一个出具证明编译器原型系统的实现

刘 诚,陈意云,葛 琳,华保健
LIU Cheng,CHEN Yi-yun,GE Lin,HUA Bao-jian

中国科学技术大学 计算机科学技术系,合肥 230027
Department of Computer Science and Technology,University of Science and Technology of China,Hefei 230027,China
E-mail:liuc5@mail.ustc.edu.cn

LIU Cheng,CHEN Yi-yun,GE Lin,et al.Implementation of certifying compiler prototype.Computer Engineering and Applications,2007,43(21):99-102.

Abstract: Certifying compiler is such a tool coming up with the increasing demands for higher reliability and safety of computer software.It combines the techniques in programming languages and program verifications.This paper describes some details in implementing our prototype of a certifying compiler.

Key words: software safety;certifying compiler;verification condition;formal proof method;proof generator

摘 要:出具证明编译器是随着人们对现今的软件提出更高的可靠性和安全性要求而产生的工具,它结合了以往程序设计和程序安全性证明的技术。论文介绍了一个出具证明编译器原型系统的实现。

关键词:软件安全;出具证明编译器;验证条件;形式化证明方法;证明生成器
文章编号:1002-8331(2007)21-0099-04 **文献标识码**:A **中图分类号**:TP311

1 引言

随着计算机科学的迅速发展,计算机软件逐渐演化为庞大的软件体系。现在人们在考虑软件执行速度和效率的同时,也开始关注于软件的可靠性和安全性。出具证明的编译器(Certifying Compiler)就是随着人们要求软件具有更高的可靠性和安全性而出现的一个研究方向。它能够在编译器生成目标代码的同时也生成与之对应的证明,通过证明检查来保证目标代码满足一定的性质。

在关于出具证明编译器的研究中,Necula 最先提出了 Proof-Carrying Code(PCC)[1]的概念,并为一个类型安全的 C 语言子集 SafeC 实现了出具证明的编译器(Touchstone)[2]。但是他限制了其中的指针类型和动态存储管理,因此这个编译器只能用于处理一些数据结构比较简单的程序。之后,Colby 等人针对 Java 语言的较大子集实现了一个从字节码编译到目标代码的出具证明编译器(Special J)[3],和 Touchstone 相比,它拥有管理对象,动态方法调度,异常处理的能力,并且还加入了更多的代码优化方面的考虑。

基于已有的研究成果,设计并实现了一个出具证明编译器的原型系统[9]。与以往的出具证明编译器相比,实现的原型系统能够接受具有更丰富的关于指针类型及其运算的源程序,并且还提供显式的动态存储管理函数,更适合用于系统程序的编写。同时,程序的设计者可以使用源语言级的断言语言来表达他所编写的源代码应该满足的程序性质,并借助定理证明辅助工具完成相应的证明,而后由出具证明编译器生成描述目标代码级程序性质的证明。这样的原型系统向程序设计者提供源代码级而不是目标代码级的程序性质证明方法,提高了安全程序的开发效率。本文将详细介绍这个出具证明编译器原型系统中和整数类型相关的程序性质的证明以及实现,文章的组织如下:第二部分将介绍原型系统的工作流程;第三部分简要介绍了源语言级证明程序性质的框架;第四部分介绍目标代码级程序性质证明框架;第五部分描述原型系统中的证明生成,其中包括了代码生成器和证明生成器的接口、断言和证明的生成;第六部分是总结。

2 原型系统的工作流程

我们设计并实现的出具证明编译器原型系统的工作流程可以表示在图 1 中。

首先,编译器接受程序设计者使用一种称为 PointerC 的类 C 的程序设计语言编写的源程序代码,其中含有使用源语言级的断言语言描述的程序应该满足的性质;然后,编译器在对源代码进行词法语法分析以及类型检查之后,会为证明源代码满足所要求的性质生成一些验证条件(Verification Condition),并且证明这些验证条件,其中不能自动证明的部分由程序设计者利用证明工具交互地完成证明。最后,编译器在把源程序翻译成目标代码的同时,将源程序满足程序性质的证明翻译成目标代码满足等效程序性质的证明。在我们实现的原型系统中,将

基金项目:国家自然科学基金(the National Natural Science Foundation of China under Grant No.60673126);Intel 中国研究中心资助。
作者简介:刘诚(1984-),男,硕士生,主要研究方向:现代编译技术;陈意云(1946-),男,教授,主要研究方向:程序设计语言理论和实现技术、形式化描述技术、软件安全等;葛琳(1979-),女,博士生,主要研究领域为程序验证、软件安全、类型系统和理论;华保健(1979-),男,博士生,主要研究领域为程序验证、类型系统、软件安全。

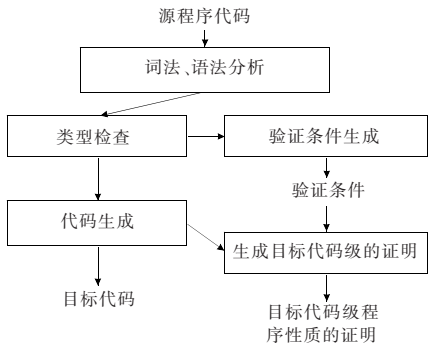

图 1 原型系统工作流程

用于生成验证条件,生成目标代码和目标代码级证明的模块相应地命名为验证条件生成器、代码生成器和证明生成器。

3 源语言程序性质的证明

在设计并实现的出具证明编译器原型系统中,源语言程序性质包括了与指针类型数据安全相关的性质以及和整数类型相关的性质。其中与指针类型数据相关的验证条件的证明将通过指针逻辑来完成,而整数类型相关的验证条件的证明将使用定理证明辅助工具 Coq[7,8]由程序设计者交互地完成。在余下的部分里将介绍与整数类型相关的程序性质的证明,与指针类型相关的程序性质的证明请参看[9]。

3.1 一个例子

图 2 中所示的是一个使用原型系统提供的 PointerC 编写的函数,程序员要求该函数满足输入的参数 n 大于等于 0,并且这个函数的返回值应该为由 1 至 n 的累加和 $n*(n+1)/2$,这些对程序性质的要求表示为该函数的前后条件。同时,程序员也为源代码中的每个循环提供其循环不变式以更好地说明源程序的性质,它们都放置在形如/*@...*/这样的注释里。

```
1   /*@n>=0*/
2   int sum(int n)
3   {int s,i;
4     i=0;
5     s=0;
6     while(i<=n)/*@(s*2==(i-1)*i)&&(i<=n+1)*/
7     {s=s+i;
8       i=i+1;
9     }
10    return s;
11  }/*@ result*2==n*(n+1)*/
```

图 2 源程序示例

3.2 验证条件

为了证明程序设计者编写的代码满足所要求的程序性质,出具证明编译器将使用验证条件生成器生成验证条件。在考察与整数类型数据相关的程序性质时,验证条件生成器将对源程序由后向前采用 Hoare 逻辑[6]的最弱前条件演算获得各个程序点的断言和验证条件。在上节的例子中,验证条件生成器使用该函数的后条件开始演算,在遇到 while 语句时生成两个验证条件。其中一个(Loop1)是循环结束应当满足的验证条件;而另一个(Loop2)则是使用循环不变式对循环体内的代码进行演算获得,它表示了进入循环体应当满足的验证条件。然后,使用循环不变式对 while 语句之前的代码进行演算,可以获得一个函数入口处的验证条件(Entry)。这三个验证条件均以定理证明辅助工具 Coq 中引理(Lemma)的方式表示在下图 3 中。

```
Lemma Entry:forall n,n>=0->n+1>=0.
Lemma Loop1:forall s n,(s*2==(i-1)*i&&i<=n+1)->i>
n->(s*2==n*n+1).
Lemma Loop2:forall s n,(s*2==(i-1)*i&&i<=n+1)->
i<=n->(s*2==(i-1)*i&&i<=n).
```

图 3 验证条件

3.3 验证条件的证明

程序设计者将使用定理证明辅助工具 Coq 来完成验证条件的证明。Coq 提供了灵活而且强大的策略(Tactics)来辅助程序员完成证明。策略是 Coq 中内建的特殊程序。策略的应用不仅简化证明过程还帮助程序员理解证明的走向。Coq 系统内建了许多证明策略,例如,用于引入和利用前提的 intro、intros、apply、assumption 等;适合于逻辑连接词证明的 split、left、right、elim、exists 等;适用于证明自然数和整数相关的 omega。

如果验证条件能够证明,那么就说明程序设计者编写的源代码满足所输入的程序性质。在上面的例子中,若程序员编写的第 5 行代码为 $s=1$,那么生成的验证条件 Entry 将对应为如下的引理。不难发现这个引理是无法证明的,即程序员编写的源代码与要求的程序性质出现了矛盾。

Lemma Entry:forall n,$n>=0->(2=0)\wedge(n+1>=0)$

4 目标程序性质的证明

为了进行目标代码级程序性质的证明,基于 CAP 和 SCAP[4,5]设计了目标程序性质证明框架 FCAP。CAP 和 SCAP 都是在 PCC 的框架内,对汇编程序使用 Hoare 风格的程序验证来开发携带证明的汇编代码的通用框架,它们采用半自动的证明方式,实现了简化的机器模型上内存管理。

4.1 证明框架

目标代码级程序性质的证明需要依赖于特定的目标机器模型,我们选用的是一个简化的 Intel x86 机器模型,它使用的指令和操作语义的定义参考了 Intel x86 的指令集。程序在这个机器模型上由代码堆、程序状态和当前执行的代码块组成。其中代码块为函数内的标号到指令序列的映射;程序状态包含了内存和寄存器两部分。而指令序列使用了机器模型中定义的目标代码指令来表示。

基于 FCAP 这个证明框架,完成目标代码中与整数类型相关的程序性质的证明过程与 CAP 和 SCAP 相似,都按照图 4 中所示的步骤进行。首先,采用形式化的方法定义目标代码及其目标代码中各个代码块应该满足的性质;然后,以引理的方式描述不同层次的合式公式(well-formed formula)并且完成这些引理的证明。关于合式公式的证明是从代码块的合式公式开始的,代码块的合式公式保证了在当前程序的上下文中如果代码块的前条件成立并且代码块满足性质,那么代码块的后条件也成立。

4.2 一个例子

第三部分中的累加函数经过原型系统中的代码生成器编译为下图中所示的目标代码。源代码中变量 n,i,s 相应的地址分别距离桢指针 ebp 的偏移为 12,-20,-16。同时,标号为 L0 的代码块为函数 sum 的出口代码块。

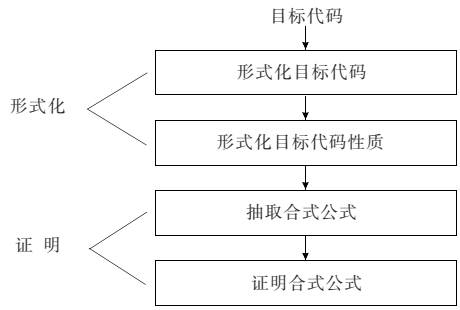

图 4　证明目标程序性质的步骤

```
.L3:  movl   $0,%eax
      movl   %eax,-20(%ebp)
      movl   $0,%eax
      movl   %eax,-16(%ebp)
.L5:  movl   -20(%ebp),%eax
      pushl  %eax
      movl   12(%ebp),%eax
      popl   %esi
      cmpl   %esi,%eax
      jge    .L1
.L4:  movl   -16(%ebp),%eax
      jmp    .L0
.L1:  movl   -20(%ebp),%eax
      addl   -16(%ebp),%eax
      movl   %eax,-16(%ebp)
      movl   $1,%eax
      addl   -20(%ebp),%eax
      movl   %eax,-20(%ebp)
      jmp    .L5
.L0:  …
```

图 5　函数 sum 编译后的目标代码

4.3　目标代码及其性质的形式化

为了进行目标代码性质的证明，目标代码需要转化为 FCAP 中相应的表示。目标代码中的每个代码块将表示为 FCAP 的机器模型中的代码块,并以指令序列(iseq)的形式来表示各条目标代码指令。下面所示的 C_L5 即为 4.2 节中代码块 L5 对应的形式,它采用了 Coq 中的定义(Definition)表示。

```
Definition C_L5:iseq:=
    seq(movld(-20)ebp eax)(seq(
    push eax)(seq(movld 12 ebp eax)(seq (
    pop esi)(seq(cmpr esi eax)(jge_L1_L4)))))
```

目标代码的性质表现为各个代码块应该满足的前后条件,也就是每个代码块入口和出口处的程序状态应该满足的性质。它们均以 FCAP 中的断言(assertion)表示。FCAP 中断言是与程序状态有关的函数。同时,由于目标代码中的各个代码块存在着承接关系,一个代码块的后条件将可以使用其后继代码块的前条件表示。

以代码块 L4 为例，这个代码块将把变量 s 的值复制到用于保存函数返回值的寄存器 eax 中,其前条件应该描述离栈指针 ebp 偏移为-16 地址上值的性质，即源程序中变量 s 的性质。该条件在 Coq 中表示为下面的定义(T_L4),其中的 getHp(s1,-16,ebp) 表示了源程序中变量 s 的值，而 getHp(s1,12,ebp)表示了距离栈指针偏移为 12 的值,即源代码中该函数参数 n 的值。而代码块 L4 的后继代码块是程序的出口代码块 L0,因此,代码块 L4 的后条件就是代码块 L0 的前条件(T_L0),它描述了保存函数返回值的寄存器 eax 应该满足的性质。

```
Definition T_L4:assert:=
fun s0 s1:state=>getHp (s1,-16,ebp)*2=(getHp(s1,12,ebp)*
(getHp(s1,12,ebp)+1)
Definition T_L0:assert:=
fun s0 s1:state=>getRf (s1,eax)*2=(getHp (s1,12,ebp)*(getHp
(s1,12,ebp)+1)
```

4.4　合式公式及其证明

FCAP 中需要证明的合式公式包括了五个层次,依次为合式代码块、合式函数体、合式函数、合式代码堆和合式程序。它们的证明是从合式代码块开始的。上面例子中代码块 L5 代码块的合式公式表示如下,其中 T_L5 为其前条件。

```
Lemma CB_L5_wf:lookupCT ct sum sum_fct ->Infer ct
sum sg T_L5 C_L5.
```

证明合式代码块的过程实际上是将代码块的前条件转换到后条件的过程。因此在证明的过程中,我们需要寻找经过每一条指令后,程序状态应该满足的性质,即寻找相邻两条指令间的断言。因此,在证明的过程中程序的布局可以表示在图 6 中。

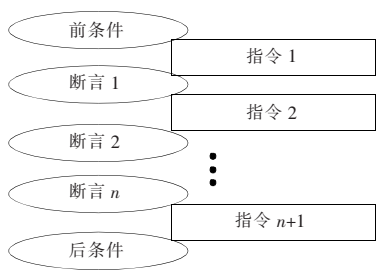

图 6　程序的布局

合式公式的证明也将使用 Coq 中提供的策略。在证明合式代码块的引理时,通过诸如 apply 这样的策略来应用各条指令间的断言,逐步将前条件转化为后条件,从而完成证明。

5　代码和证明的生成

在实现的出具证明编译器原型系统中,代码生成器和证明生成器将完成源代码到目标代码的编译以及生成关于目标代码性质的证明。本节中将先描述代码生成器和证明生成器各自的功能,然后介绍证明生成器中断言的翻译以及证明的生成。

5.1　证明生成器和代码生成器

证明生成器用于构造关于目标程序性质的证明。基于上文的描述,证明生成器生成的关于目标程序性质的证明主要包括三个部分:目标代码的形式描述、合式公式和它们各自的证明。同时,代码块合式公式证明的完成将依赖于代码块内每两条指令间插入的断言,这些断言又需要由源代码中的程序性质翻译并计算获得。因此证明生成器将按照图 7 中所示的步骤生成目标代码级的证明。

代码生成器主要负责完成将源代码编译为目标代码的工作。在我们实现的原型系统中,代码生成器的输入是编译器前端使用的符号表和经过类型检查后获得的抽象语法树结构。与传统编译器的抽象语法树不同,该抽象语法树上具有一些和程序员编写的函数前后条件,循环不变式等表示程序性质相匹配的节点。随后,代码生成器会经过若干个步骤生成目标代码在

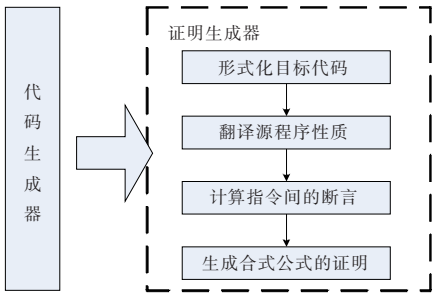

图 7 证明生成器的工作步骤

编译器内部的表示,这种表示将输出为最后的目标代码,也将交给证明生成器用于形式化目标代码以及计算指令间的断言。

在实现的原型系统中,代码生成器具有掌握代码编译过程中所有信息的能力,因此它能够为证明生成器提供各种所需要的信息以辅助证明的生成。代码生成器为证明生成器提供的信息包括:

(1)源程序代码中变量名与实际机器中地址的映射关系:它由编译器前端的符号表转换得到。这个映射关系将用于翻译与源代码相关的程序性质(函数的前后条件,循环不变式等)。特别在实现的出具证明编译器中,这部分信息还将为目标代码上基于指针逻辑的检查准备好必要的类型信息;

(2)目标代码中代码块信息:在证明目标代码性质的过程中,代码生成器需要为证明生成器提供源程序中的程序性质与所生成目标代码中代码块的对应关系;

(3)与函数调用相关的信息:在完成与函数调用相关的程序性质证明时,证明生成器需要获得被调用函数的前后条件和调用时刻的活动记录栈状态。

5.2 源程序性质的翻译和断言的生成

证明生成器从代码生成器获得编译后的目标代码的同时,也需要获得目标代码的性质。表示这些性质的目标代码块的前后断言将由证明生成器对源语言中的程序性质进行翻译得到,翻译的过程包括了转化和定位两个步骤。

首先,证明生成器会将源程序中的程序性质进行转化,这个过程根据代码生成器提供的符号表信息,把源语言级程序性质中的变量名替换为实际机器中的内存地址或者寄存器名。例如,前文中函数 sum 的后条件里的变量名 *n* 将被栈上的地址代替,而函数的返回值则被替换为寄存器 eax;

然后,第一个步骤中获得断言将作为一些目标代码块的前后条件。在上文的例子中,源程序的前后条件将对应于代码块 L3 的前条件和代码块 L4 的后条件。源代码中的循环不变式也将作为代码块 L5 的前条件。同时,证明生成器还可以根据代码块之间的承接关系来填充一些代码块的前后条件。例如上文例子中代码块 L3 的后条件即为代码块 L5 的前条件。

在完成源程序性质的翻译后,证明生成器将计算每个代码块内指令间的断言。在实现的证明生成器中,用于计算指令间断言的模块所接受的输入为指令和该指令的后断言,而后由该指令在 FCAP 中的操作语义获得前断言。因此,指令间断言是从每个代码块的后条件开始沿着指令序列由后向前计算获得。同时,对于在翻译过程结束后仍旧缺少的代码块前后条件,证明生成器将使用计算获得的断言来填充。以上文例子中的代码块 L4 为例,根据 mov 语句的操作语义,证明生成器就可以获得 mov 语句的前断言并将其作为该代码块的前条件(T_L4)。

5.3 证明的生成

证明生成器在完成翻译源语言的程序性质以及计算指令间的断言之后,已经能够将目标程序组织为图 6 中所示的程序布局。依次使用 apply 策略使用那些计算获得的断言,证明生成器就可以生成与下面类似的用于证明合式代码块的策略序列。

```
intros F0.
unfold C_L5.
apply I_SEQ with (fct:=sum_fct) (p':=_A4_L5) .
apply F0.
apply I_SEQ with (fct:=sum_fct) (p':=_A3_L5) .
…
apply lemma_L5.
apply H15.
Qed.
```

对于那些前后断言由证明生成器计算获得的代码块,它们的合式代码块的证明将比较简单。但是,在一些代码块的入口处(例如计算循环的条件表达式的代码块),证明合式代码块的过程中证明生成器还需要应用额外的引理,这些引理表示了由源代码中的程序性质翻译得到的断言可以蕴含在该程序点由计算获得的断言。下面所示的 Lemma_L5 就是这样的一条引理,它在证明代码块 L5 的合式公式时使用,T_L5 是由循环不变式翻译获得的断言,_A5_L5 则为由计算获得的代码块 L5 出口处的断言。

```
Lemma lemma_L5:forall S0 S,(T_L5 S0 S)->(_A5_L5 S0 S).
```

对于整数类型的数据,由于指令间断言和源代码中验证条件都是采用了从后向前的方式计算获得,因此这些额外的引理其实是与源语言级的验证条件相对应的,并且它们的证明也将借助于那些验证条件。上面所示的 lemma_L5 中 _A5_L5 展开后形如$(i=n\Rightarrow T_L1)\wedge(i<n\Rightarrow T_L1)\wedge(i>n\Rightarrow T_L4)$,其中的 T_L1,T_L4 分别代表代码块 C_L1 和 C_L4 的前条件。证明生成器利用已经由程序员证明的验证条件 Loop1 和 Loop2 就生成了该引理的证明。

6 总结

本文介绍了一个出具证明编译器原型系统的实现,包括源语言级与整数类型数据有关的程序性质验证条件的生成、代码生成器与证明生成器的接口以及目标代码级证明的生成。在实现这个原型系统的过程中,还发现诸如断言的化简,目标代码的优化等问题有待于在以后的研究工作中解决,因此本文的工作将为出具证明的编译器进一步向实用性迈进打下基础。

(收稿日期:2007 年 1 月)

参考文献:

[1] Necula G.Compiling with proofs[D].School of Computer Science, Carnegie Mellon Univ,1998.

[2] Necula G C,Lee P.The design and implementation of a certifying compiler[C]//Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation,1998:333-344.

[3] Colby C,Lee P,Necula G C,et al.A certifying compiler for Java[C]//Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation,2000:95-107.

[4] Yu Dachuan,Hamid N A,Shao Zhong.Building certified libraries

(下转 114 页)