**北京航空航天大学计算机学院**

**硕士学位论文开题报告**

**论文题目**：安全C编译器的形式化验证方法的研究和实现

**专 业**：软件工程

**研究方向**：形式化验证

**研 究 生**：陈志伟

**学 号**：SY1406108

**指导教师**：马殿富 教授

**北京航空航天大学计算机学院**

2015年11月29日

目 录

1 课题来源 1

2 论文选题的背景与意义 1

3 国内外研究现状及发展动态 2

3.1 测试方法 2

3.2 模型检验方法 3

3.2.1 基本思想 3

3.2.2 状态爆炸问题 3

3.2.3 抽象方法 4

3.3 定理证明方法 5

3.4 翻译确认方法 7

3.4.1 编译过程正确性 7

3.4.2 基本思想 7

3.4.3 证明过程 8

4 论文的研究内容及拟采取的技术方案 8

4.1 研究目标 8

4.2 主要研究内容 9

4.3 拟采取的技术方案 9

5 关键技术与难点 11

6 论文研究计划 12

7 主要参考文献 12

**论文题目**

# 课题来源

民机专项“符合DO-178B/C的A级机载软件开发与认证技术研究”

# 论文选题的背景与意义

随着计算机应用的飞速发展，软件已渗透到国民经济和国防建设的各个领域，在信息社会中扮演着至关重要的角色。安全攸关软件，如航空机载软件，作为各类安全关键系统的构成部分，其内部结构越来越复杂、应用环境越来越开放，这些因素使得人们更加关注其可信性问题。TCG(trusted computing group)对可信的定义是：一个实体在实现预定目标时，若其行为总是符合预期，则该实体是可信的。一个软件若是可信的，则必然是正确的，能够如预期那样工作。

在工业上，硬件的正确性验证已经比较成熟，芯片设计师通过硬件验证工具确保芯片正确地执行机器指令。软件的可信性验证技术也越来越受到广泛的关注和重视，如2007年我国国家自然科学基金委员会启动了“十一五”期间“可信软件基础研究”重大研究计划，由信息科学部、数学物理科学部和管理科学部联合组织，实施周期6年，计划经费1.5亿元。然而，目前在软件验证技术上所取得的一定进展都只是局限在实验室的范围内，远没有达到在工业界广泛应用的程度，这是由于软件的可信性验证技术的复杂程度和不完善造成的。

编译器作为软件开发过程中的关键工具，是实现软件从设计到能在硬件上运行的桥梁，它是否具备安全、可信的性质？是进行软件开发所面临的重要难题。如果编译器不可信，则无法保证其所生成代码的可信性，非可信编译器在对程序代码进行编译的过程中，很可能篡改其原本语义，生成不安全的目标代码。特别在安全关键领域中，如航天、核工业等，编译器的安全可信有着至关重要的作用，由编译所引入的错误可能会带来灾难性的后果，如2010年，由于软件故障，导致美国海军的一架无人直升机发生系统故障，闯进华盛顿上空的禁飞区。随着计算机技术的发展，对编译的要求越来越高，例如高级语言编译器中增加了大量的优化，而优化又可能带来不可预测的问题，导致编译过的可执行代码在运行过程中产生非期望的输出。因此在软件开发阶段必须对编译器的可信性进行充分验证。

传统的编译器验证方法是大量的进行软件测试，但是软件测试难以达到完全覆盖，并不能充分地保证编译器的安全可信，同时，软件测试的验证方法日益受到复杂软件测试的正确性以及软件测试效率等问题的挑战。为了应对上述挑战，近年来，形式化验证方法在编译器的可信性验证中得到了持续的关注。形式化验证方法可以从数学角度对编译器进行描述，对编译过程的语义和语言属性的等价性进行证明，能够充分地保证编译器可信性。形式化验证是分析系统是否具有所期望性质的过程，主要分有两种途径：模型检验和定理证明。同时，还有另一种形式验证方法——翻译确认技术。

实践中在安全相关系统中使用 C 语言，那么必须对语言的使用加以限制，避免那些确实可以产生问题或编译器支持的不完善的地方，直到它是可以应用的。MISRA-C，汽车制造业嵌入式C编码标准，从MISRA-C:2004开始其应用范围扩大到其他高安全性系统，由该规范定义的Ｃ语言被认为是易读、可靠、可移植、易于维护的。C安全子集将MISRA-C与航天型号软件的特点相结合，重新定义了一系列C语言软件的编程准则，为安全相关领域的C语言软件提供了相应的安全语言规范和编译要求。C安全子集严格要求了编译器的成熟度及稳定性，编译器必须忠实地反映源语言的代码结构和语义，以方便编译前后的代码审查、比较和追踪，确保编译后代码的可信。

# 国内外研究现状及发展动态

## 测试方法

软件测试是通过执行软件来判断软件是否具备所期望的性质,是可信软件开发中一个行之有效的、必不可少的、客观地评估软件可信性的方法。在高可信软件开发中, 软件测试的开销往往大于50%。

编译器测试常用的有两种策略:动态测试和静态测试。动态测试是使用一系列由测试环境控制的测试数据执行程序,然后比较测试程序的实际输出结果和测试用例中的预期输出结果。静态测试就是在不执行程序的前提下对程序进行的测试。对编译器的测试往往使用动态测试策略,选择一个已被验证的编译器测试套件对编译器进行测试。动态测试可以被划分为白盒测试技术和黑盒测试技术两种[1]。

白盒测试技术是基于对编译器内部结构的检查，即使用测试数据对程序的控制结构、数据流等逻辑进行检查。白盒测试技术是基于对测试源代码的利用，主要用来测试编译器的单独模块，以保证其每个部分都可以正常运作。相反，基于测试套件的黑盒测试往往被用来对编译器进行性能测试和鲁棒性测试，测试数据只取决于编译器的规格说明书。测试数据不关心被测程序的结构，只关注被测程序的特征和外部行为。黑盒测试技术被用来测试编译器对编程语言声明和对用户的接口，确认编译器对语言标准的实现情况己经成为黑盒测试的一个越来越重要的方向。

目前，对于安全攸关系统，软件测试技术依旧面临着重大挑战。安全攸关软件不仅要求在其环境处于正常状态时保证系统的安全性，而且要求环境处于非正常状态时也能使系统安全地进入安全状态，因而往往难以获得充分的数据来测试软件应付危险情况的能力，不能满足可靠安全性测试的需求。

## 模型检验方法

### 基本思想

模型检测[2~4]是一种自动形式化验证技术，用于对一个计算机系统的正确行为属性进行判断。模型检测的基本方法是用一个状态迁移图*M* 来表示所要检测的系统的模型，并用模态/时序逻辑（如计算树逻辑（computation tree logic, CTL）、命题线性时序逻辑（linear temporal logic, LTL）、命题μ演算等）公式φ 来描述系统的正确行为属性，然后通过对模型状态空间穷举搜索来判断该公式是否能够在模型上被满足。如果公式在模型上满足，即M |= φ，则系统的正确性得到证实（verified）；否则，就表明系统中存在错误，即M |= ~φ，系统正确性被证伪（falsified）。

模态/时序逻辑是模型检测的基础。常用的模态逻辑有三种，分别是：计算树逻辑[5]、命题线性时序逻辑[6]和命题μ演算逻辑[7~8]。

### 状态爆炸问题

模型检测在实际中应用的主要瓶颈是状态爆炸问题（state‐explosion problem）。由于模型检测基于穷举搜索对正确属性进行判断，所以它适用于对有限的、状态空间较小的系统进行分析。然而，在实际应用中经常存在着状态空间庞大的系统。如对于软件系统，由于存在着无限数据域（如整数）、无界数据类型（如链表）、以及复杂的控制结构（如递归），导致软件系统的状态空间可以是无限大，直接对它们进行模型检测在实际中是不可行的。因此，正如图灵奖得主Clarke所说，解决状态爆炸问题是模型检测研究中的一个最根本的工作[9]。

### 抽象方法

抽象方法是解决状态爆炸问题的一个重要的方法。它的基本思想是首先构造一个比原系统的具体模型小的有限抽象模型，然后通过正确属性在抽象模型上的检测结果推测出其在原来的具体模型上是否可满足。抽象方法的依据是，对于所给定的某种正确属性而言，待检测的原系统的许多信息（如某些程序变量的取值、进程的标识符、调用栈中活动记录等）是无关的。因此，这些信息可以从具体模型中抽象出去。这样不仅简化了模型，同时保留了必要的信息，使得抽象的模型检测得以有效地进行[10, 11]。

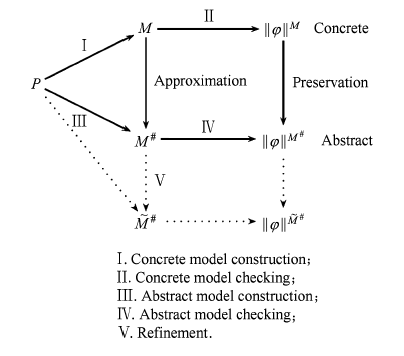
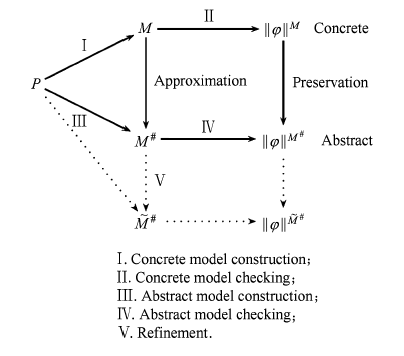
软件模型检测中的抽象的主要过程为：

Fig.1 Overview of abstraction in software model

Checking.

给定一个程序P以及欲对其进行分析的行为属性φ，具体的软件模型检测的过程主要包括2个步骤：1）模型构建。使用一个状态转换系统M来表示程序P的语义（如图1中I所示），称其为P的模型。2）模型检测。用时序逻辑公式φ 描述所关心的行为属性，通过计算||φ||*M*的值来验证（如图1中II所示）。由于程序的状态空间巨大甚至是无限，通常无法直接构造模型*M*。因此我们需要构造一个精简的有限抽象模型M#（如图中III所示），使用它来逼近具体模型M。通过抽象实现对φ在M#上的有效验证（如图1中IV所示），并进一步根据M#与M之间的逼近关系所对应的程序属性的保持关系，判断φ在M上成立与否。当由于抽象导致无法得到确定的模型检测结果时，需要通过抽象精化[12]（abstraction refinement）构造更加精确的抽象模型（如图1中V所示）。

## 定理证明方法

定理证明技术是将软件系统和性质都用逻辑方法来规约,通过基于公理和推理规则组成的形式系统，以如同数学中定理证明的方法来证明软件系统是否具备所期望的关键性质。

* + 1. 霍尔逻辑方法

霍尔逻辑[13, 14]描述程序正确性的一般形式为：

{Pre}P{Post}

其中，Pre称为前置断言，Post称为后置断言，P 为程序代码。若P 的每一次计算开始于满足Pre 的状态，执行终止且终止时的状态满足Post，则正确性公式为真，程序P具有完全正确性。

在霍尔逻辑中存在一组证明规则，称为霍尔规则。这些规则是语法制导的，它们把证明一条复合命令的部分正确性断言简化成证明它的直接子命令的部分正确性断言。霍尔规则中的赋值规则和推论规则如下所示：

赋值规则：

{P[E/x]} x := E{P}

表示把P中变量x的所有自由出现都替换成表达式E得到的谓词。赋值公理表示如果执行赋值语句ｘ:= E后P为真，则在执行赋值语句之前P[E/x]为真。

推论规则：

P’→P{P} S{Q} Q→Q’

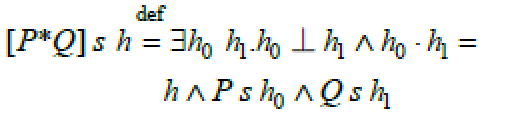
{P’} S {Q’}

表示如果一个Hoare三元组为真，那么把前置条件加强或者后置条件减弱也为真。

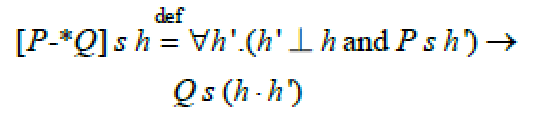
用霍尔规则进行推导能得到部分正确性断言的形式化证明，所以霍尔逻辑能用于机器证明。在证明的过程中，过于形式化的推导会分散人们在证明过程上的精力，可以把这方面的工作交给一些辅助证明工具如LCF或HOL去完成。实践中, 使用霍尔逻辑证明很小的程序的正确性也不是那么容易。后续的研究工作集中于扩展霍尔逻辑, 以便验证更复杂的语言结构, 以及寻求更好的方式来构造、表示和检查形式证明。

* + 1. 分离逻辑方法

分离逻辑[15, 16]是对霍尔逻辑的一个扩展，通过提供表达显示分离的逻辑连接词以及相应的推导规则，消除了共享的可能，能够以自然的方式来描述计算过程中内存的属性和相关操作, 从而简化了对指针程序的验证工作。分离逻辑被证明具有更强的验证能力，如对并发程序和资源管理的验证，使得程序验证和推理技术前进了一大步。因此, 继霍尔逻辑之后，分离逻辑有望成为程序可信验证的一种重要方法。

 分离逻辑中，前置条件和后置条件中的程序状态主要由栈s和堆h构成,栈是变量到值的映射,而堆是有限的地址集合到值的映射。在程序验证时,可以将栈看作对寄存器内容的描述,而堆是对可寻址内存内容的描述。分离逻辑中引入了两个新的分离逻辑连接词：分离合取\*和分离蕴含-\*。[P\*Q] s h 表示整个堆h被分成两个不相交的部分h0和h1，并且对子堆h1断言P成立，而对子堆h1断言 Q 成立。形式化表示如下：

其中h0⊥h1表示堆h0和h1不相交, h0 . h1表示堆h0和h1的联合。

[P\_\*Q] s h表示如果当前堆h 通过一个分离的部分h’扩展，并且对h’断言 P 成立，则对扩展后的堆(h ⋅ h')断言 Q 成立。形式化表示如下：

同样。分离逻辑引入了一些新的推导规则。如Frame规则

{P}C{Q}

{P\*R}C{Q\*R}

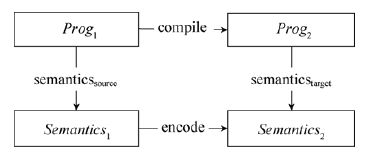
其中，代码段C不会对断言R中的自由变量赋值。分离逻辑作为一种近年来提出的逻辑，因其本身所蕴含的分离思想，在验证包含指针的程序时，能够简洁、优雅地支持进行局部推理和模块化推理，已经在程序验证领域得到了重视和广泛使用。但是，将分离逻辑用于解决更复杂的软件系统的源代码级验证仍然需要进一步解决许多技术难题，如对语言类型的支持范围、验证过程的自动化程度等。

最后，定理证明目前比较好的方式是使用编程和证明统一的框架，如Twelf[17]、Coq[18]和Isabelle[19]等。

## 翻译确认方法

### 编译过程正确性

编译过程正确性的形式化定义可用如下图2的转换示意图表示，故对其形式化证明就是证明对应的转换示意图[20]的成立。

Fig.2 Compiler correctness diagram

图中的箭头可看成是函数映射过程。编译正确性可以用如下等式来表示：

*encode(semanticssource(P)) = semanticstarget(compile(P))*

程序*P*可以使用不同的语义[21]来解释，如操作语义、公理语义、指称语义等。这种示意图最早起源于McCarthy和Painter的工作，他们使用操作语义证明了源语言是简单数学表达式的编译器的正确性[22, 23]。图2证明编译过程的正确性具有通用性，但是如何将源程序与编译后目标程序的语义关联对应起来是证明的关键问题。

### 基本思想

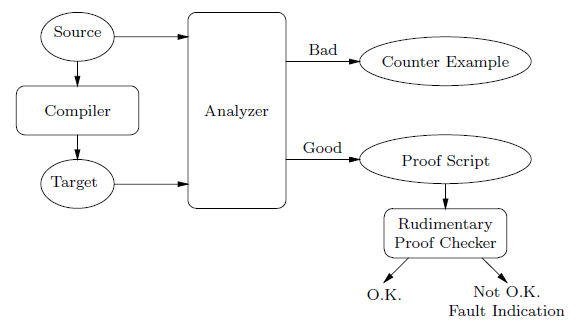
翻译确认[24~25]是一种用于确认编译器或代码生成器的源和目标之间的语义等价性的形式化方法，它通过证明源代码和目标代码的语义等价性来证明编译器的正确性。使用翻译确认方法需要构造一个确认器（validator），确认器在编译器每一次运行后形式化地证明生成的目标代码是源代码的一个正确翻译。确认器不关心编译器的具体实现，只对编译器的源代码和目标代码进行处理，如果验证成功则编译继续进行，如果发现语义矛盾之处则输出一个警报或取消编译。

编译器的形式化验证可以减弱为对确认器进行形式化验证工作。相对于编译器而言，确认器的形式化验证工作是比较简单的，从而大大减轻了证明的难度及工作量。同时，由于确认器不关心编译器的具体实现，因此没有限制编译器的设计以及未来的优化完善等，且确认器是可重用的。

### 证明过程

一个自动化的翻译确认器应该包括以下要素：（1）一个用于描述源语言和目标语言的公共语义框架；（2）基于公共语义框架形式化地建立的目标代码和源代码之间的“正确执行”定理；（3）一个有效的证明方法，它允许证明代表着生成的目标代码的一个语义框架的模型，正确的实现了代表着源代码的另一个模型；（4）通过Analyzer执行证明方法的自动化，如果成功则生成一个证明脚本；（5）一个证明检查器，用于对Analyzer产生的证明脚本进行检查。

翻译确认的过程如下图3所示：

Fig.3 Translation validation process

# 论文的研究内容及拟采取的技术方案

## 研究目标

在安全攸关软件的开发中，编译器作为软件开发过程中的关键工具，实现把系统的源代码编译成目标代码的过程，其是否具备安全、可信的性质，是软件能否达到A级开发标准的重要因素。本课题的研究目标是构建一个可以实现对编译器建模及形式化验证的工具，该工具可以对编译前后的源代码和目标代码进行变换和推导，验证其语义是否保持一致，从而得出编译器是否安全可信。

## 主要研究内容

为了实现上述研究目标，本文拟进行如下几个方面的研究：

1. 形式化验证工具系统整体框架设计；
2. 开发由单条安全C文法结构生成的目标代码的形式化验证算法；
3. 研究从单条安全C文法结构到整个程序的目标代码的综合方法；
4. 形式化验证工具证明策略库的构建与使用；
5. 形式化验证工具用户交互界面的研究与改进；

## 拟采取的技术方案

### 形式化验证工具系统框架架构

本课题要实现一个形式化验证工具，其系统框架架构如图4所示：

输出

Good

格式化代码

待证明代码

应用公理

选择公理

用户界面

配置参数

证明结果

公理库

规范解析

形式化验证算法

证明序列

Fig.4 Formal verification tools framework

* 公理库提供不同指令集（如MIPS、PowerPC）的指称语义给验证算法使用；
* 程序规范解析将待证明的程序转化为计算机可读的中间形式；
* 形式化验证算法为此开发工具的核心，需要完成推导和证明过程；
* 用户界面辅助用户进行形式化验证。

### 公理库的构建与使用

* 开发不同指令集的指称语义集，如MIPS、PowerPC、Arms等；
* 通过交互界面来选择使用不同的指称语义集，语义集必须和源代码对应。

### 形式化验证算法

* 把C程序抽取为多条具有完整语义的C文法结构；
* 把每条C文法结构用指称语义进行形式化，直接得到C文法所代表的指称语义；
* 把每条具有完整语义的C文法结构编译生成的目标代码代换为指称语义的形式；
* 基于得到的指称语义形式进行推理，获得目标代码的最简指称形式；
* 确认每条C文法结构的语义是否和编译后的目标代码的语义一致。

### 用户交互界面的研究与改进

* 加入证明策略智能提示功能，为用户交互证明时提供证明帮助和引导。
* 自动精简证明序列，删除冗余证明项，提取程序形式化表示

# 关键技术与难点

## C文法结构

语境表示待证明序列中每一个证明项所在的环境和上下文，蒙太古语用学中提到相同对象在不同语境中语义不同。因而，在把C程序抽取为多条C文法结构，如if语句、while语句等，由于其语境确定（主要由局部变量和全局变量等组成），则每个C文法结构就具有完整的语义，于是便可以基于每个文法的语义进行形式验证。

难点在于把C程序代码抽取为多个C文法结构的算法的设计，这是我们的整个课题能继续向下进行的基础。

## 指称语义

指称语义是采用形式系统方法，用相应的数学对象（如set, function等）对一个即定形式语言的语义进行注释的学问。指称语义还可以解释为：存在着两个域，一个是语法域，在语法域中定义了一个形式语言系统；另外一个是数学域（或称之为已知语义的形式系统）。课题中我们将用指称语义的方法来分别表示C文法结构和目标代码的语义，难点在于如何正确的对它们进行建模和形式化表示，这一点还需要大量的研究。

## 语义一致

编译器的任务是将正确的源程序翻译为正确的目标程序。编译器自身的可信判断的依据就是编译变换过程是否使得源代码与编译生成的目标代码之间具有一致的语义。因而课题中需要构造一个确认器（validator）来验证源代码和目标代码的语义是否一致。

虽然源代码和目标代码的语义都是用指称语义的形式表示，但两种语言的抽象层次不同，即目标代码为汇编语言更接近于硬件，因此二者的指称语义表达形式有较大的差异，如何为确认器设计算法来关注本质的东西而忽略掉差异是课一个有待解决的难题。

# 论文研究计划

* 2015年12月－2016年01月 研究相关资料和技术
* 2016年02月－2016年03月 技术尝试，概要设计
* 2016年04月－2016年05月 详细设计，撰写小论文
* 2016年06月－2016年09月 编码实现，测试分析
* 2016年10月－2016年12月 整理资料，撰写毕业论文

# 主要参考文献

1. A.S.Boujanvah, K.Salehf. Compiler test case generation methods: a survey and assessment. Information and Software Technology, 39 -1997, 617-625.
2. Clarke E M, Grumber O, Peled D. Model Checking[M]. Cambridge: MIT Press, 1999.
3. Baier C, Katoen J P. Principles of Model Checking[M]. Cambridge: MIT Press, 2008.
4. Lin Huimin, Zhang Wenhui. Model checking: Theories, techniques and applications[J]. Acta Electronica Sinica, 2002, 30(12): 1907-1912.
5. Clarke E M, Emerson E A, Sistla A P. Automatic verification of finite state concurrent system using temporal logic specifications[A]. 10th Annual ACM Symposium on Principle of Programming Languages[C]. New York: ACM Press, 1983, 117-126.
6. M Y Vardi, P Wolper. An automata – theoretic approach to automatic program verification[A]. 1st IEEE Symposium on Logic in Computer Science[C]. Los Alamitos: IEEE Computer Society, 1986, 322-331.
7. E AIIen Emerson, Chin-Laung Lei. Efficient modeI checking in fragments of the propositional Mu-CaIcuIus[A]. 1st IEEE Symposium on Logic in Computer Science[C]. Los Alamitos：IEEE Computer Society, 1986, 267 - 278.
8. Colin Stirling, David Walker. LocaI model checking in the modaI Mu-Calculus[A]. Lecture Notes in Computer Science 351 - 3rd International Joint Conference on Theory and Practice of Software Development[C]. Berlin: Springer-Verlag, 1989, 369 - 383.
9. Clarke E M． My 27-year quest to overcome the state explosion problem ［C］ //Proc of the 24th Annual IEEE Symp on Logic in Computer Science(LICS’09). Piscataway, NJ: IEEE, 2009: No, 3.
10. Cousot P, Cousot R, Mauborgne L. Theories, solvers and static analysis by abstract interpretation[J]. Journal of the ACM (JACM), 2012, 59(６): 1-56.
11. Li Mengjun, Li Zhoujun, Chen Huowang. Program verification techniques based on abstract interpretation theory[J]. Journal of Software, 2008, 19(1): 17-26.
12. Clarke E, Grumberg O, Jha S, et al. Counterexample-guided abstraction refinement for symbolic model checking[J]. Journal of the ACM, 2003, 50(5): 752-794.
13. Hoare C A R. An axiomatic basis for computer programming[J]. Communications of the ACM, 1969, 12: 576-580.
14. Floyd R W. Assigning meanings to programs[C]//Proceedings of Symposium on Applied Mathematics, 1967, 19-31.
15. Magill S, Nanevski A, Clarke E, et al. Inferring invariants in separation logic for imperative list-processing programs[C]//Proc of the 3rd Workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE 2006), Charleston, 2006: 47-60.
16. Ireland A. Towards automatic assertion refinement for separation logic[C]//Proc of the ASE 2006. [S.l.]: IEEE Computer Society, 2006: 309-312.
17. Schürmann C. The Twelf proof assistant[M]//Theorem Proving in Higher Order Logics. Springer Berlin Heidelberg, 2009: 79-83.
18. Huet, Gérard, Gilles Kahn, and Christine Paulin-Mohring. "The Coq Proof Assistant A Tutorial." Rapport Technique 178 (1997).
19. Paulson, Lawrence C., and Markus Wenzel. Isabelle/HOL: a proof assistant for higher-order logic. Vol. 2283. Springer, 2002.
20. McCarthy J, Painter J. Correctness of a compiler for arithmetical expressions[C]//Mathematical Aspects of Computer Science 19: Proc of Symposia in Applied Mathematics, 1967: 33−41.
21. Charles N. Fischer, Ronald K. Cytron, Richard J. LeBlanc, Jr. 编译器构造. 北京: 清华大学出版社, 2012: 7~10.
22. Thatcher J W, Wagner E G, Wright J B. More on advice on structuring compilers and proving them correct[J]. Theoretical Computer Science, 1981, 15(3): 223−249.
23. Stephenson K. Compiler correctness using algebraic operational semantics, CSR 1-97[R/OL]. University of Wales Swansea, 1997. http://www-compsci.swan.ac.uk/reports/yr1997/CSR1-97.pdf.
24. Pnueli A, Siegel M, Singerman E. Translation Validation[C]//Proc. 4th Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems. 1998:151-166.
25. Fang Yi. Translatin of Optimizing Compilers[D]. New York University, 2005.
26. Leroy X. Formal verification of a realistic compiler[M]//Communications of the ACM, 2009.
27. Leroy X. A formally verified compiler back-end[J]. Journal of Automated Reasoning, 2009, 43(4): 363-446.
28. Leroy X. Mechanized semantics for compiler verification[M]//Certified Programs and Proofs. Springer Berlin Heidelberg, 2012: 4-6.
29. Russell B. Correctness of the compiling process based on axiomatic semantics[J]. Acta Informatica, 1980, 14(1): 1-20.
30. 王蕾, 石刚, 董渊, 等. 一个 C 语言安全子集的可信编译器[J]. 计算机科学, 2013, 40(9): 30-34.
31. 何炎祥, 吴伟, 刘陶, 等. 可信编译理论及其核心实现技术: 研究综述[J]. 计算机科学与探索, 2011, 5(1): 1-22.