ret{=}\mathsf{f}(act)\ \{(\Psi \wedge Q)[arg \leftarrow act][res \leftarrow ret]\}}$$

(RULE 11)

If *act* is an ineffective pointer, the inference rule is similar to (RULE 11).

b) return statement.

Since local variables of a function can not be accessed after the function has returned, pointer information about them should be deleted from Ψ after the function has returned, and memory leaks should also be checked. The following two inference rules are used for this purpose. ($Q_{\texttt{res}}$ is the assertion which only concerns `res`.)

$$\frac{\{\Psi_0\}\ v_1\texttt{=NULL}\ \{\Psi_1\} \qquad \ldots \qquad \{\Psi_{k-1}\}\ v_k\texttt{=NULL}\ \{\Pi_k \wedge \mathcal{N}_k \wedge \mathcal{D}_k\}}{\{\Psi_0\}\ \textsf{return}\ \texttt{res}\ \{\Pi_k \wedge (\mathcal{N}_k - \{v_1, \ldots, v_k\}) \wedge \mathcal{D}_k\}} \quad \text{(RULE 12)}$$

$$\frac{\{\Psi\}\ \textsf{return}\ \texttt{res}\ \{\Psi'\}}{\{\Psi \wedge Q_{\texttt{res}}\}\ \textsf{return}\ \texttt{res}\ \{\Psi' \wedge Q_{\texttt{res}}\}} \quad \text{(RULE 13)}$$

You can refer to section 5 for an example of the usage of these rules.

3 Pointer logic at assembly level

At assembly level, we have defined a target machine to simulate the behavior of the actual machine. The reasoning of assembly code is based on the target machine. A framework for assembly code certification, called Function-based Certified Assembly Programming (FCAP)[2], has been defined to reason about primary stack safety and integer-related properties of assembly programs. This framework is much similar to CAP and SCAP [5, 6]. To reason about pointer-related properties at assembly level, another pointer logic system, which is similar to that at source level, has also been defined in FCAP. In this section, we present the assembly-level pointer logic system and we only present its distinctions from that at source level.

If the operands of an instruction have pointer types, the instruction is a pointer operation. In order to know whether an instruction is a pointer operation, we have designed a simple type system at assembly level. We defined a typing environment based on the target machine. For each structure type, the typing environment indicates the size of the structure (the structure size is used to avoid out-of-bound accesses to dynamically allocated storage), and the offsets of pointer fields in this structure. For each stack frame (*i.e.*, function), the typing environment also indicates the offsets which contain pointers (*i.e.*, the local variables or parameters with pointer types). Unlike at source level, the types of some objects in the assembly-level typing environment are mutable during function execution. For example, registers and the top of stack can store temporary values with different types. We can determine whether the current instruction is a pointer operation by referring the typing environment, and we can also modify the mutable parts of the typing environment. See the article in our website[3] for the design details of the type system. Besides type checking, the information in the typing environment can also be used for computing Π, $\mathcal{N}$ and $\mathcal{D}$. For example, information of structure types is used to adjust $\mathcal{D}$ when a dynamic storage

[2] Li Z P. A framework of function-based certified assembly programming. http://ssg.ustcsz.edu.cn/lss/doc/index.html.
[3] Li Z P. Coq implementation of the soundness proof of FCAP. http://ssg.ustcsz.edu.cn/lss/software/index.html.

is requested; type information of local variables is used to adjust $\mathcal{D}$ at the entry point of a function.

The type system can only determine whether an instruction is a pointer operation, and the legality of the pointer operation is determined by the pointer logic system at assembly level. The relationship and main differences between the pointer logic systems at assembly level and source level are as follows:

1. The type system at assembly level is not independent, and all of its typing rules are merged into the corresponding pointer logic inference rules.
   Since the type system at assembly level only concerns typing of pointers and the inference rules in assembly-level pointer logic system also concern pointers, they can be combined together. See (RULE 14).
2. The inference rules for assignments in the source-level pointer logic can be easily modified to be the inference rules for the corresponding assembly instructions.
3. The assembly level pointer logic system has some special assignment rules.
4. The inference rules for function call or `return` statement in the source-level pointer logic correspond to the combination of the rules of several instructions at assembly level.

We will explain the last 3 points one by one.

Besides call instructions, including `malloc` and `free`, the instructions which may change the pointer information include `movl`, `pushl` *etc.* (Here we call them assignment instructions.) The inference rules for these instructions in the assembly-level pointer logic are almost the same as the corresponding assignment rules at source level, except that variables are replaced with 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.

For example, suppose p and q are effective pointers and they are not equal, an inference rule in the assembly-level pointer logic system is:

$$\frac{\mathsf{pointer\_assign}(p,q) \wedge (p.\mathsf{eq}\,!\!=\!\emptyset \wedge q.\mathsf{eq}\,!\!=\!\emptyset \wedge p\,!\!=\!q) \wedge \mathsf{no\_leak}(p)}{\{\Pi \wedge \mathcal{N} \wedge \mathcal{D}\}\ \texttt{movl}\ q,p\ \{((\Pi - \{q.\mathsf{eq}\})/p \cup \{q.\mathsf{eq}/p \cup \{p\}\}) \wedge \mathcal{N}\backslash p \wedge \mathcal{D}\backslash p\}} \quad \text{(RULE 14)}$$

This rule is almost the same as the corresponding source-level rule, except that it has an additional premise for determining whether the instruction is a pointer operation ($\mathsf{pointer\_assign}(p,q)$).

Registers can store both integers and pointers, so there may be instructions whose operands have no explicit types. For example, the instruction `movl -8(%ebp), %eax`, where `-8(%ebp)` has pointer type, and `eax` has no explicit type. The same problem arises in instruction `pushl`. And furthermore, after the execution of `pushl`, the access paths in Ψ, which begin with `esp`, will need adjustment. Such instructions must have special rules. Luckily, it is straightforward to design rules for them according to the assignment rules in the source-level pointer logic. In source programs, such cases do not exist because variable types are not mutable.

A function call statement or a return statement at source level is compiled into several instructions at assembly level. The change of pointer information caused by the call or return statement at source level corresponds to the change caused by those instructions at assembly level. So the rule for function call or return statement at source level corresponds to the composition of several rules at assembly level. For example, a function call statement "`a = f(5);`" will be compiled into the following instructions:

```
pushl   $5        –push the parameter
call    f         –push return address and transfer the control to f
movl    %eax, a   –store the return value to  a
```

Rules for these three instructions must be composed to achieve the influence of the function call rule at source level.

The soundness of FCAP (including the assembly-level pointer logic system in it) is similar to that of SCAP [6]. Its aim is to guarantee that the runtime behavior of a program will satisfy its specifications when it can pass the static proof-checking. To prove the soundness theorem, we need to prove three lemmas first. Two of them are "Progress" and "Preservation", which are the same as those in SCAP. The other is to confirm that the assembly-level pointer logic system is sound with respect to the operational semantics of the target machine.

We have finished proving the soundness theorem formally using Coq[4]. See the article in our website[5] for the soundness theorem and its proof.

4 Certifying compiler

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.

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

[5] Li Z P. Coq implementation of the soundness proof of FCAP. http://ssg.ustcsz.edu.cn/lss/software/index.html.

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 by the programmers. The VCs and their proofs show that the source program satisfies its specifications.

In this section, we present our work on the certifying compiler. We focus on the modules for reasoning about pointer-related properties, including verification condition generation, code and assertion generation, and proof generation.

4.1 Verification condition generation

At first, we discuss how to generate VCs for functions with pointer-type data. Figure 3 shows the structure of a function chosen from PointerC syntax, some irrelevant details are omitted.

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

Fig. 3. Structure of function

The pointer logic is fit for collecting pointer information forward, so the VC generation is based on the strongest postcondition calculus, which is also in a forward manner. Figure 4 shows the main rules for the strongest postcondition calculus (function sp) in the pointer logic and the procedure of VC generation (for data with pointer type). These rules are recursively defined according to the syntactic structures in a function. In Fig. 4, 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 in this paper 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).
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

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 $\mathsf{sp}(lval = id(Exp),\ \Psi) =$ $(\Psi' \wedge Q)[arg \leftarrow Exp][\texttt{res} \leftarrow lval]$, 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 Fig. 3). 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: $\mathsf{sp}(\mathsf{return}\ \texttt{res},\ \Psi) = \Psi$

Fig. 4. The Strongest postcondition calculus and the VC generation for pointer-type data

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 Fig. 4).
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. Such details

are not presented in Fig. 4, and interested readers could refer to another article[6].

5. At the entry point of a function, the initial values of the pointer-type formal parameters and local variables should be checked in the pointer logic. And at the exit of the function, the effectiveness of them should also be checked to avoid memory leaks. But for the space limit, the VC generation in Fig. 4 does not reflect this.

For integer-type data, we adopted a complemented approach of Hoare logic—weakest precondition calculus (`wp`) [11]—to generate assertions or VCs at each program point. Note that 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.

4.2 Code and assertion generation

In our FCAP 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 instructions; and in our design, the instruction sequence is ended 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 Hoare triple $\{P\}\ B\ \{Q\}$. 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 ensure that each basic block has pertinent pre- and postconditions, assertions should be generated at proper program points during code generation. Using the calculi in section 4.1, we can get and insert a proper assertion at any program point, and we can also get proper VCs. If we only consider pointer-related assertions, all the work can be done via a one-pass inspection of source programs during compilation. But 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.

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

[6] Chen Y Y, Hua B J, Ge L, *et al.* A pointer logic for safety verification of pointer programs. http://ssg.ustcsz.edu.cn/lss/papers/index.html. (in Chinese)

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.
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 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.

4.3 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 Fig. 4 and their proofs respectively. And they are usually used in proving that a basic block's postcondition should imply its successor's precondition.

The proof of a basic block satisfying its pre- and postconditions is produced as follows: first, assertions between the instructions in the basic block are deduced in a forward manner. These assertions can be generated from the pre- and postconditions of the basic block according to the inference rules of the assembly-level pointer logic system. In each step of the deduction, the evidence on which the instruction is determined as a pointer operation and the inference rule used in the deduction are recorded and output together with the basic block. We will explain these using an example in the next section.

5 Example: deleting a node from BST

In this section, we take the function

```
struct node * DeleteNode(struct node *p)
```

as an example to present the application of the pointer logic. We will show how to prove safety of a pointer program, how to translate pointer-related assertions and how to prove that the compiled assembly code satisfies the translated assertions. The function `DeleteNode` deletes a node from a binary search tree and reconnects its left or right child.

The parameter of the function is a pointer which points to a tree. We don't know the precise layout of pointers in the tree, but we do know that the tree satisfies the following definition. Suppose tree node is defined as

```
struct node
{int data; struct node *l; struct node *r; },
```

the definition of the tree is:

$$\mathsf{tree}(p) \triangleq \{p\}_{\mathcal{N}} \vee (\{p\} \wedge \mathsf{tree}(p\texttt{->l}) \wedge \mathsf{tree}(p\texttt{->r})),$$

where p is the root of the tree. This definition, as well as some properties derived from it could be used in the proofs. Two of the properties are used in the following example: one is that there is no dangling pointer in a tree, the other is that effective pointers in a tree vary from one another.

Figure 5 shows the annotated source program of the example and the assertions between statements. These assertions are generated according to the pointer logic. For the conditional branch which states that neither the left nor the right child of the parameter `p` is null (the function requires `p` not null), most of the assertions are inserted between the statements; for other parts of the code, only the assertions at some key points are inserted.

According to the rule for function call, the following formula holds.

$\pmb{\{}\{q1\} \wedge \{q2\}_{\mathcal{N}} \wedge \mathsf{tree}(q1)\pmb{\}}$
$q2$ `= DeleteNode(`$q1$`)`
$\pmb{\{}(\{q1, q2\} \vee (\{q2\} \wedge \{q1\}_{\mathcal{D}}) \vee \{q1\}_{\mathcal{D}}) \wedge \mathsf{tree}(q2)\pmb{\}}$

And after the assignment $q1$=NULL, we derive a clearer postcondition: $\pmb{\{}\{q1\}_{\mathcal{N}} \wedge \mathsf{tree}(q2)\pmb{\}}$

We take the asserted program fragment

$\pmb{\{}\{\texttt{p,q}\} \wedge \{\texttt{p->r}\} \wedge \{\texttt{p->l,s}\} \wedge \{\texttt{s->r}\} \wedge \ldots\pmb{\}}$
```
  q = s; s = s->r;
```
$\pmb{\{}\{\texttt{p}\} \wedge \{\texttt{p->r}\} \wedge \{\texttt{p->l, q}\} \wedge \{\texttt{q->r, s}\} \wedge \ldots\pmb{\}}$

as an example to explain how to translate assertions and how to prove that a basic block satisfies its pre- and postconditions. (Note that these pre- and postconditions are pointer-related assertions.) Its corresponding basic block at assembly level and the pre- and postconditions corresponding are as follows:

$\pmb{\{}\{\texttt{ebp->8}, \texttt{ebp->-4}\} \wedge \{\texttt{ebp->8->8}\} \wedge \{\texttt{ebp->8->4}, \texttt{ebp->-8}\} \wedge$
$\{\texttt{ebp->-8->8}\} \wedge \ldots\pmb{\}}$
```
  movl -8(%ebp), %eax
  movl %eax, -4(%ebp)
  movl -8(%ebp), %eax                                 (1)
  movl 8(%eax), %eax
  movl %eax, -8(%ebp)
```
$\pmb{\{}\{\texttt{ebp->8}\} \wedge \{\texttt{ebp->8->8}\} \wedge \{\texttt{ebp->8->4}, \texttt{ebp->-4}\} \wedge$
$\{\texttt{ebp->-4->8}, \texttt{ebp->-8}, \texttt{eax}\} \wedge \ldots\pmb{\}}$

```
$\boldsymbol{\{}\{\texttt{p}\} \wedge \mathsf{tree}(\texttt{p})\boldsymbol{\}}$
struct node * DeleteNode(struct node *p)
{ struct node *q, *s;
  $\boldsymbol{\{}\{\texttt{p}\} \wedge \{\texttt{q, s, res}\}_{\mathcal{D}} \wedge \mathsf{tree}(\texttt{p})\boldsymbol{\}}$
  if(p->r==NULL) /* right child is null, reconnect left child */
  { q = p; s = p->l; free(q); return s;     $\boldsymbol{\{}\{\texttt{p}\}_{\mathcal{D}} \wedge \mathsf{tree}(\texttt{res})\boldsymbol{\}}$}
  else if(p->l==NULL)/* left child is null, reconnect right child */
  { q = p; s = p->r; free(q); return s;     $\boldsymbol{\{}\{\texttt{res}\} \wedge \{\texttt{p}\}_{\mathcal{D}} \wedge \mathsf{tree}(\texttt{res})\boldsymbol{\}}$}
  else     /* neither the left nor the right child is null */
  { $\boldsymbol{\{}\{\texttt{p}\} \wedge \{\texttt{p->l}\} \wedge \{\texttt{p->r}\} \wedge \{\texttt{q, s, res}\}_{\mathcal{D}} \wedge \mathsf{tree}(\texttt{p->l}) \wedge \mathsf{tree}(\texttt{p->r})\boldsymbol{\}}$
    q = p; s = p->l;
    if(s->r == NULL)     /* reconnect *q's left child */
    { q->l = s->l; p->data = s->data; free(s); return p;
      $\boldsymbol{\{}\{\texttt{p, res}\} \wedge \mathsf{tree}(\texttt{res})\boldsymbol{\}}$}
    else
    { $\boldsymbol{\{}\{\texttt{p, q}\} \wedge \{\texttt{p->r}\} \wedge \{\texttt{p->l, s}\} \wedge \{\texttt{s->r}\} \wedge \{\texttt{res}\}_{\mathcal{D}} \wedge$
        $\mathsf{tree}(\texttt{p->l->l}) \wedge \mathsf{tree}(\texttt{p->l->r}) \wedge \mathsf{tree}(\texttt{p->r})\boldsymbol{\}}$
      q = s; s = s->r;
      $\boldsymbol{\{}\exists n : N.(\{\texttt{p}\} \wedge \{\texttt{p->r}\} \wedge \forall i : 0..n-1.\{\texttt{p->l(->r)}^{i}\} \wedge \{\texttt{p->l(->r)}^{n}, \texttt{q}\} \wedge$
        $\{\texttt{p->l(->r)}^{n+1}, \texttt{s}\} \wedge \{\texttt{res}\}_{\mathcal{D}} \wedge$
        $\forall i : 0..n.\mathsf{tree}(\texttt{p->l(->r)}^{i}\texttt{->l}) \wedge \mathsf{tree}(\texttt{p->l(->r)}^{n+1}) \wedge \mathsf{tree}(\texttt{p->r}))\boldsymbol{\}}$
        /* — loop invariant */
      while(s->r != NULL)     /* turn left, and then go on to the end of the right side */
      { q = s; s = s->r;}
        $\boldsymbol{\{}\exists n : N.(\{\texttt{p}\} \wedge \{\texttt{p->r}\} \wedge \forall i : 0..n-1.\{\texttt{p->l(->r)}^{i}\} \wedge \{\texttt{p->l(->r)}^{n}, \texttt{q}\} \wedge$
          $\{\texttt{p->l(->r)}^{n+1}, \texttt{s}\} \wedge \{\texttt{s->r}\}_{\mathcal{N}} \wedge \{\texttt{res}\}_{\mathcal{D}} \wedge$
          $\forall i : 0..n.\mathsf{tree}(\texttt{p->l(->r)}^{i}\texttt{->l}) \wedge \mathsf{tree}(\texttt{p->l(->r)}^{n+1}) \wedge \mathsf{tree}(\texttt{p->r}))\boldsymbol{\}}$
        p->data = s->data;
        q->r = s->l;     /* reconnect *q's right child */
        $\boldsymbol{\{}\exists n : N.(\{\texttt{p}\} \wedge \{\texttt{p->r}\} \wedge \forall i : 0..n-1.\{\texttt{p->l(->r)}^{i}\} \wedge \{\texttt{p->l(->r)}^{n}, \texttt{q}\} \wedge \{\texttt{s}\} \wedge$
          $((\{\texttt{s->l}, \texttt{q->r}\} \wedge \{\texttt{s->r}\}_{\mathcal{N}}) \vee \{\texttt{s->r}, \texttt{s->l}, \texttt{q->r}\}_{\mathcal{N}}) \wedge \{\texttt{res}\}_{\mathcal{D}} \wedge$
          $\forall i : 0..n.\mathsf{tree}(\texttt{p->l(->r)}^{i}\texttt{->l}) \wedge \mathsf{tree}(\texttt{p->l(->r)}^{n+1}) \wedge \mathsf{tree}(\texttt{p->r}))\boldsymbol{\}}$
        free(s);
        $\boldsymbol{\{}\exists n : N.(\{\texttt{p}\} \wedge \{\texttt{p->r}\} \wedge \forall i : 0..n-1.\{\texttt{p->l(->r)}^{i}\} \wedge \{\texttt{p->l(->r)}^{n}, \texttt{q}\} \wedge$
          $(\{\texttt{q->r}\} \vee \{\texttt{q->r}\}_{\mathcal{N}}) \wedge \{\texttt{s, res}\}_{\mathcal{D}} \wedge$
          $\forall i : 0..n.\mathsf{tree}(\texttt{p->l(->r)}^{i}\texttt{->l}) \wedge \mathsf{tree}(\texttt{p->l(->r)}^{n+1}) \wedge \mathsf{tree}(\texttt{p->r}))\boldsymbol{\}}$
        return p;
        $\boldsymbol{\{}\{\texttt{p, res}\} \wedge \mathsf{tree}(\texttt{res})\boldsymbol{\}}$
    }
  }
}
```

Fig. 5. Example: deleting node from BST

In these instructions, the addresses of `p`, `q` and `s` in the stack are represented by `8(%ebp)`, `-4(%ebp)` and `-8(%ebp)` respectively; the offsets of the fields `l` and `r` in structure `tree` are 4 and 8 respectively; `ebp` is the base pointer of current stack frame.

Now, we explain how to obtain a proof-carrying basic block. The behavior of the first instruction here is to assign an effective pointer to a null pointer. And according to the corresponding inference rule of the pointer logic, we can get the precondition of the next instruction:

$$\begin{aligned}&\boldsymbol{\{}\{\texttt{ebp->8}, \texttt{ebp->-4}\} \wedge \{\texttt{ebp->8->8}\}\wedge \\ &\ \{\texttt{ebp->8->4}, \texttt{ebp->-8}, \texttt{eax}\} \wedge \{\texttt{ebp->-8->8}\} \wedge \ldots\boldsymbol{\}} \end{aligned} \tag{2}$$

The rest instructions are all assignments between effective pointers. (RULE 14) presented in section 3 can be applied to all these instructions except the third one which is an assignment between equal pointers. The preconditions of the third, fourth and fifth instructions are:

$$\begin{aligned}&\boldsymbol{\{}\{\texttt{ebp->8}\} \wedge \{\texttt{ebp->8->8}\} \wedge \{\texttt{ebp->-8->8}\}\wedge \\ &\ \{\texttt{ebp->8->4}, \texttt{ebp->-8}, \texttt{ebp->-4}, \texttt{eax}\} \wedge \ldots\boldsymbol{\}} \end{aligned} \tag{3}$$

$$\begin{aligned}&\boldsymbol{\{}\{\texttt{ebp->8}\} \wedge \{\texttt{ebp->8->8}\} \wedge \{\texttt{ebp->-8->8}\}\wedge \\ &\ \{\texttt{ebp->8->4}, \texttt{ebp->-8}, \texttt{ebp->-4},\ \ \texttt{eax}\} \wedge \ldots\boldsymbol{\}} \end{aligned} \tag{4}$$

$$\begin{aligned}&\boldsymbol{\{}\{\texttt{ebp->8}\} \wedge \{\texttt{ebp->8->8}\} \wedge \{\texttt{ebp->-8->8}, \texttt{eax}\}\wedge \\ &\ \{\texttt{ebp->8->4}, \texttt{ebp->-8},\ \ \texttt{ebp->-4}\} \wedge \ldots\boldsymbol{\}} \end{aligned} \tag{5}$$

The postcondition of the fifth instruction is also the postcondition of the basic block. The asserted basic block (1), concatenated with the assertion sequence (2)-(5) and the rules to derive these assertions, forms a proof of (1).

Note that all the assertions at assembly level are parameterized by a machine state except Π, $\mathcal{N}$ and $\mathcal{D}$, because Π, $\mathcal{N}$ and $\mathcal{D}$ are concentrated on the equality of pointers rather than the values of them. Moreover, the change of Π, $\mathcal{N}$ and $\mathcal{D}$ caused by the change of a state has already been embodied in the inference rules of the pointer logic.

6 Experimental results

The certifying compiler is still under active development and we continue to extend the range of programs being compiled and certified. In this section we present some early experimental results with the intension to shed some light on the questions such as how large the safety proofs are, what percentage of the safety proofs can be derived automatically and how expensive proof checking compared to certifying compilation.

We show in Table 1, for a few of our internal test cases the proof sizes compared with the code sizes, the automatically generated proof sizes compared with total proof sizes (including those written manually) and the compilation times (including VC and proof generating times) compared with the proof checking times. This data shows that the sizes of proofs for these programs are rather large compared with the programs themselves, but over 70% proofs can be generated automatically and the proof checking time is negligible compared with the compilation time. All measurements were performed on a machine using an Intel Celeron CPU running at 2 GHz with 1 G memory.

Test case	reversal	insert	clearList	creatList	preorder
Code size (byte)	734	653	518	504	431
Total proof size (kb)	37.1	41.4	28.4	39.3	28.2
Auto-generated proof size (kb) (% of total proof size)	26.4 (71%)	34.3 (83%)	21.5 (76%)	27.5 (70%)	22.1 (78%)
Compilation time (ms)	1186	7402	194	10710	1870
Checking time (ms)	20	22	14	21	13

Table 1. The experimental results for a few of our internal compiler test cases.

7 Related work

An important characteristic of Hoare logic is its use of variable substitution to capture the semantics of assignments. The extension to Hoare logic in this paper is essentially a kind of pointer-analysis tool which captures the influences of pointer operations using addition, deletion and substitution of access paths. Pointer analysis has been studied for over 20 years, and in history it mainly tried to answer the question: what is the possible set of objects pointed to by a pointer at runtime? Like other static techniques, pointer analysis is bothered with undecidability, so for most languages, the solution is always approximative.

Different applications of pointer analyses demand different precisions and efficiencies. And precisions and efficiencies can be achieved by different algorithms of analyses. For example, Bjarne Steensgaard presented a flow-insensitive, interprocedural, context-insensitive points-to analysis for a small imperative pointer language which captured the important properties of languages like C [12]. The algorithm is based on a non-standard type system and uses type inference to perform points-to analysis. Marc Berndl *et al.* used Binary Decision Diagrams to solve a flow-insensitive, context-insensitive points-to analysis, which solved the efficiency problems [13]. Michael Hind summarized the related work on pointer analysis and outlined some problems that remained open [14].

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. These restrictions do not influence the language functionality of PointerC. And they make it possible for us to express the collection of pointer information in the pointer logic rules.

As for proving program properties, Bornat also used Hoare logic to reason about properties of pointer programs [15]. Inspired by the work of Burstall [16], he treated the heap as a pointer-indexed collection of objects, each of which was a name-indexed collection of components. Then he extended the assignment axiom of Hoare logic to accommodate the assignment of object components and used it to prove some properties of pointer programs. Mehta and Nipkow adopted a similar approach when reasoning about pointer programs in a higher-order logic system Issabellet/HOL [17]. And so did Marché *et al.* in their prototype tool Caduceus [18].

The common feature in the approaches of Bornat's and ours is to extend the assignment axiom of Hoare logic based on the judgement of pointer aliasing through the equivalence of pointer indexes. But Bornat's approach can only be used in the languages without explicit deallocation (the memory managements of these languages usually depend on garbage collection). Our approach can be used to support deallocation, but this also increases the complexity of the pointer logic. For example, considering free(p), to avoid dangling pointer dereferences, we need to guarantee that there is no access to the freed object through p or any other pointers which equal p; when assigning something to an effective pointer p, to avoid memory leaks, we need to make sure that at least one other pointer equals p.

In the research field of certified compiler, Moore was one of the first to mechanically verify semantic preservation for a compiler [19], 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 [20] by Leroy. A front end translating a subset of C to Cminor is under development and certification. One of the novel features of Leroy's 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. Therefore, certifying compiler has more possibility to be used in practice first than certified compiler. Necula's Touchstone [8] is composed of a traditional optimizing compiler and a certifier which automatically produces a proof for each assembly program. A proof checker can be used to automatically check the generated proofs. Since the source programs compiled by Touchstone are written in a very small safe subset of C, their type safety and memory safety are easy to be checked. The major advance of Special J [9] over Touchstone is the scope of the source language compiled, which in turn necessitates the handling of non-trivial run-time mechanisms such as object representation, dynamic method dispatch and exception handling. Moreover, Special J is 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.
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.
3. 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.

8 Conclusion

In order to take the technology of certifying compiler and PCC into a more realistic programming language, we have designed a pointer logic system for PointerC and a reasoning framework for Intel x86 assembly code, and we have also designed a certifying compiler for PointerC and implemented its prototype. This prototype is now able to generated proof-carrying code for some functions about singly-linked list or binary search tree. Such proof-carrying code can be automatically checked by a proof checker in our prototype.

In future work, we plan to relax the restrictions on pointer arithmetic operations and allow calloc which is ubiquitous in C programs. Although we have proved the soundness of the assembly-level reasoning system, it is still necessary to prove the safety of PointerC and the soundness of the source-level pointer logic system. Moreover, a deeper comparison between separation logic and our pointer logic will be discussed in future. Improving the modularity of program-property reasoning by augmenting the language with object-oriented structures will also be considered.

In the implemented prototype, integer-related VCs are proved using Coq by programmers and pointer-related ones are proved by a simple theorem prover which is embedded in the prototype. The compiler design, as well as the proof-checking at assembly level, suffers from the inconsistency of the two kinds of VCs. So we plan to use an embedded theorem prover for all VCs in the next version of our certifying compiler. And this will make it easy to design and simplify the syntax of the proof or proof hint. The influence of PCC and certifying compiler on code optimization is the further future work to be considered.

Acknowledgments 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. G. C. Necula. Proof-carrying code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. New York: ACM Press, 1997, 106–119.
2. J. G. Morrisett, D. Walker, K. Crary, *et al.* From system F to typed assembly language. In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. New York: ACM Press, 1998, 85–97.
3. Y. Mandelbaum, D. Walker, R. Harper. An effective theory of type refinements. In: Proceedings of the 8th ACM SIGPLAN international conference on Functional programming. New York: ACM Press, 2003, 213–225.
4. A. W. Appel. Foundational proof-carrying code. In: Proceedings of the 16th Annual IEEE Symposium on Logic in computer science. Washington: IEEE Computer Society, 2001, 247–258.
5. D. C. Yu, N. A. Hamid, Z. Shao. Building certified libraries for pcc: dynamic storage allocation. Science of Computer Programming, 2004, 50(1-3): 101–127.
6. X. Y. Feng, Z. Shao, A. Vaynberg, *et al.* Modular verification of assembly code with stack-based control abstractions. In: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation. New York: ACM Press, 2006, 401–414.
7. H. W. Xi. Applied type system (extended abstract). In: post-workshop Proceedings of TYPES 2003. Lecture Notes in Computer Science, Vol 3085. Berlin: Springer-Verlag, 2004, 394–408.
8. G. C. Necula, P. Lee. The design and implementation of a certifying compiler. In: Proceedings of the 1998 ACM SIGPLAN Conference on Prgramming language design and implementation. New York: ACM Press, 1998, 333–344.
9. C. Colby, P. Lee, G. C. Necula, *et al.* A certifying compiler for Java. ACM SIGPLAN Notices, 2000, 35(5): 95–107.
10. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In: Proceedings of the 17th Annual IEEE Symposium on Logic in computer science. Washington: IEEE Computer Society, 2002, 55–74.
11. E. W. Dijkstra. A discipline of programming. Englewood Cliffs, New Jersey: Prentice-Hall, 1976.
12. B. Steensgaard. Points-to analysis in almost linear time. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. New York: ACM Press, 1996, 32–41.
13. M. Berndl, O. Lhoták, Qian F., *et al.* Points-to analysis using BDDs. In: Proceedings of the 2003 ACM SIGPLAN Conference on Programming language design and implementation. New York: ACM Press, 2003, 103–114.
14. M. Hind. Pointer analysis: haven't we solved this problem yet? In: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. New York: ACM Press, 2001, 54–61.
15. R. Bornat. Proving pointer programs in Hoare logic. In: Proceedings of the 5th International Conference on Mathematics of program construction. London: Springer-Verlag, 2000, 102–126.

16. R.M. Burstall. Some techniques for proving correctness of programs which alter data structures. Machine Intelligence, New York: American Elsevier, 1972, 7:23–50.
17. F. Mehta, T. Nipkow. Proving pointer programs in higher-order logic. Information and Computation, 2005, 199(1-2):200–227.
18. J. C. Filliâtre, C. Marché. Multi-Prover Verification of C Programs. In: Proceedings of the 6th International conference on formal engineering methods. Seattle: Springer-Verlag, 2004, 15–29.
19. J. S. Moore. Piton: a mechanically verified assembly-language. Norwell: Kluwer Academic Publishers, 1996.
20. X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. New York: ACM Press, 2006, 42–54.