汇编器的形式化验证研究

化学化工学院

F1911001

姚栋文

摘要

软件的可靠性和可信性越来越受到人们的关注,而编译器作为软件开发的基础,其正确性的验证一直都是个重要且迫切的问题。汇编器负责将汇编语言转换为二进制机器码。它作为编译器的关键组成部分，其正确运行是保证源程序正确执行的基本前提。然而，传统的编译器形式化验证（如著名的CompCert）往往终止于汇编语言层面，极少考虑汇编器的正常运行。Coq是一个正式的证明管理系统。它提供了一种形式语言来编写数学定义、可执行算法和定理，以及用于机器检查证明的半交互式开发的环境。我们通过对《software foundations》和Xavier Leroy在2017年DeepSpec暑期学校的演讲的学习和完成其附带项目，来熟悉使用Coq证明助理在汇编器方面的实现及验证，并且使用Coq完成了项目中CompCertElf和CAV21中对instruction定义的转化和一致性证明。

关键词：Coq，形式化验证，汇编器

Abstract

The reliability and trustworthiness of software have been paid more and more attention by people. As the foundation of software development, the verification of the correctness of the compiler has always been an important and urgent issue. The assembler is responsible for converting assembly language into binary machine code. As a key component of the compiler, its correct operation is the basic premise to ensure the correct execution of the source program. However, traditional compiler formal verification (such as the well-known CompCert) often ends at the assembly language level, with little regard for the normal operation of the assembler. Coq is a formal proof management system. It provides a formal language for writing mathematical definitions, executable algorithms and theorems, and an environment for semi-interactive development of machine-checked proofs. Our main work is to prove the correctness of the core assembly process using verification tools. We are familiar with the implementation and verification of the assembler using the Coq proof assistant by studying "software foundations" and Xavier Leroy's speech at the 2017 DeepSpec summer school and completing its accompanying projects, and using Coq to complete the CompCertElf and Transformation and Consistency Proof of Instruction Definitions in CAV21.

**KEY WORDS**: Coq, formal verification, assembler

第一章 绪论

1.1 引言

随着社会的发展，计算机软件已经成为了社会生产发展过程中不可或缺的一部分。对于电信、能源和金融等安全性关键领域来说，软件的一个小小缺陷、失效或者崩溃可能会危害到数千万人的生命和财产安全。如何设计安全可靠的软件系统，是当前迫切且重要的问题之一。而其中的一个核心问题便是如何利用严格精确的形式化方法对程序进行验证以及程序正确性的自动推导与证明[1]。基于定理证明器 Coq 对各种基础软件的语义进行形式化描述，并对其重要属性进行正确性的证明是当前的一个研究热点，是提高基础软件质量的一个切实可行的研究方向。通过将语义机械化，可以构造出机器可验证的、完全正确的编译器、解释器、验证器等重要的基础软件。确保基础软件的完全正确，其意义和重要性是不言而喻的[2]。然而在 Coq 中构造实际可用的编译器需要花费巨大的精力，证明代码量大[2]。而使用程序验证器接受经过规范标注的源代码生成验证条件,并将验证条件交给约束求解器自动求解,这种方式自动化程度较高[3]。本项目研究如何使用形式化方法构建极高可靠度的汇编器(Assembler)，这对于推动编译器形式化验证的实用化有重要意义。我们的主要工作是使用验证工具证明核心汇编过程的正确性。在指导老师已有成果的基础之上，利用CAV可以自动生成证明，我们通过证明CAV和CompCert的指令可以互相转换，以此来实现CompCert的自动生成证明。

构建可靠的软件非常地困难。现代系统的规模、复杂度、参与构建过程的人数，还有置于系统之上的需求范围，让构建或多或少地正确的软件变得极为困难， 更不用说百分之百地正确了。同时，由于信息处理技术继续渗透到社会的各个层面，人们为程序错误和漏洞付出的代价变得越来越高昂。为了应对这些挑战，计算机科学家和软件工程师们发展了一套完整的提升软件质量的方法，从为管理软件项目的团队提供建议（如极限编程，Extreme Programming）， 到库的设计原理（如模型-视图-控制器，Model-View-Controller；发布-订阅模式，Publish-Subscribe）以及编程语言的设计哲学（面向对象编程，Object Oriented Programmin；面向剖面编程，Aspect Oriented Programming；函数式编程，Functional Programming），还有用于阐明和论证软件性质的数学技术，以及验证这些性质的工具。

1.2 软件基础（Software Foundations）介绍

《软件基础》着重于上述的最后一种方法。

本书将以下三种概念穿插在一起：（1）逻辑学中的基本工具，用于准确地提出并论证关于程序的假设；（2）证明助理用于构造严谨的逻辑论据；（3）函数式编程思想，同时作为一种编程方法来简化程序的论证，以及架起程序和逻辑学之间的桥梁。

1.3 Coq

证明助理是一种混合式工具，它能将证明的构建中比较常规的部分自动化， 而更加困难的部分则依赖人类来解决。广泛使用的证明助理包括Isabelle、Agda、Twelf、ACL2、PVS 以及 Coq 等等。

Coq 为机器验证的形式化论证的交互式开发提供了丰富的环境。Coq 系统的内核是一个简单的证明检查器，它保证只会执行正确的推理步骤。 在此内核之上，Coq 环境提供了高级的证明开发功能，包括一个庞大的库， 其中包含各种定义和引理；强大策略，用于半自动化构造证明； 还有一个专用的编程语言，能够为特殊情况定义新的自动证明策略。Coq 已成为跨计算机科学和数学研究的关键推动者：

作为一个编程语言的建模平台， Coq 成为了研究员对复杂的语言定义进行描述和论证的标准工具。 例如，它被用来检查 JavaCard 平台的安全性，得到了最高等级的通用准则验证， 它还被用在 x86 和 LLVM 指令集以及 C 等编程语言的形式化规范中。

作为一个形式化软件验证的开发环境，Coq 被用来构建：CompCert，一个完全验证的C优化编译器；CertiKos，一个完全验证的工具，用于证明涉及浮点数的精妙算法的正确性；Coq 也是CertiCrypt的基础，一个用于论证密码学算法安全性的环境。Coq还被用来构建开源 RISC-V 处理器架构的验证实现。

作为一个依赖类型函数式编程的现实环境，Coq激发了大量的创新。例如Ynot系统嵌入了关系式霍尔推理（一个霍尔逻辑的扩展）。

作为一个高阶逻辑的证明助理，Coq 被用来验证数学中一些重要的结果。例如Coq可在证明中包含复杂计算的能力，使其开发出了第一个形式化验证的四色定理证明。此前数学家们对该证明颇有争议，因为它需要用程序对大量组态进行检验。在Coq的形式化中，所有东西都被检验过，自然也包括计算的正确性。近年来，Feit-Thompson定理经过了更大的努力用Coq形式化了，它是对有限单群进行分类的十分重要的第一步。

第二章 《软件基础》的学习

2.1 引言

本书的主题包括基本的逻辑概念、计算机辅助定理证明、Coq 证明助理、函数式编程、操作语义、用于论证软件的逻辑和技术、静态类型系统、基于性质的随机测试、以及对实践中 C 代码的验证。

函数式编程风格建立在简单的、日常的数学直觉之上：若一个过程或方法没有副作用，那么在忽略效率的前提下，我们需要理解的一切便只剩下它如何将输入映射到输出了——也就是说，我们只需将它视作一种计算数学函数的具体方法即可。这也是 “函数式编程”中“函数式”一词的含义之一。程序与简单数学对象之间这种直接的联系，时支撑了对程序行为进行形式化证明的正确性以及非形式化论证的可靠性。

函数式编程中“函数式”一词的另一个含义是它强调把函数作为一等的值——这类值可以作为参数传递给其它函数，可以作为结果返回，也可以包含在数据结构中等等。这种将函数当做数据的方式，产生了大量强大而有用的编程习语。

其它常见的函数式语言特性包括能让构造和处理丰富数据结构更加简单的代数数据类型（Algebraic Data Type）和模式匹配（Pattern Matching），以及用来支持抽象和代码复用的'多态类型系统（Polymorphic Type System）。Coq 提供了所有这些特性。

2.2 学习路线

在《软件基础》中我们依次学到了Coq中类型的定义，归纳法证明，使用结构化的数据，多态与高阶函数，Coq中的逻辑系统，归纳定义的命题。进一步，我们掌握了全映射和偏映射，柯里-霍华德同构，二元关系的性质。最终，我们实现了简单的指令式程序，用Coq实现了词法分析和语法分析，并掌握了自动证明技术和从Coq中抽取ML的方法。从而为后续的项目打下理论基础。

在书中指定了形式化的算术和布尔表达式的抽象语法（Abstract Syntax）。

Inductive aexp : Type :=  
  | ANum (n : [nat](http://coq.inria.fr/library//Coq.Init.Datatypes.html" \l "nat))  
  | APlus (a1 a2 : [aexp](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "aexp))  
  | AMinus (a1 a2 : [aexp](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "aexp))  
  | AMult (a1 a2 : [aexp](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "aexp)).

Inductive bexp : Type :=  
  | BTrue  
  | BFalse  
  | BEq (a1 a2 : [aexp](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "AExp.aexp))  
  | BLe (a1 a2 : [aexp](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "AExp.aexp))  
  | BNot (b : [bexp](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "bexp))  
  | BAnd (b1 b2 : [bexp](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "bexp)).

作为对比，下面是用约定的 BNF（巴克斯-诺尔范式）文法定义的同样的抽象语法：

a ::= nat  
        | a + a  
        | a - a  
        | a × a

  b ::= true  
        | false  
        | a = a  
        | a ≤ a  
        | ¬b  
        | b && b

我们可以对算术表达式进行'求值（Evaluation）并得到数值。

Fixpoint aeval (a : [aexp](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "AExp.aexp)) : [nat](http://coq.inria.fr/library//Coq.Init.Datatypes.html" \l "nat) :=  
  match [a](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "a) with  
  | [ANum](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "AExp.ANum) n ⇒ n  
  | [APlus](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "AExp.APlus) a1 a2 ⇒ [(](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "b3eea360671e1b32b18a26e15b3aace3)[aeval](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "aeval) a1[)](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "b3eea360671e1b32b18a26e15b3aace3) [+](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "b3eea360671e1b32b18a26e15b3aace3) [(](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "b3eea360671e1b32b18a26e15b3aace3)[aeval](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "aeval) a2[)](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "b3eea360671e1b32b18a26e15b3aace3)  
  | [AMinus](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "AExp.AMinus) a1 a2 ⇒ [(](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "9482aae3d3b06e249765c1225dbb8cbb)[aeval](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "aeval) a1[)](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "9482aae3d3b06e249765c1225dbb8cbb) [-](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "9482aae3d3b06e249765c1225dbb8cbb) [(](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "9482aae3d3b06e249765c1225dbb8cbb)[aeval](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "aeval) a2[)](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "9482aae3d3b06e249765c1225dbb8cbb)  
  | [AMult](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "AExp.AMult) a1 a2 ⇒ [(](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "697e4695610f677ae98a52af81f779d<sub>2</sub>)[aeval](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "aeval) a1[)](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "697e4695610f677ae98a52af81f779d<sub>2</sub>) [×](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "697e4695610f677ae98a52af81f779d<sub>2</sub>) [(](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "697e4695610f677ae98a52af81f779d<sub>2</sub>)[aeval](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "aeval) a2[)](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "697e4695610f677ae98a52af81f779d<sub>2</sub>)  
  end.

Example test\_aeval1:  
  [aeval](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "AExp.aeval) ([APlus](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "AExp.APlus) ([ANum](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "AExp.ANum) 2) ([ANum](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "AExp.ANum) 2)) [=](http://coq.inria.fr/library//Coq.Init.Logic.html" \l "1c39bf18749e5cc609e83c0a0ba5a372) 4.

Proof. reflexivity. Qed.

同样，对布尔表达式求值会得到布尔值。  
Fixpoint beval (b : [bexp](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "AExp.bexp)) : [bool](http://coq.inria.fr/library//Coq.Init.Datatypes.html" \l "bool) :=  
  match [b](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "b) with  
  | [BTrue](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "AExp.BTrue) ⇒ [true](http://coq.inria.fr/library//Coq.Init.Datatypes.html" \l "true)  
  | [BFalse](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "AExp.BFalse) ⇒ [false](http://coq.inria.fr/library//Coq.Init.Datatypes.html" \l "false)  
  | [BEq](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "AExp.BEq) a1 a2 ⇒ [(](http://coq.inria.fr/library//Coq.Init.Nat.html" \l "97a8d8c92e88d1d68fb55a13a4fcfc6d)[aeval](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "AExp.aeval) a1[)](http://coq.inria.fr/library//Coq.Init.Nat.html" \l "97a8d8c92e88d1d68fb55a13a4fcfc6d) [=?](http://coq.inria.fr/library//Coq.Init.Nat.html" \l "97a8d8c92e88d1d68fb55a13a4fcfc6d) [(](http://coq.inria.fr/library//Coq.Init.Nat.html" \l "97a8d8c92e88d1d68fb55a13a4fcfc6d)[aeval](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "AExp.aeval) a2[)](http://coq.inria.fr/library//Coq.Init.Nat.html" \l "97a8d8c92e88d1d68fb55a13a4fcfc6d)  
  | [BLe](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "AExp.BLe) a1 a2 ⇒ [(](http://coq.inria.fr/library//Coq.Init.Nat.html" \l "01f84f7b4db7ca25b5fbdf8f04a7bebc)[aeval](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "AExp.aeval) a1[)](http://coq.inria.fr/library//Coq.Init.Nat.html" \l "01f84f7b4db7ca25b5fbdf8f04a7bebc) [<=?](http://coq.inria.fr/library//Coq.Init.Nat.html" \l "01f84f7b4db7ca25b5fbdf8f04a7bebc) [(](http://coq.inria.fr/library//Coq.Init.Nat.html" \l "01f84f7b4db7ca25b5fbdf8f04a7bebc)[aeval](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "AExp.aeval) a2[)](http://coq.inria.fr/library//Coq.Init.Nat.html" \l "01f84f7b4db7ca25b5fbdf8f04a7bebc)  
  | [BNot](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "AExp.BNot) b1 ⇒ [negb](http://coq.inria.fr/library//Coq.Init.Datatypes.html" \l "negb) ([beval](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "beval) b1)  
  | [BAnd](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "AExp.BAnd) b1 b2 ⇒ [andb](http://coq.inria.fr/library//Coq.Init.Datatypes.html" \l "andb) ([beval](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "beval) b1) ([beval](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "beval) b2)  
  end.

接下来我们要为算术和布尔表达式加上变量。'机器的状态'（简称'状态'）表示程序执行中某一时刻'所有变量'的值。对Imp程序而言，由于每个变量都存储了一个自然数，因此我们可以将状态表示为一个从字符串到整数的映射，并且用 0 作为存储中的默认值。由于需要查找变量来获得它们的具体值，因此我们使用了[Maps](https://coq-zh.github.io/SF-zh/lf-current/Maps.html)一章中的映射。

Definition state := [total\_map](https://coq-zh.github.io/SF-zh/lf-current/Maps.html" \l "total_map) [nat](http://coq.inria.fr/library//Coq.Init.Datatypes.html" \l "nat).

Inductive aexp : Type :=  
  | ANum (n : [nat](http://coq.inria.fr/library//Coq.Init.Datatypes.html" \l "nat))  
  | AId (x : [string](http://coq.inria.fr/library//Coq.Strings.String.html" \l "string)) (\* <--- 新增 \*)  
  | APlus (a1 a2 : [aexp](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "aexp))  
  | AMinus (a1 a2 : [aexp](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "aexp))  
  | AMult (a1 a2 : [aexp](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "aexp)).

算术和布尔求值器被扩展成以很显然的方式来处理变量， 它接受一个状态作为额外的参数：

Fixpoint aeval (st : [state](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "state)) (a : [aexp](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "aexp)) : [nat](http://coq.inria.fr/library//Coq.Init.Datatypes.html" \l "nat) :=  
  match [a](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "a) with  
  | [ANum](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "ANum) n ⇒ n  
  | [AId](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "AId) x ⇒ [st](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "st) x (\* <--- 新增 \*)  
  | [APlus](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "APlus) a1 a2 ⇒ [(](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "b3eea360671e1b32b18a26e15b3aace3)[aeval](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "aeval) [st](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "st) a1[)](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "b3eea360671e1b32b18a26e15b3aace3) [+](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "b3eea360671e1b32b18a26e15b3aace3) [(](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "b3eea360671e1b32b18a26e15b3aace3)[aeval](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "aeval) [st](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "st) a2[)](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "b3eea360671e1b32b18a26e15b3aace3)  
  | [AMinus](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "AMinus) a1 a2 ⇒ [(](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "9482aae3d3b06e249765c1225dbb8cbb)[aeval](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "aeval) [st](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "st) a1[)](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "9482aae3d3b06e249765c1225dbb8cbb) [-](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "9482aae3d3b06e249765c1225dbb8cbb) [(](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "9482aae3d3b06e249765c1225dbb8cbb)[aeval](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "aeval) [st](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "st) a2[)](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "9482aae3d3b06e249765c1225dbb8cbb)  
  | [AMult](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "AMult) a1 a2 ⇒ [(](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "697e4695610f677ae98a52af81f779d<sub>2</sub>)[aeval](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "aeval) [st](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "st) a1[)](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "697e4695610f677ae98a52af81f779d<sub>2</sub>) [×](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "697e4695610f677ae98a52af81f779d<sub>2</sub>) [(](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "697e4695610f677ae98a52af81f779d<sub>2</sub>)[aeval](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "aeval) [st](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "st) a2[)](http://coq.inria.fr/library//Coq.Init.Peano.html" \l "697e4695610f677ae98a52af81f779d<sub>2</sub>)  
  end.

Fixpoint beval (st : [state](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "state)) (b : [bexp](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "bexp)) : [bool](http://coq.inria.fr/library//Coq.Init.Datatypes.html" \l "bool) :=  
  match [b](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "b) with  
  | [BTrue](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "BTrue) ⇒ [true](http://coq.inria.fr/library//Coq.Init.Datatypes.html" \l "true)  
  | [BFalse](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "BFalse) ⇒ [false](http://coq.inria.fr/library//Coq.Init.Datatypes.html" \l "false)  
  | [BEq](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "BEq) a1 a2 ⇒ [(](http://coq.inria.fr/library//Coq.Init.Nat.html" \l "97a8d8c92e88d1d68fb55a13a4fcfc6d)[aeval](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "aeval) [st](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "st) a1[)](http://coq.inria.fr/library//Coq.Init.Nat.html" \l "97a8d8c92e88d1d68fb55a13a4fcfc6d) [=?](http://coq.inria.fr/library//Coq.Init.Nat.html" \l "97a8d8c92e88d1d68fb55a13a4fcfc6d) [(](http://coq.inria.fr/library//Coq.Init.Nat.html" \l "97a8d8c92e88d1d68fb55a13a4fcfc6d)[aeval](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "aeval) [st](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "st) a2[)](http://coq.inria.fr/library//Coq.Init.Nat.html" \l "97a8d8c92e88d1d68fb55a13a4fcfc6d)  
  | [BLe](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "BLe) a1 a2 ⇒ [(](http://coq.inria.fr/library//Coq.Init.Nat.html" \l "01f84f7b4db7ca25b5fbdf8f04a7bebc)[aeval](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "aeval) [st](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "st) a1[)](http://coq.inria.fr/library//Coq.Init.Nat.html" \l "01f84f7b4db7ca25b5fbdf8f04a7bebc) [<=?](http://coq.inria.fr/library//Coq.Init.Nat.html" \l "01f84f7b4db7ca25b5fbdf8f04a7bebc) [(](http://coq.inria.fr/library//Coq.Init.Nat.html" \l "01f84f7b4db7ca25b5fbdf8f04a7bebc)[aeval](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "aeval) [st](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "st) a2[)](http://coq.inria.fr/library//Coq.Init.Nat.html" \l "01f84f7b4db7ca25b5fbdf8f04a7bebc)  
  | [BNot](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "BNot) b1 ⇒ [negb](http://coq.inria.fr/library//Coq.Init.Datatypes.html" \l "negb) ([beval](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "beval) [st](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "st) b1)  
  | [BAnd](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "BAnd) b1 b2 ⇒ [andb](http://coq.inria.fr/library//Coq.Init.Datatypes.html" \l "andb) ([beval](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "beval) [st](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "st) b1) ([beval](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "beval) [st](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "st) b2)  
  end.

我们为具体状态的全映射声明具体的记法，即使用 (\_ !-> 0) 作为空状态。

Definition empty\_st := ([\_](https://coq-zh.github.io/SF-zh/lf-current/Maps.html" \l "bc1c5e34127128228512ba20d13f5577) [!](https://coq-zh.github.io/SF-zh/lf-current/Maps.html" \l "bc1c5e34127128228512ba20d13f5577)-> 0).

现在我们可以为“单例状态（singleton state）”添加新的记法了， 即只有一个绑定到值的变量。

Notation "x '!->' v" := ([t\_update](https://coq-zh.github.io/SF-zh/lf-current/Maps.html" \l "t_update) [empty\_st](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "empty_st) x v) (at level 100).

Example aexp1 :  
    [aeval](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "aeval) ([X](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "X) [!](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "ba84b24570fb3a89fbbf23462f3909cd)-> 5) (3 [+](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "040a5752226a8185c7d6fc9125b7a4ea) [(](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "040a5752226a8185c7d6fc9125b7a4ea)[X](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "X) [×](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "ecd9390a06cb87a38cb0a2350ca063ec) 2[)](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "040a5752226a8185c7d6fc9125b7a4ea))  
  [=](http://coq.inria.fr/library//Coq.Init.Logic.html" \l "1c39bf18749e5cc609e83c0a0ba5a372) 13.

Proof. reflexivity. Qed.

现在我们可以定义 Imp '指令（Command）'（有时称作'语句（Statement）'） 的语法和行为了。

指令 c 可以用以下 BNF 文法非形式化地描述。

 c ::= SKIP | x ::= a | c ;; c | TEST b THEN c ELSE c FI  
         | WHILE b DO c END

下面是指令的抽象语法的形式化定义：  
Inductive com : Type :=  
  | CSkip  
  | CAss (x : [string](http://coq.inria.fr/library//Coq.Strings.String.html" \l "string)) (a : [aexp](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "aexp))  
  | CSeq (c1 c2 : [com](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "com))  
  | CIf (b : [bexp](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "bexp)) (c1 c2 : [com](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "com))  
  | CWhile (b : [bexp](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "bexp)) (c : [com](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "com)).

至于表达式，我们可以用一些 Notation 声明来让 Imp 程序的读写更加方便。  
Bind Scope imp\_scope with com.  
Notation "'SKIP'" :=  
   [CSkip](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "CSkip) : imp\_scope.  
Notation "x '::=' a" :=  
  ([CAss](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "CAss) x a) (at level 60) : imp\_scope.  
Notation "c1 ;; c2" :=  
  ([CSeq](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "CSeq) c1 c2) (at level 80, right associativity) : imp\_scope.  
Notation "'WHILE' b 'DO' c 'END'" :=  
  ([CWhile](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "CWhile) b c) (at level 80, right associativity) : imp\_scope.  
Notation "'TEST' c1 'THEN' c2 'ELSE' c3 'FI'" :=  
  ([CIf](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "CIf) c1 c2 c3) (at level 80, right associativity) : imp\_scope.

例如，下面是个阶乘函数，写成 Coq 的形式化定义：  
Definition fact\_in\_coq : [com](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "com) :=  
  ([Z](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "Z) [:](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "1415d381b898f10db78e714a0a964c<sub>11</sub>):= [X](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "X)[;;](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "5604146224f04ce8ff933bc02eb29824)  
  [Y](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "Y) [:](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "1415d381b898f10db78e714a0a964c<sub>11</sub>):= 1[;;](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "5604146224f04ce8ff933bc02eb29824)  
  [WHILE](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "83a07334581e0deb166d05032f77769f) [~(](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "e5a2f8c4cfacb82c023ce3487633feff)[Z](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "Z) [=](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "102d8c3ecbd7653efa317bf9b975bd1a) 0[)](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "e5a2f8c4cfacb82c023ce3487633feff) [DO](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "83a07334581e0deb166d05032f77769f)  
    [Y](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "Y) [:](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "1415d381b898f10db78e714a0a964c<sub>11</sub>):= [Y](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "Y) [×](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "ecd9390a06cb87a38cb0a2350ca063ec) [Z](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "Z)[;;](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "5604146224f04ce8ff933bc02eb29824)  
    [Z](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "Z) [:](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "1415d381b898f10db78e714a0a964c<sub>11</sub>):= [Z](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "Z) [-](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "bec100dd937ffc06b7e2521070b6294c) 1  
  [END](https://coq-zh.github.io/SF-zh/lf-current/Imp.html" \l "83a07334581e0deb166d05032f77769f)).

关于它的求值，我们知道Coq 不仅是一个函数式编程语言，还拥有逻辑一致性， 因此任何有可能不会停机的函数都会被拒绝。下面是一段无效的(!) Coq 程序，它展示了假如 Coq 允许不停机的递归函数时会产生什么错误：

Fixpoint loop\_false (n : nat) : False := loop\_false n.

也就是说，像 False 这样的命题会变成可证的（例如 loop\_false 0 就是个对 False 的证明），这对 Coq 的逻辑一致性来说是一场灾难。因此我们需要的技巧是将一个'附加'的参数传入求值函数中来告诉它需要运行多久。非正式地说，我们会在求值器的“油箱”中加一定数量的“汽油”，然后允许它运行到按一般的方式终止'或者'耗尽汽油，此时我们会停止求值并说最终结果为空内存（empty memory）。

第三章 编译器形式验证研究

3.1 项目介绍

本项目定义了从Imp语言到虚拟机的编译器（Java虚拟机的一个小子集）并证明了这个编译器保留源程序的语义。我们的目标是通过与讲座一起进行的Coq项目，熟悉编译器验证的基本概念。

3.2 虚拟机部分

机器操作代码 [c] (固定的指令列表) 和三个可变组件：

- 程序计数器，表示 [c] 中的位置

- 将整数值分配给变量的状态

- 一个评估堆栈，包含整数。

机器的指令集

Inductive instruction: Type :=

| Iconst(n: nat) (\*\*r push integer [n] on stack \*)

| Ivar(x: id) (\*\*r push the value of variable [x] \*)

| Isetvar(x: id) (\*\*r pop an integer, assign it to variable [x] \*)

| Iadd (\*\*r pop [n2], pop [n1], push back [n1+n2] \*)

| Isub (\*\*r pop [n2], pop [n1], push back [n1-n2] \*)

| Imul (\*\*r pop [n2], pop [n1], push back [n1\*n2] \*)

| Ibranch\_forward(ofs: nat) (\*\*r skip [ofs] instructions forward \*)

| Ibranch\_backward(ofs: nat) (\*\*r skip [ofs] instructions backward \*)

| Ibeq(ofs: nat) (\*\*r pop [n2], pop [n1], skip [ofs] forward if [n1=n2] \*)

| Ibne(ofs: nat) (\*\*r pop [n2], pop [n1], skip [ofs] forward if [n1<>n2] \*)

| Ible(ofs: nat) (\*\*r pop [n2], pop [n1], skip [ofs] forward if [n1<=n2] \*)

| Ibgt(ofs: nat) (\*\*r pop [n2], pop [n1], skip [ofs] forward if [n1>n2] \*)

| Ihalt. (\*\*r terminate execution successfully \*)

并且定义指令的列表（代码）

Definition code := list instruction.

为了得到代码中当前执行位置的具体指令，我们定义code\_at函数

Fixpoint code\_at (C: code) (pc: nat) : option instruction :=

match C, pc with

| nil, \_ => None

| i :: C', O => Some i

| i :: C', S pc' => code\_at C' pc'

end.

并且定义栈为一个自然数列表

Definition stack := list nat.

在小步语义中，状态 [st] 中的命令 [c] 的语义通过形成从 [c, st] 开始的递减序列来确定。

- 有限序列：零、一个或几个约简（自反传递闭包）

- 无限序列：无限多的减少。

首先虚拟机的语义以小步语义的方式给出， 作为机器配置之间的过渡关系：三元组 （程序计数器、评估堆栈、变量状态）。

Definition configuration := (nat \* stack \* state)%type.

转换关系由代码 [c] 参数化。 每种指令都有一个转换规则， 除了 [Ihalt]，它没有转换。

Inductive transition (C: code): configuration -> configuration -> Prop :=

| trans\_const: forall pc stk s n,

code\_at C pc = Some(Iconst n) ->

transition C (pc, stk, s) (pc + 1, n :: stk, s)

| trans\_var: forall pc stk s x,

code\_at C pc = Some(Ivar x) ->

transition C (pc, stk, s) (pc + 1, s x :: stk, s)

| trans\_setvar: forall pc stk s x n,

code\_at C pc = Some(Isetvar x) ->

transition C (pc, n :: stk, s) (pc + 1, stk, t\_update s x n)

| trans\_add: forall pc stk s n1 n2,

code\_at C pc = Some(Iadd) ->

transition C (pc, n2 :: n1 :: stk, s) (pc + 1, (n1 + n2) :: stk, s)

| trans\_sub: forall pc stk s n1 n2,

code\_at C pc = Some(Isub) ->

transition C (pc, n2 :: n1 :: stk, s) (pc + 1, (n1 - n2) :: stk, s)

| trans\_mul: forall pc stk s n1 n2,

code\_at C pc = Some(Imul) ->

transition C (pc, n2 :: n1 :: stk, s) (pc + 1, (n1 \* n2) :: stk, s)

| trans\_branch\_forward: forall pc stk s ofs pc',

code\_at C pc = Some(Ibranch\_forward ofs) ->

pc' = pc + 1 + ofs ->

transition C (pc, stk, s) (pc', stk, s)

| trans\_branch\_backward: forall pc stk s ofs pc',

code\_at C pc = Some(Ibranch\_backward ofs) ->

pc' = pc + 1 - ofs ->

transition C (pc, stk, s) (pc', stk, s)

| trans\_beq: forall pc stk s ofs n1 n2 pc',

code\_at C pc = Some(Ibeq ofs) ->

pc' = (if beq\_nat n1 n2 then pc + 1 + ofs else pc + 1) ->

transition C (pc, n2 :: n1 :: stk, s) (pc', stk, s)

| trans\_bne: forall pc stk s ofs n1 n2 pc',

code\_at C pc = Some(Ibne ofs) ->

pc' = (if beq\_nat n1 n2 then pc + 1 else pc + 1 + ofs) ->

transition C (pc, n2 :: n1 :: stk, s) (pc', stk, s)

| trans\_ble: forall pc stk s ofs n1 n2 pc',

code\_at C pc = Some(Ible ofs) ->

pc' = (if leb n1 n2 then pc + 1 + ofs else pc + 1) ->

transition C (pc, n2 :: n1 :: stk, s) (pc', stk, s)

| trans\_bgt: forall pc stk s ofs n1 n2 pc',

code\_at C pc = Some(Ibgt ofs) ->

pc' = (if leb n1 n2 then pc + 1 else pc + 1 + ofs) ->

transition C (pc, n2 :: n1 :: stk, s) (pc', stk, s).

使用小步语义，我们形成机器转换序列来定义代码的行为。 我们总是从 [pc = 0] 开始和一个空的评估堆栈。 会得到三种情况：

-如果 [pc] 指向[Ihalt] 指令并且评估堆栈为空，则我们成功停止

-如果我们从起始状态能得到一个无限递减的代码，则定义为发散

-如果执行到最后一步，[pc]没有指向[Ihalt]指令或者评估堆栈不为空，则程序错误

下面为相应的代码

Definition mach\_terminates (C: code) (s\_init s\_fin: state) :=

exists pc,

code\_at C pc = Some Ihalt /\

star (transition C) (0, nil, s\_init) (pc, nil, s\_fin).

如果 [R] 是二元关系，则 [star R] 是它的自反传递闭包。

Definition mach\_diverges (C: code) (s\_init: state) :=

infseq (transition C) (0, nil, s\_init).

同样，[infseq R] 表示 [R] 转换的无限序列。

Definition mach\_goes\_wrong (C: code) (s\_init: state) :=

exists pc, exists stk, exists s\_fin,

star (transition C) (0, nil, s\_init) (pc, stk, s\_fin)

/\ irred (transition C) (pc, stk, s\_fin)

/\ (code\_at C pc <> Some Ihalt \/ stk <> nil).

我们可以很容易地证明虚拟机的性质代码具有确定性。即从给定的配置，它最多可以转换到其他配置。

Lemma machine\_deterministic:

forall C config config1 config2,

transition C config config1 -> transition C config config2 -> config1 = config2.

作为这种决定论的结果，可以得出以下结论： 终止程序的最终状态是唯一的， 并且一个程序不能既终止又发散， 或终止并出错，或分歧并出错。

3.3 编译方案

算术表达式的代码 [a]

- 按顺序执行（无分支）

- 将 [a] 的值存放在栈顶 -

保留变量状态。

Fixpoint compile\_aexp (a: aexp) : code :=

match a with

| ANum n => Iconst n :: nil

| AId v => Ivar v :: nil

| APlus a1 a2 => compile\_aexp a1 ++ compile\_aexp a2 ++ Iadd :: nil

| AMinus a1 a2 => compile\_aexp a1 ++ compile\_aexp a2 ++ Isub :: nil

| AMult a1 a2 => compile\_aexp a1 ++ compile\_aexp a2 ++ Imul :: nil

end.

布尔表达式 [b] 的代码 [compile\_bexp b cond ofs]

- 如果 [b] 计算结果为 [cond]（布尔值），则向前跳过 [ofs] 以下指令 -

如果 [b] 求值为 [cond] 的否定，则按顺序执行

- 保持堆栈和变量状态不变。

Fixpoint compile\_bexp (b: bexp) (cond: bool) (ofs: nat) : code :=

match b with

| BTrue =>

if cond then Ibranch\_forward ofs :: nil else nil

| BFalse =>

if cond then nil else Ibranch\_forward ofs :: nil

| BEq a1 a2 =>

compile\_aexp a1 ++ compile\_aexp a2 ++

(if cond then Ibeq ofs :: nil else Ibne ofs :: nil)

| BLe a1 a2 =>

compile\_aexp a1 ++ compile\_aexp a2 ++

(if cond then Ible ofs :: nil else Ibgt ofs :: nil)

| BNot b1 =>

compile\_bexp b1 (negb cond) ofs

| BAnd b1 b2 =>

let c2 := compile\_bexp b2 cond ofs in

let c1 := compile\_bexp b1 false (if cond then length c2 else ofs + length c2) in

c1 ++ c2

end.

命令的代码 [c]

- 按照 [c] 的规定更新变量状态

- 保留堆栈

- 在生成的代码之后立即完成下一条指令。

Fixpoint compile\_com (c: com) : code :=

match c with

| SKIP =>

nil

| (id ::= a) =>

compile\_aexp a ++ Isetvar id :: nil

| (c1 ;; c2) =>

compile\_com c1 ++ compile\_com c2

| IFB b THEN ifso ELSE ifnot FI =>

let code\_ifso := compile\_com ifso in

let code\_ifnot := compile\_com ifnot in

compile\_bexp b false (length code\_ifso + 1)

++ code\_ifso

++ Ibranch\_forward (length code\_ifnot)

:: code\_ifnot

| WHILE b DO body END =>

let code\_body := compile\_com body in

let code\_test := compile\_bexp b false (length code\_body + 1) in

code\_test

++ code\_body

++ Ibranch\_backward (length code\_test + length code\_body + 1)

:: nil

end.

现在我们编译一个程序只需要在其后加上一条Ihat指令

Definition compile\_program (p: com) : code :=

compile\_com p ++ Ihalt :: nil.

3.4 语义保护

关于代码序列的辅助结果。

要推理编译代码的执行，我们需要考虑更大代码中位于 [pc] 位置的代码序列 [C2] 序列 [C = C1 ++ C2 ++ C3]。 以下谓词 [codeseq\_at C pc C2] 就是这样做的。

Inductive codeseq\_at: code -> nat -> code -> Prop :=

| codeseq\_at\_intro: forall C1 C2 C3 pc,

pc = length C1 ->

codeseq\_at (C1 ++ C2 ++ C3) pc C2.

生成的表达式代码的正确性。

对于我们为生成的代码提供的非正式规范对于算术表达式 [a]。 应该

- 按顺序执行（无分支）

- 将 [a] 的值存入栈顶

-保留变量状态。

我们可以证明代码 [compile\_aexp a] 履行了这个契约。

在《软件基础》中我们定义了大步语义，而小步语义，呈现为一步减少关系 [ c / st --> c' / st' ]，表示命令 [c]， 在初始状态 [st'] 中执行，并执行一个基本的计算步骤。 [st'] 是此步骤之后的更新状态。 [c'] 是剩余命令，捕获所有仍有待完成的计算。 其中不仅命令的执行在各个步骤中被分解， 还有对算术和布尔表达式的求值。 我们仍然处理求值，从而脱离了这种语义的算术和布尔表达式作为一个原子“大步” 。 表达式的非原子求值在多个命令的并行（交错）执行时会有所不同。在纯粹的顺序执行中，它等效（并且更简单） 在一个原子步骤中求值表达式，因为它们的求值总是终止。

为加强生成表达式代码的正确性结果，使其不仅限于终止的源程序，同时也适用于发散的源程序。我们放弃大步命令的语义并切换到带有延续的小步语义。

Inductive compile\_cont (C: code): cont -> nat -> Prop :=

| ccont\_stop: forall pc,

code\_at C pc = Some Ihalt ->

compile\_cont C Kstop pc

| ccont\_seq: forall c k pc pc',

codeseq\_at C pc (compile\_com c) ->

pc' = pc + length (compile\_com c) ->

compile\_cont C k pc' ->

compile\_cont C (Kseq c k) pc

| ccont\_while: forall b c k pc ofs pc' pc'',

code\_at C pc = Some(Ibranch\_backward ofs) ->

pc' = pc + 1 - ofs ->

codeseq\_at C pc' (compile\_com (WHILE b DO c END)) ->

pc'' = pc' + length (compile\_com (WHILE b DO c END)) ->

compile\_cont C k pc'' ->

compile\_cont C (Kwhile b c k) pc

| ccont\_branch: forall ofs k pc pc',

code\_at C pc = Some(Ibranch\_forward ofs) ->

pc' = pc + 1 + ofs ->

compile\_cont C k pc' ->

compile\_cont C k pc.

小步语义的配置 [(c,k,st)] 匹配机器的配置 [(C, pc, stk, st')], 如果满足以下条件：

-内存状态相同：[st' = st]。

- 机器堆栈为空：[stk = nil]。

- [pc] 点的机器代码是 [c] 的编译代码： [codeseq\_at C pc (compile\_com c)]。

- 点 [pc + length (compile\_com c)] 处的机器码匹配延续 [k]，在上述 [compile\_cont] 的意义上。

Inductive match\_config (C: code): com \* cont \* state -> configuration -> Prop :=

| match\_config\_intro: forall c k st pc,

codeseq\_at C pc (compile\_com c) ->

compile\_cont C k (pc + length (compile\_com c)) ->

match\_config C (c, k, st) (pc, nil, st).

转化关系如下图

match\_config

c / k / st ----------------------- machconfig

| |

| | + or ( \* and |c',k'| < |c,k} )

| |

v v

c' / k' / st' ----------------------- machconfig'

match\_config

注意右边更强有力的结论：

- 虚拟机执行一次或多次转换

- 或者它执行零次、一次或多次转换，但 [c,k] 的大小对严格减少。

它相当于状态：

- 虚拟机执行一次或多次转换

- 或者它执行零转换，但 [c,k] 对的大小严格减小。

为防止无限次的转换。经过反复试验，我们发现以下措施有效。它是焦点下的命令 [c] 和所有出现在延续 [k] 中的命令之和 。

Fixpoint com\_size (c: com) : nat :=

match c with

| SKIP => 1

| x ::= a => 1

| (c1 ;; c2) => com\_size c1 + com\_size c2 + 1

| IFB b THEN ifso ELSE ifnot FI => com\_size ifso + com\_size ifnot + 1

| WHILE b DO c1 END => com\_size c1 + 1

end.

Fixpoint cont\_size (k: cont) : nat :=

match k with

| Kstop => 0

| Kseq c k' => com\_size c + cont\_size k'

| Kwhile b c k' => cont\_size k'

end.

Definition measure (impconf: com \* cont \* state) : nat :=

match impconf with (c, k, m) => com\_size c + cont\_size k end.

从而我们可以证明出上述图表中的转化关系。即：

Lemma simulation\_step:

forall C impstate1 impstate2 machstate1,

kstep impstate1 impstate2 ->

match\_config C impstate1 machstate1 ->

exists machstate2,

(plus (transition C) machstate1 machstate2

\/ (star (transition C) machstate1 machstate2 /\ measure impstate2 < measure impstate1))

/\ match\_config C impstate2 machstate2.

第四章 CVA21和Compcert的转换

4.1 引言

本项目基于此前的PRP项目“指令编码及译码的形式化研究”和指导教师于程序语言领域顶级会议OOPSLA‘2020（CCF-A类会议）发表的论文"CompCertELF的工作。

指令编码器和解码器的验证是必不可少的用于形式化对机器代码的操作。现有的方法不能保证关键的一致性属性，即编码器与其对应的解码器互逆。我们观察到一致的编码器-解码器对可以自动源自嵌入在指令格式中的双射。基于这一观察，我们的课题组开发了一个框架来编写捕获这些双射的规范，以自动生成编码器和这些规范中的解码器，并用于正式验证生成的编码器和解码器的一致性和健全性在 Coq 中合成证明，得到自动生成证明的CVA21[4]。我们的工作主要是实现CAV21指令集和CompCertELF指令集的转化，并证明了这种转化的正确性和一致性，从而使CompCertELF也能够自动生成证明。

4.2 转换的实现

之前的工作已经定义了从寄存器到长度为三的比特数的转换。

Program Definition encode\_ireg\_u3 (r:ireg) : res u3 :=

do b <- encode\_ireg r;

if assertLength b 3 then

OK (exist \_ b \_)

else Error (msg "impossible").

因此，我们定义了上述encode函数的decode函数，它接受一个长度为3的比特数，得到一个寄存器。

Definition decode\_ireg (bs: u3) : res ireg :=

let bs' := proj1\_sig bs in

let n := bits\_to\_Z bs' in

if Z.eqb n 0 then OK(RAX) (\*\*r b["000"] \*)

else if Z.eqb n 1 then OK(RCX) (\*\*r b["001"] \*)

else if Z.eqb n 2 then OK(RDX) (\*\*r b["010"] \*)

else if Z.eqb n 3 then OK(RBX) (\*\*r b["011"] \*)

else if Z.eqb n 4 then OK(RSP) (\*\*r b["100"] \*)

else if Z.eqb n 5 then OK(RBP) (\*\*r b["101"] \*)

else if Z.eqb n 6 then OK(RSI) (\*\*r b["110"] \*)

else if Z.eqb n 7 then OK(RDI) (\*\*r b["111"] \*)

else Error(msg "reg not found").

接下来，我们通过证明decode的内容经过encode后再进行一次decode得到和原来一模一样，从而得到encode和decode操作互逆的性质。

Lemma ireg\_decode\_consistency :

forall r encoded,

decode\_ireg encoded = OK(r) ->

encode\_ireg\_u3 r = OK(encoded).

Proof.

intros.

destruct encoded as [b Hlen].

(\*\* extract three bits from b \*)

destruct b as [| b0 b]; try discriminate; inversion Hlen. (\*\*r the 1st one \*)

destruct b as [| b1 b]; try discriminate; inversion Hlen. (\*\*r the 2nd one \*)

destruct b as [| b2 b]; try discriminate; inversion Hlen. (\*\*r the 3rd one \*)

destruct b; try discriminate. (\*\*r b is a empty list now, eliminate other possibility \*)

(\*\* case analysis on [b0, b1, b2] \*)

destruct b0, b1, b2 eqn:Eb;

unfold decode\_ireg in H; simpl in H; (\*\*r extract decoded result r from H \*)

inversion H; subst; (\*\*r subst r \*)

unfold encode\_ireg\_u3; simpl; (\*\*r calculate encode\_ireg\_u3 \*)

unfold char\_to\_bool; simpl;

replace eq\_refl with Hlen;

try reflexivity; (\*\*r to solve OK(exsit \_ \_ Hlen) = OK(exsit \_ \_ Hlen) \*)

try apply proof\_irr. (\*\*r to solve e = eq\_refl \*)

Qed.

对于浮点寄存器进行同样的操作。我们可以得到类似的函数和结论。

函数：encode\_freg\_u3 (r:freg) ,decode\_freg (bs: u3)

结论： freg\_decode\_consistency

对于整数类型Z，我们也执行转换，将它编码成长度为8的比特数，长度为16的比特数或者长度为32为的比特数。

Program Definition encode\_ofs\_u32 (ofs :Z) :res u32 :=

let ofs32 := bytes\_to\_bits\_opt (bytes\_of\_int 4 ofs) in

if assertLength ofs32 32 then

OK (exist \_ ofs32 \_)

else Error (msg "impossible").

同时，也定义它的逆操作 decode\_ofs\_u32 (r:u32) ，并证明了两者的一致性。

在CAV21中的地址模式如下：

Inductive AddrE: Type :=

| AddrE12(uvar3\_0:u3)

| AddrE11(uvar32\_0:u32)

| AddrE10(uvar2\_0:u2)(uvar3\_1:u3)(uvar3\_2:u3)

| AddrE9(uvar2\_0:u2)(uvar3\_1:u3)(uvar32\_2:u32)

| AddrE8(uvar3\_0:u3)

| AddrE7(uvar32\_0:u32)

| AddrE6(uvar3\_0:u3)(uvar32\_1:u32)

| AddrE5(uvar2\_0:u2)(uvar3\_1:u3)(uvar3\_2:u3)(uvar32\_3:u32)

| AddrE4(uvar3\_0:u3)(uvar32\_1:u32).

我们定义从ccelf的地址模式到CAV21地址模式的转换 translate\_Addrmode\_AddrE (sofs: Z) (i:instruction) (addr:addrmode)。

同时也定义从CAV21地址模式到ccelf地址模式的转换translate\_AddrE\_Addrmode (sofs: Z) (i:instruction) (addr:AddrE)，并进行了两者的一致性证明。

到此，我们可以正式定义从ccelf指令与CAV21指令的相互转换关系了。

由于指令过多，在此节选部分代码如下：

Definition translate\_instr (ofs: Z) (i:instruction) : res Instruction :=

match i with

| Asm.Pmov\_rr rd r1 =>

do rdbits <- encode\_ireg\_u3 rd;

do r1bits <- encode\_ireg\_u3 r1;

OK (Pmovl\_rr rdbits r1bits)

| Asm.Pmovl\_rm rd addr =>

do rdbits <- encode\_ireg\_u3 rd;

do a <- translate\_Addrmode\_AddrE ofs i addr;

OK (Pmovl\_rm a rdbits)

| Asm.Pmovl\_ri rd imm =>

do rdbits <- encode\_ireg\_u3 rd;

do imm32 <- encode\_ofs\_u32 (Int.intval imm);

OK (Pmovl\_ri rdbits imm32)

|\_=>Error (msg “Not exists”)

end.

同时，定义相反的转换，节选如下：

Definition translate\_Instr (ofs: Z) (i:Instruction) : res instruction :=

match i with

|Pmovl\_rr rdbits r1bits =>

do rd <- decode\_ireg\_u3 rdbits;

do r1 <- decode\_ireg\_u3 r1bits;

OK (Pmov\_rr rd r1)

|Pmovl\_rm a rdbits =>

do rd <- decode\_ireg\_u3 rdbits;

do addr <- translate\_AddrE\_Addrmode ofs i a;

OK (Pmovl\_rm rd addr)

|Pmovl\_ri rdbits imm32 =>

do rdbits <- decode\_ireg\_u3 rd;

do imm32 <- decode\_ofs\_u32 (Int.intval imm);

| \_=>Error (msg "unfinished")

end.

最后我们证明了这样的转换是正确的。

**Lemma translate\_instr\_consistency1 : forall ofs i I,**

**translate\_instr ofs i = OK I ->**

**translate\_Instr ofs I = OK i.**

**Lemma translate\_Instr\_consistency1 : forall ofs i I,**

**translate\_Instr ofs I = OK i ->**

**translate\_instr ofs i = OK I.**

第五章 结论

通过前期的学习，我们学到了如果用Coq定义类型，函数和关系等，并且学会了Coq中的一系列证明策略。在software foundations和DeepSpec暑期学校的学习，我们积累了验证编译器的各个环节的定义和证明。最终，我们完成了CAV21与CompCert指令集的转换与证明，成功使CompCert实现自动证明。

第六章 参考文献

[1] 盛枫、窦亮、杨宗源. <基于Coq的命令式语言编译器机械验证的研究与实现\_盛枫.pdf> [J]. 2014-07-10.

[2] 杨宗源 窦 刘. <定理证明器Coq与机械语义研究\_窦亮.pdf> [J]. 2014-02-21.

[3] 付明 张. <基于Z3的Coq自动证明策略的设计和实现\_张恒若.pdf> [J]. 2016-06-20.

[4] XU X, WU J, WANG Y, et al. Automatic Generation and Validation of Instruction Encoders and Decoders [M]. Computer Aided Verification. 2021: 728-51.

第七章 致谢

从加入这个项目到最终完成论文撰写，期间困难重重。项目的正常推进离不开老师和师兄师姐们的关心和帮助。首先，感谢汪宇霆老师教会了我们如何正确地对待当前所遇到的困难，并怎样采取措施去攻克它。其次，感谢组内师兄师姐们的帮助。其中，吴进华学长亲自为我们讲解课题组的研究方向和已经完成的工作，并且手把手地带我们参与到我们课题组的项目中。而徐翔哲学长帮助我们找到项目中我们所需要的函数，并为我们一一讲解它们的用法。他们让我感受到了科研学子对于科学研究的的那份热爱、专注与敬畏。我也会向他们学习，在科研道路上加倍用心与努力！