# 技术视角——一个编译器的故事

作者：格雷格·莫里塞特
译者：董　渊

关键词：编译器　形式化

在20世纪70年代初，佛洛德（Floyd）、迪杰斯特拉（Dijkstra）和霍尔（Hoare）等先驱者认为应该给出程序的形式化规范并证明其正确性。然而，在过去的40年里，大多数计算机科学家都认为这是一个不切实际的目标。也许最重要的原因就是基于一个简单的经济学因素：纵观20世纪80年代和90年代，工业界更关心的是产品上市的时间，而不是与（程序）正确性相关的问题。

然而，在经过这些年争论以后，情况发生了极大变化：在诸如嵌入式系统等快速增长且与生命财产安全密切相关的领域中，安全性和可靠性已经成为最关键的考虑因素。即便在不那么关键的领域中，开发人员也必须为缓冲区溢出和竞态条件等可能造成安全性暴露的程序错误而提心吊胆。研究人员已经开发出各种各样的诸如强静态类型检查器、软件模型检查器、抽象解释器等工具，所有这些工具均可以（也已经）用于增强各种程序安全性。可见，在今天形式化方法已经被广泛应用，尽管它是隐藏在各种工具的背后。

然而，这些工具通常都应用在源代码层面（或至多是在虚拟机字节码层面），而不是在机器码层面。在使用这些工具时，必须对编译器从源代码到机器码的工作方式做一些假定。比如说，尽管C语言规范中没有规定函数参数的求值顺序，但是大多数分析工具都假定编译器采用从左到右的求值策略，其原因就是因为分析所有可能的求值策略花费时间太多。如果有此类不满足前提假定的情形或者编译器自身有错误，将很容易导致这类分析工具失效。

在下一篇文章中，泽维尔·乐华（Xavier Leroy）处理上述问题的方式是构造一个编译器，并用Coq证明开发系统验证了这个编译器。虽说他不是第一位开发出经过验证的程序转换器的人，但由于以下3个原因，他所开发的编译器值得称道并令人振奋：第一，该编译器将一种很有用的源语言（C的一个重要子集）翻译为PowerPC汇编代码，使之可直接用于范围广泛的嵌入式应用的开发；第二，该编译器包含了很多分析与优化，诸如活跃分析和图着色寄存器分配，这使得其生成的代码可以匹敌于gcc-0产生的代码；第三，其正确性证明经过机械化检查，达到了我们所能期望的最高可信程度。简而言之，开发人员能够确信“即便经过优化，乐华的编译器所生成的代码仍将保持与输入源代码相同的行为”。

构造除了一个经过充分验证的优化编译器之外，这项工作还有另外一些贡献：该编译器采用了模块化的管道-过滤器设计模式，将整个编译器划分为一系列规范良好的中间语言变换模块，这为加入新优化算法或者是在别的项目中重用这些模块提供了可能。比如，C语言子集的规范可以进一步用于构建新的经过验证的工具，例如源码分析器等。

同时，这个编译器还展示了如何巧妙地使用翻译确认（translation validation）来代替完整验证（full verification）这个方法是一个明智之选。采用这一方法，我们可以使用某些未经验证的计算模块进行计算（比如，变量的寄存器分配），而只需检查其输出结果的合法性（不把相互冲突的变量分配到同一个寄存器）。要保证可靠性，只需要验证相

关的检查器即可，而这种验证工作通常要比确认一个完整的分析和转换程序要容易得多。可以说，作为一种工程技术，翻译确认能大幅度减轻构建经过验证的系统的工作负担。

本文也说明了要想使形式化验证成为生产编译器或其他工具的常规方法，我们还有多远的路要走，其中最重要的就是在代码演化过程中构造和维护相关证明的开销。乐华的正确性证明的规模大约是编译器自身代码的5～6倍，这使人们很难在不破坏已有证明的情况下对编译器做较大规模的修改。而且，证明基本上是手工构造的，没有利用半自动决策过程或者证明搜索技术，而这些技术正是最近十年来进步最显著的研究领域。事实上，在乐华引领下开展这方面工作的其他研究人员认为，完全有可能将证明的规模削减一个数量级。

或许我们所面临的最大挑战是程序规范。编译器具有相对清晰的“正确性”概念（输出代码应该表现出与输入代码相同的行为），然而其他大多数软件系统的情况并不是这样。比如，对于操作系统或网络浏览器，正确性的含义是什么？我们希望能将这些系统应该遵循的某些安全方面的特性形式化，并随着我们对程序失效和攻击机理的理解的改进主动实现这些特性。这就需要有一个程序验证架构让我们可以像修改代码那样频繁地修改规范。非常幸运的是，那些经过验证的编译器让我们可以根据需要在高级语言层次进行这种修改，而不会牺牲对所生成机器代码的正确性保证。

总而言之，我认为我们正接近一种开发安全攸关的软件系统（safety- and security software system）的全新工程范型，在这种范型中，我们信赖的认证是形式化的、可用机器检查的证明，而不是人工的审计。乐华的编译器正是向着这一目标大步迈进中的令人印象深刻的一步。■

**作　者：**

**格雷格·莫里塞特**

哈佛大学计算机科学与工程系Allen B. Cutting讲席教授，副系主任。

## 《CCF会刊条例》修订内容获得通过

2010年1月30日，CCF九届四次常务理事会议原则通过了CCF学术工委对CCF主办会刊修订的若干规则。

学会现有的《CCF会刊条例》主要针对的是合作刊物，对于由学会主办或联合主办的会刊没有明确的要求或规则。为此，学术工委对原有条例进行了补充和修订，增加了关于CCF主办刊物的条目，明确了对主编、副主编及编委的要求。九届四次常务理事会要求学术工委按照新规则修订《CCF会刊条例》，成文后提交下次常务理事会表决。

## CCF将在2010年加大对会员的服务力度

2010年1月30日，CCF秘书长杜子德在九届四次常务理事会议上介绍了增强对会员服务及将2010年定为CCF服务会员年的若干设想。

2010年开展的会员服务项目包括：继续开办学科前沿讲习班（CCF ADL）和CCF走进高校系列演讲会，设立CCF青年学者访问计划，开展会员活动日，探索会员导师制和CCF会员成就展示推广等，希望通过这些项目，使会员得到更好的服务。

# 真实编译器的形式化验证*

作者：泽维尔·乐华
译者：董　渊

关键词：编译器　形式化验证　CompCert

**摘要**

本文报道CompCert编译器的开发和在形式化验证（语义保持性证明）方面的进展，该编译器将Clight（C语言的一个较大子集）编译为PowerPC汇编代码，用Coq证明辅助工具来编写编译器并证明其正确性。在关注关键软件及其形式化验证的环境中，这种经过验证的编译器是非常有用的：这些编译器的验证可保证，在源代码证明了的安全性质能够在编译产生的可执行代码里继续得到保持。

## 引言

你能信任自己用的编译器吗？通常大家认为编译器是语义透明的，即编译生成的代码，其行为理所当然地符合源代码程序的语义。然而，编译器，特别是优化编译器，是一种实施复杂符号变换的软件。即便经过了严格测试，编译器中的错误依然屡见不鲜。这些错误可能会导致编译器的编译时崩溃，而更可怕的是它们会不声不响地将原本正确的源程序变换为错误的可执行代码。

对于安全保证性能要求较低的软件而言，只需通过测试来确认其有效性即可，编译器的错误影响不大。此类测试的对象是编译器所生成的可执行代码；而严格的测试在发现源程序错误的同时，还应该发现编译器引入的错误。需要特别关注的是，编译器引入的错误通常都是那种极难发现、极难定位的错误。而对于安全攸关的、要求高安全保证的软件，情况则大不相同。此时，单纯采用测试手段已经不够了，需要采用诸如模型检测、静态分析和程序证明等一系列形式化方法作为补充甚至替代。然而，人们几乎无一例外地，都只在程序的源代码级运用各种形式化验证工具，已验证的源代码很有可能由于编译器内部的小错误而变成一个有问题的可执行程序。人们煞费苦心地通过形式化方法得到的所有特性都将随着编译器的错误而付之东流。将来，当形式化方法在源程序级别中普遍应用的时候，编译器有可能成为从程序规范到可执行代码这个完整变换链条中最为薄弱的一环。安全攸关软件产业已经开始意识到这方面问题的严重性，并已经开始使用各种技术来缓解之，诸如“在停用任何编译器优化的情况下对生成的汇编代码进行人工检查”等。然而，即使在开发时间和程序性能方面付出巨大的代价，这些技术还是治标不治本，无法从根本上解决这个难题。

* 译自《ACM通讯》2009年第7期文章Formal Verification of a Realistic Compiler；本文初稿发表于第33届ACM编程语言原理研讨会（POPL 2006）论文集。

用形式化方法对编译器自身进行验证是一个显而易见的好办法，这样做能保证编译器在进行翻译变换的同时保持源代码的语义。过去5年来，我们一直在开发一个具备这种特性的“实际的、经过验证”（realistic,verified）的编译器，并将其命名为CompCert。称其为“经过验证的”，是指这个编译器具有一个可用机器自动检查的证明机制，表明该编译器具有语义保持性，即所生成的机器代码的行为准确反映源程序的语义。称其为“实际的”，是指这个编译器可以实实在在地应用于关键软件的生产过程。就是说，该编译器处理的是一个经常被用于关键性的嵌入式软件开发的语言，既不是JAVA，也不是ML（一种通用函数式编程语言），更不是汇编代码，而是C语言的一个很大的子集。而且该编译器生成的是常用于嵌入式系统的处理器的代码。我们选择的是航空工业中非常普及的PowerPC处理器；最后，该编译器所生成的代码应该足够高效、足够紧凑，能够满足关键嵌入式系统的需求。这意味着该编译器是一个支持多遍编译、具备良好的寄存器分配算法和基本优化能力的编译器。

证明编译器正确性这个想法由来已久，第一个这方面的证明工作发表于1967年[16]，证明的是算术表达式到栈式机器代码（stack machine code）的编译，这一工作的机械化验证完成于1972年[17]。此后人们提出各种各样的证明，其覆盖范围从针对简单模型语言的单遍编译器到拥有复杂代码优化的编译器[8]。在CompCert的实践中，我们将这方面工作发展到一个复杂编译链的端到端验证方案，该编译器从一个结构化的命令式语言开始，经历8种中间语言，最终变换为汇编代码。在对CompCert的验证过程中，我们也发现，虽然许多非优化的变换在编译方面的文献中常常被认为是理所当然的，但其正确性的形式化证明竟然复杂得异乎寻常。

本文概述CompCert编译器以及使用Coq证明辅助工具给出的机械化证明[3,7]。该编译器采用经典的模块结构，其前端将C语言的子集Clight翻译为一种称为Cminor的低级结构化中间语言；而包含少量优化功能的后端从Cminor生成PowerPc汇编代码。关于Clight的详细描述参见文献[5]，编译器前端工作参看文献[4]，而后端相关的工作参看文献[11,13]，基于Coq开发并带有详尽注释的全部源代码可从项目网站[12]获得。

本文后续部分组织如下：第二节比较了几种“建立可信编译结果”的方法并给出形式化定义；第三节介绍CompCert编译器的结构、性能以及在使用Coq工具编写编译器代码的同时进行正确性证明的方法。限于文章篇幅，我们无法深入介绍每一个编译遍（crucial pass）的形式化验证，因此在第四节中介绍其中最为关键的一个技术要点——寄存器分配。最后，第五节给出初步结论和下一步研究工作。

# 通向可信编译器之路

## 语义保持的记法

考虑源程序$S$和编译器生成的程序$C$，我们的目标是证明编译过程中$S$的语义始终保持不变。为了更精确地描述语义保持特征，我们假设给定的源语言和目标语言的语义将外部可见行为$B$关联到程序$S$和$C$。用记号$S \Downarrow B$来表示程序$S$的执行具有外部可见行为$B$。在CompCert中的可见行为包括终止、发散和执行错误（调用可能导致系统崩溃的无定义操作，如数组访问、越界访问等）。在所有情况下，程序行为也包括运行过程中执行的输入、输出操作（系统调用）的轨迹。这些行为准确反映程序用户或者一般而言程序与之交互的外部世界能够观察到的所有内容。

关于编译过程中语义保持的最强的记法是源程序$S$和编译结果程序$C$具有完全等价的外部可见行为。

$$\forall B, S \Downarrow B \Leftrightarrow C \Downarrow B \qquad (1)$$

公式(1)的要求太强,因而根本无法使用。在源语言具有非确定性，那就允许编译器选择源程序的一种可能行为。（例如，C编译器可以在C规范允许的各种求值顺序中选择某一特定的求值顺序）。

在这种情况下，程序$C$的可能行为将比$S$少。更进一步，编译器的优化算法还可能消除某些出错行为。比如，如果$S$在整数被0除的情况下出错，编译器则根据该计算结果没有引用的实际情况而消除这个计算过程，这种情况下$C$程序就不会出同样的错误。考虑到编译器需要这种适度的自由，我们放松定义（1）中要求，得到下式：

$$S_{safe} \Rightarrow (\forall B, C \Downarrow B \Rightarrow S \Downarrow B) \quad (2)$$

（这里的$S_{safe}$表示$S$的可能行为中不包含出错的行为）。换言之，如果$S$不出错，则$C$也不会出错；而且，$C$的所有可见行为均为$S$的可接受行为。

在CompCert的实践中以及本文剩余部分，我们所关注的源和目标语言均是确定的（程序行为的变化仅为对输入变化的响应，而不是程序的内部选择），并且执行环境也是确定性的（提交给程序的输入可以由此前的输出惟一确定）。在这些条件下，就只存在惟一的行为$B$使得$S \Downarrow B$，对$C$也一样。在这种情况下，很容易证明性质(2)等价于

$$\forall B \notin Wrong, S \Downarrow B \Rightarrow C \Downarrow B \quad (3)$$

（这里的$Wrong$是指出错行为的集合）。性质（3）通常比性质（2）更容易证明，因为我们可以通过对$S$的执行进行归纳来完成证明，这正是本文工作所采用的方法。

从形式化方法的观点看，我们真正关心的是编译器生成代码是否满足该应用程序的功能规范。假定该规范是通过描述可见行为的谓词$Spec(B)$的方式给出。如果$C$不出错($C_{safe}$)并且$C$的所有行为均满足规范，即$(\forall B, C \Downarrow B \Rightarrow Spec(B))$，我们就认为$C$满足规范并记为$C \models Spec$。我们所期望的编译器正确性性质是源代码符合其规范这一事实能够在编译过程中得到保持，而这一事实需要通过另行形式化地验证源代码程序而单独获得：

$$S \models Spec \Rightarrow C \models Spec \quad (4)$$

很容易看到，对于所有的规范$Spec$，性质（2）都蕴含性质（4）。而且，只要一次工作建立了性质（2），就不需要针对每一个感兴趣规范去逐个建立性质（4）。

作为性质（4）的一个特例而一直备受关注的重要特性是类型和存储安全性保持，这一性质可以总结为“如果$S$不出错，则$C$也不会出错”：

$$S_{safe} \Rightarrow C_{safe} \quad (5)$$

结合利用另外一个可靠检查得到的$S$良类型特性，性质（5）蕴含着“$C$执行中不会出现存储违规”的结论。而类型保持性编译[8]是通过另外的方式获得到同样结果：在假定$Validate(S,C)$是良类型的条件下，在一个可靠地类型系统中证明$C$是良类型的，则可以保证$C$不会出错。证明性质（2）或（3）也可提供相同的安全保证，而无需为目标语言和所有中间语言设计可靠的类型系统，或证明编译器的类型保持特性。

## 已验证、已确认和已认证的编译器

现在我们来讨论构建上一小节所讨论语义保持编译器的几种方法。后面我们使用$S \approx C$来表示上小节中（1）~（5）五个语义保持性质中的某一个，其中$S$为源程序，$C$为编译后代码。

经过验证编译器（Verified Compilers）。我们用一个全函数$Comp$作为编译器的模型，它要么从源代码得到编译后的代码（记为$Comp(S)=\mathrm{OK}(C)$），要么出现编译时错误（记为$Comp(S)=\mathrm{Error}$）。编译时错误对应于编译器无法生成代码的情况，比如输入的源代码不正确（语法错误、类型错误等），或者它超出了编译器的处理能力。如果编译器$Comp$具有一个形式化证明，而且该证明满足下述性质：

$$\forall S, C, Comp(S) = \mathrm{OK}(C) \Rightarrow S \approx C \quad (6)$$

则称其为“经过验证的”。换言之，对于一段源程序$S$，已验证编译器要么报告一个错误，要么生成一段具备我们所期望的正确性特性的代码。请注意，一个总是失败的编译器，（对于所有的$S$都有$Comp(S)=\mathrm{Error}$）也确实是一个经过验证的编译器，虽然它毫无用处。是否能够成功编译人们关心的源代码并不是正确性的问题，而是一个实现质量的问题，这个问题可以通过测试等非形式化方法来处理。从形式化验证的角度来看，更为重要的特征是编译器决不能不声不响地生成不正确的代码。

基于定义（6）的含义去验证一个编译器，实

质上就是以上一小节所定义的某一性质作为编译器的高层规范，将程序证明技术应用于编译器自身的源代码。

**用经过验证的确认器所做的翻译确认**（Translation Validation with Verified Validators）。在翻译确认方法[20,22]中不需要去验证编译器，而是用一个确认器作为编译器的补充。确认器实现布尔值函数$Validate(S,C)$，用于在编译后验证性质$S \approx C$。如果$Comp(S)$=OK($C$)且$Validate(S,C)$=true，那么就认为编译生成代码$C$是可信任的。确认可以通过多种方式完成，从对$S$和$C$的符号化解释和静态分析，一直到基于模型检测或自动定理证明生成验证条件等。性质$S \approx C$通常是不可判定的，因此确认器必定是不完备的，它们在无法得到$S \approx C$时将回答“false”。

翻译确认可以为所生成代码增加正确性方面的可信程度，但是确认过程本身不像经过验证的编译器那么强的形式化保证，因为确认器本身有可能是不正确的。为了排除这种可能性，我们要求确认器$Validate$是已验证的，为其附带一个具有下面性质的形式化证明：

$$\forall S, C, Validate(S,C) = true \Rightarrow S \approx C \qquad (7)$$

经过验证的确认器$Validate$和未经验证的编译器$Comp$组合起来，就可以提供和已验证编译器等价的形式化保证。事实上，考虑如下函数：

```
Comp'(S) =
match Comp(S) with
| Error → Error
| OK(C) → if Validate(S,C) then OK(C)
else Error
```

这个函数本身就是一个满足基于定义（6）的经过验证的编译器。如果确认器能够比编译器更小巧且更简单，那么基于翻译确认器的验证就是编译器验证的一种颇具吸引力的替代方案。

**携证明代码和认证的编译器**（Proof-Carrying Code and Certifying Compilers，PCC & CC）。携证明代码（PCC）方法[1,19]不求建立源程序和所生成程序之间的语义保持特性，而是着眼于生成可以独立检查的证据，该证据表明编译所生成的代码$C$满足诸如类型安全和存储安全等行为规范$Spec$。PCC使用了一种认证编译器（certifying compiler），可以用函数$CComp$描述。该函数要么失败，要么在生成代码$C$的同时给出性质$C \models Spec$的一个证明$\pi$。这里的证明$\pi$也称为证书（certificate），它可以由代码的使用方独立检查。这样就既不再需要信任代码的提供方，也不再需要形式化地验证编译器自身，惟一需要信任的东西就是位于用户方的检查器。该检查器是一个程序，用于检查证书$\pi$是否保证了性质$C \models Spec$。

与翻译确认的方法类似，认证编译方法只需形式化地验证用户方的检查器，就可获得等价于验证编译器提供性质（4）那样的正确性保证。与此对应，只要有关的验证是在某个逻辑里遵循“命题即类型、证明即程序”的模式做出来的，那么至少从理论上讲，利用已验证编译器就可以构造出一个认证编译器，具体构造方法详见文献[11]第2节。

## 编译遍的组合

编译器可以很自然地划分为若干个遍，它们之间通过中间语言而相互联系。幸运的是，已验证编译器也可以采用同样的方式分解。考虑两个已验证编译器$Comp_1$和$Comp_2$分别实现语言$L_1$到$L_2$和$L_2$到$L_3$的翻译。假定语义保持特性$\approx$满足传递性（前面讨论的性质（1）到（5）均满足），考虑$Comp_1$和$Comp_2$的包含错误传播的组合：

```
Comp(S) = match Comp1(S) with
| Error → Error
| OK(I) → Comp2(I)
```

不难证明，这个函数就是一个从语言$L_1$到$L_3$的已验证编译器。

## 小结

上述讨论的结论很简单，它也定义了我们验证CompCert编译器后端所遵循的方法。首先，假如编译器的目标语言具有确定性的语义，那么适用于编译器正确性证明的规范可以用定义（3）和（6）的

组合：

$\forall S, C, B \notin \text{Wrong}, \mathit{Comp}(S) = \text{OK}(C) \wedge S \Downarrow B \Rightarrow C \Downarrow B$

其次，经过验证的编译器应该由多个编译遍组合而成，就和通常编译器一样。当然，需要为所有中间语言提供合适的形式语义。

最后，对于每一个编译遍，我们可以有两个选择，要么直接去证明该遍的代码实现，要么采用非可信代码实施代码翻译，然后用一个已验证确认器来验证其翻译结果。后一方法可以大大减少需要验证的代码量。

## CompCert编译器概览

### 源语言

CompCert编译器的源语言称为Clight[5]，这是一个C语言大子集，与通常推荐用于编写关键嵌入式软件的C子集差不多。该语言支持几乎全部C数据类型（包括指针、数组、结构和联合类型），所有结构化控制（`if/then`、各种循环、`break`、`continue`、Java风格的`switch`）及包含递归函数和函数指针在内的函数支持。省略掉的主要是扩展精度类型的算术运算（`longlong`和`longdouble`）；`goto`语句；达夫设备(Duff 's device)之类的非结构化switch；将结构和联合作为参数传递和结果返回；以及带有变长参数的函数。Clight缺少的其他C语言特征通过在语法解析过程中的代码展开（de-sugaring）来支持，包括表达式的副作用（Clight表达式没有副作用）和块作用域的变量（Clight只支持全局变量和具有函数作用域的局部变量）。

Clight语义的形式化定义采用大步操作语义的方式给出。其语义是确定性的，同时还将ISO C标准中未加规范和未加定义的一些行为精确化，比如各种数据类型的大小、有符号算术操作溢出情况下的结果以及求值顺序等。另外一些C中未定义的行为都统一处理为“出错”行为，诸如引用空指针或者数组访问越界等。内存采用不相交块的集合来描述，每一个块内均可以通过字节偏移量访问，指针值是由块标识符和字节偏移量组成的数据对。这样，即使出现不兼容的指针类型的强制转换等情况，仍然可以精确描述指针的算术运算。

### 编译遍和中间语言

在CompCert编译器形式化验证的部分是从Clight的抽象语法到PPC抽象语法的翻译实现，其中PPC是PowerPC汇编语言的一个子集。如图1所示，该编译器包含14个遍，经过8种中间语言。图1中没有详细标出的是该编译器中未验证的部分：在前面

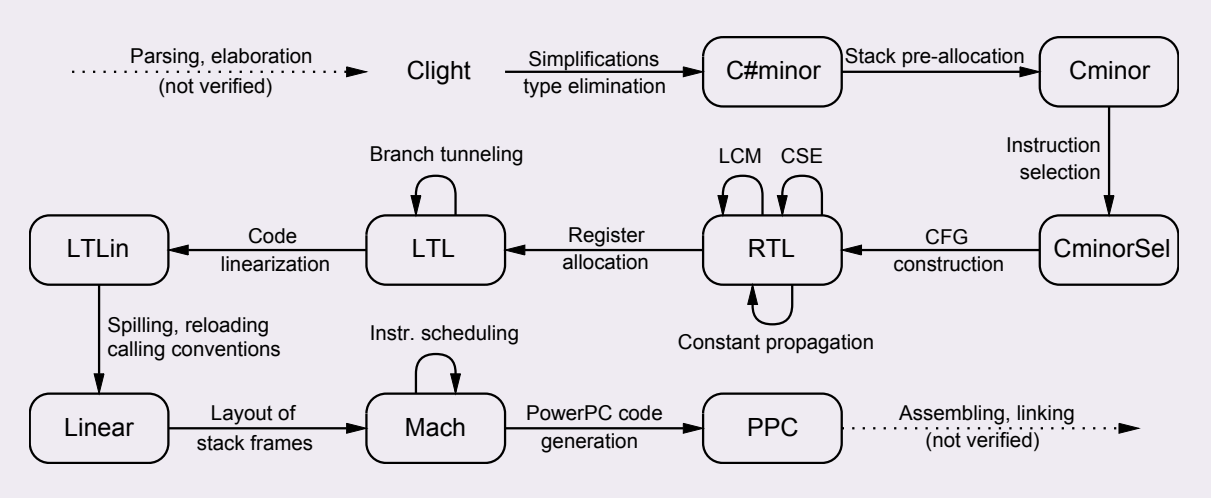



图1 编译遍和中间语言

的部分包括基于CIL（C Intermediate Language）库[21]的语法解析器、类型检查器和语法树化简器，它们完成从C源代码文件到Clight抽象语法树的生成工作。位于后面的是由PPC抽象语法树到具体汇编语法的输出程序，随后的可执行二进制文件生成部分则采用系统自带的汇编器和链接器。

编译器前端采用C#minor和Cminor两种中间语言，经过两遍翻译处理掉所有的C所专有的特性。C#minor是一种简化的无类型Clight语言变种，其中为整数、指针和浮点提供了不同的算术运算符，并且将C循环替换为无限循环附加若干语句块的形式，包含有从多层次闭块跳出的语句。第一遍编译据此翻译C循环并消除所有依赖于类型的行为：消除所有运算符重载，将内存读写以及地址计算都改为显式操作。第二种中间语言Cminor类似于C#minor，在其基础上略去了取地址运算(&)。Cminor函数内局部变量不在内存中存放，因而不能取其地址。但Cminor支持显示地在栈上分配函数活动记录里的数据。因此在从C#minor到Cminor的翻译过程中，需要识别出那些从不使用取地址操作的标量类型的本地变量，将其指派为Cminor的局部变量，并标记为稍后寄存器分配的考虑对象，而所有其他本地变量均放进活动记录中在栈上分配。

该编译器后端开始于指令选择遍，它识别出能够使用组合算术指令（如加立即数至累加器（add-immediate）、非与（not-and）、旋转及掩码（rotate-and-mask）等）和利用目标处理器提供的寻址方式等契机。这一遍通过自底向上地重写Cminor表达式的方式实现。其目标语言是CminorSel，这是一种与处理器相关的Cminor变种，它提供了更多的运算符、寻址方式和一组条件表达式（求值时就是为了得到其真值结果的表达式）。

接下来的遍将CminorSel翻译到RTL。RTL是一种经典的寄存器传输语言，其中的控制以控制流图（Control-Flow Graph，CFG）的形式给出。图中的每一个节点均代表一条以临时量（伪寄存器）为操作数的机器指令。RTL是一个便于开展基于数据流分析的优化的中间表示。当前已实现了两种此类优化：常数传播和公共子表达式消除，后者是通过扩展基本块上的值标号技术实现。此外，还单独开发了惰性代码迁移作为第三种优化，该优化将在不久集成进来。和前两种优化不同，惰性代码迁移是采用已验证确认器的方法实现的。[24]

这些优化之后是基于冲突图着色的寄存器分配[6]，这一遍的输出结果是LTL，这是一种类似于RTL的语言，不同之处是将原来的临时量全部替换为硬件寄存器或者是抽象的栈地址这样就把原有的控制流图线性化，得到的是带有显式标号、条件和无条件跳转的指令序列。接着，在引用分配到栈里的临时量的指令前后插入卸除和重取操作，在函数调用、入口处理和出口处理处插入一些转移（moves）指令以执行函数调用约定。最后，栈规划遍完成函数活动记录的布局、在该记录中确定抽象栈单元的偏移量、确定由被调函数保存（callee-save）的寄存器的偏移量、将对于抽象栈单元的引用替换为相对于栈指针的显式内存读写等等。

这时我们就得到了Mach中间语言，该语言的语义非常接近PowerPC汇编。这就可以实施基于表的指令调度或者轨迹（trace）调度了，这里再次使用了经过验证的确认器技术[23]。编译的最后一遍将Mach指令展开成为打包的PowerPC指令序列，完成对如条件寄存器等专用寄存器和非规整指令的特殊处理。目标语言PPC是PowerPC汇编语言的一个较大子集的精确模型，其中省略了CompCert未使用的部分指令和专用寄存器。

从纯粹编译的角度来看，CompCert确实是乏善可陈。所有的中间表示和编译遍均出自20世纪90年代初期的编译技术教科书。也许惟一值得关注的是它拥有如此众多的中间语言，而其中许多不过是另一语言的微小变异。但是从验证的角度来看，将具有微小差异的变种看做不同语言要比将其作为少数几个通用的中间表示的不同子集更为方便。

## 编译器的证明

CompCert的价值不在于其实现所用的编译技术，而在于如下的事实：其源语言、所有中间语言

和目标语言都有形式化定义的语义，其所有翻译和优化遍的语义保持特性都按照上一节提出的思路给出了证明。

这里的语义保持证明均采用Coq辅助证明工具实现了机械化。Coq实现了归纳与余归纳构造演算，这是一种功能强大的构造性的高阶逻辑，同时支持三种众所周知的规范书写风格：基于函数与模式匹配的风格，表示推导规则的归纳与余归纳谓词风格，以及常规的一阶逻辑谓词风格。在CompCert的开发中使用到所有上述三种风格，得到的规范和定理命题基本接近通常在编程语言研究论文中看到的形式。特别是，编译算法可以很自然地表达为函数，操作语义则基本上都定义为归纳谓词（推理规则）。Coq还提供了很多高级的逻辑特性，比如高阶逻辑、依赖类型和ML风格模块系统。我们在开发中偶尔也用到这些功能。例如，依赖类型使我们为数据结构加上逻辑不变量，而参数化模块则方便我们在几种静态分析之间重用一个通用的数据流方程求解器。

在Coq中的证明定理是一种交互式的过程：有一些判定过程能自动完成等式推论或者普瑞斯伯格（Presburger）算术，但大部分的证明则是由用户输入用以指导Coq完成证明任务的基本证明步（tactics）的序列。Coq在内部建立起许多证明项，然后用一个很小的核心验证器重新进行检查，这就大大增强了证明的可信程度。证明需要通过交互的方式开发，而证明脚本却可以批处理方式进行事后检查。

除注释和空行，整个Coq形式化定义和证明合计有42,000行Coq代码，工作量大约3人年。在这42,000行中，14%用于定义CompCert中实现的编译算法，10%用于规范相关语言的语义，其余76%则对应于编译器的正确性证明。每一个编译遍的规范及其正确性证明大约为1,500到3,000行Coq代码，相应的每一种中间语言的规范约为300到600行代码，而源语言Clight则需要1,100行。另外的10,000行Coq代码对应于所有语言和编译遍共享的基本框架，比如机器整数运算和存储模型的形式化定义等。

## 编译器的编程和运行

我们不仅用Coq作为得到语义保持证明的证明器，同时也将其用作编写CompCert编译器的所有已验证部分的编程语言。Coq的规范描述语言中包含一个很小的纯函数式语言，它基于归纳类型（ML或者哈斯克尔（Haskell）风格的树状数据类型）和模式匹配方法来处理递归函数。配合一些精巧的构思，这一语言足够编译器编写之需。编译教科书中可以看到的那些高度命令式的算法需要采用纯函数式的风格重写。我们采用的是基于平衡树的持久性数据结构，用以支持无需原地修改的高效数据更新。与此类似，单胞编程风格（monadic programming style）帮助我们用一种清晰的、组合的方式来描述异常和状态。

与采用常规的命令式语言实现编译器相比，这种非常规方法的主要优势在于，不再需要一种程序逻辑（如Hoare逻辑）将编译器代码和对应的逻辑规范联系起来。实现编译器的Coq函数是Coq逻辑里的“头等公民”，可以直接采用归纳、化简和等式推导等方法进行推理。

Coq提供了一种可以从函数式规范中自动生成Caml源代码的代码提取工具支持，我们采用这个工具来得到编译器的可执行代码。[15]将提取得到的源代码和语法解析器等手工实现的未验证的Caml代码合二为一，并用Caml编译器编译后，我们就得到一个具有标准cc风格命令行接口的编译器，它可以在Caml支持的所有平台上运行，所生成PowerPC代码的运行环境为Mac OS X（对其他运行环境的支持正在开发中）。

## 性能比较

为了评估CompCert生成代码的质量，我们将其和GCC4.0.1编译器在0、1和2优化级别的结果进行比较。很多标准基准测试程序集用到了CompCert不支持的C语言特性，因此我们只能自行构造一个很小的测试集，其中包含一些核心计算程序、加密原

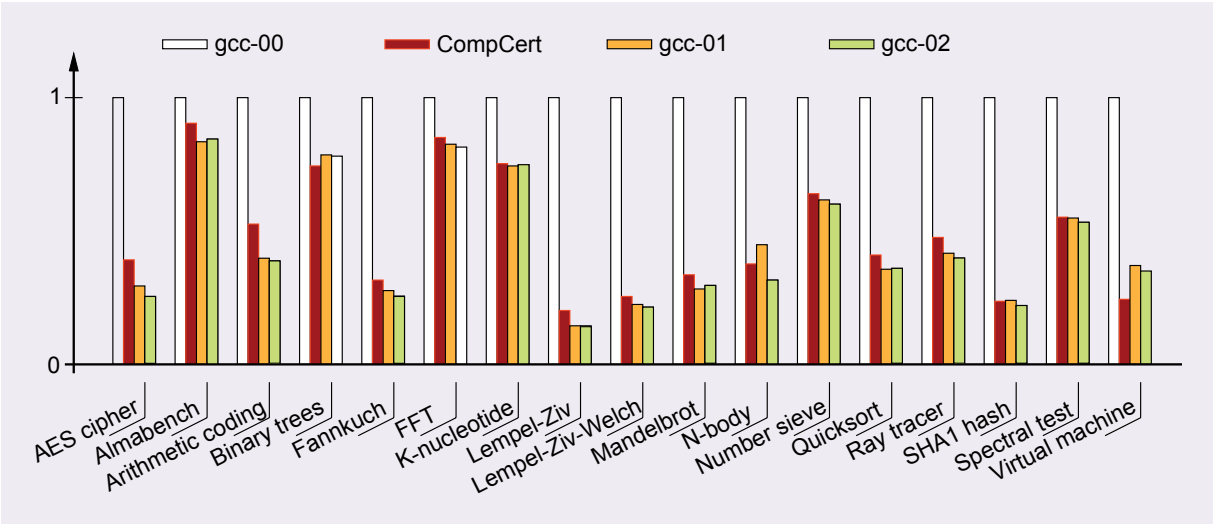



图2 生成代码执行时间比较

语、文本压缩、一个虚拟机解释程序，还有一个光线追踪程序。所有测试在一个2GHz主频的PowerPC 970“G5”处理器上完成。

时间比较结果如图2所示，CompCert所生成代码的运行效率超过不做优化的GCC所生成代码的两倍，可与GCC-O1和-O2优化结果匹敌。平均而言，CompCert代码仅比gcc-O1结果慢7%，而比gcc-O2结果慢12%。这个测试集还是太小，无法给出确切的结论，但这些结果雄辩地说明，虽然CompCert不能在高性能计算领域得奖，但其性能足以满足关键嵌入程序的需要。

CompCert的编译时间大约在gcc-O1编译时间的2倍之内，这是很合理的，这一情况说明，为了易于验证（很多短小的编译遍，没有命令式的数据结构等等）而带来的额外开销是可以接受的。

## 寄存器分配

作为已验证编译器遍的一个详细实例，本节介绍CompCert的寄存器分配遍及其正确性证明。

### RTL中间语言

寄存器分配在RTL中间表示上实施。RTL用抽象指令的控制流图表示函数，这种抽象指令大致对应于机器指令，但是其操作数为伪寄存器（也称为临时量）。每个函数可以无数量限制地使用伪寄存器，这些伪寄存器的值在函数调用过程中保持不变。下面我们用$r$和$l$分别表示伪寄存器和控制流图节点标号。

指令（Instructions）：

| | |
|---|---|
| $i$::=nop($l$) | 空操作（转标号$l$） |
| \|op($op, \vec{r}, r, l$) | 算术操作 |
| \|load($k, mode, \vec{r}, r, l$) | 内存加载 |
| \|store($k, mode, \vec{r}, r, l$) | 内存写入 |
| \|call($sig, (r \mid id), \vec{r}, r, l$) | 函数调用 |
| \|tailcall($sig, (r \mid id), \vec{r}$) | 函数尾调用 |
| \|cond($cond, \vec{r}, l_{true}, l_{false}$) | 条件分支 |
| \|return\|return($r$) | 函数返回 |

控制流图（Control-flowgraphs）：

| | |
|---|---|
| $g$::=$l \mapsto i$ | 有限映射 |

内部函数（Internalfunctions）：

| | |
|---|---|
| $F$::={name=$id$; sig=$sig$; | |
| params=$\vec{r}$; | 参数 |
| stacksize=$n$; | 栈数据块大小 |
| entrypoint=$l$; | 第一条指令的标号 |
| code=$g$} | 控制流图 |

外部函数（Externalfunctions）：

$Fe$::={name=$id$; sig=$sig$}

每一条指令从一组伪寄存器$\vec{r}$得到参数，如果有结果则存放于另一个伪寄存器$r$中。此外，指令还可能带有指明其后继指令的标号$l$。这里的指令包括算术运算$op$（包括表示寄存器到寄存器复制的op(move, $r$, $r'$, $l$)作为特例）、内存加载和存取（通过将寻址方式$mode$应用于寄存器$\vec{r}$获得内存地址，对该地址开始的$k$个单元进行操作）、条件跳转（含两个后继指令）、以及函数调用、尾调用和函数返回。

RTL程序由一个命名的内部函数和一个命名的外部函数集合组成，其中内部函数用RTL描述，说明其控制流图、控制流图的入口点和参数寄存器；而外部函数只声明但不定义，用于模拟输入/输出操作以及其他系统调用。函数和调用指令附带着签名$sig$，描述其参数和返回值的数目和寄存器类属（整型或者浮点）。

RTL的动态语义采用小步操作语言的风格，用带标转移系统的形式描述。谓词$G\vdash S \xrightarrow{t} S'$表示从状态$S$执行一步转移到状态$S'$，全局环境$G$将函数指针和函数名映射到函数定义，轨迹$t$记录该执行步中执行的输入-输出事件：调用外部函数时，$t$记录函数名称、参数和调用结果，除此之外，其他轨迹均为空($t=\varepsilon$)。

执行状态$S$形式为$S(\sum, g, \sigma, l, R, M)$，其中$g$是当前执行函数的控制流图；$l$是该函数内的当前程序执行位置；$\sigma$是包含函数的活动记录的存储块；寄存器状态$R$将伪寄存器映射为它们的当前值（需要区别对待32位整数，64位浮点数以及指针组成的辨识联合(discriminated union)）；类似地，存储状态$M$将<指针，存储量>对映射为值，映射中需要考虑多字节存储块之间的相互重叠问题[14]；最后，$\sum$描述调用栈，记录尚未结束的函数调用的$(g, \sigma, l, R)$信息。调用状态和返回状态用于描述函数返回和调用，它们的形式与执行状态略微不同，这里不再详述。

为使能让读者对RTL语义有些直观感觉，这里分给出算术操作和条件跳转两条指令的单步转移关系规则的定义：

$$\frac{g(l)=\mathrm{op}(op, \vec{r}, r, l')\quad \mathtt{eval\_op}(G, \sigma, op, R(\vec{r}))=v}{G\vdash S(\sum, g, \sigma, l, R, M) \xrightarrow{\varepsilon} S(\sum, g, \sigma, l', R\{r \leftarrow v\}, M)}$$

$$\frac{g(l)=\mathtt{cond}(cond, \vec{r}, l_{true}, l_{false})\quad l'=\begin{cases} l_{true} & \text{if } \mathtt{eval\_cond}(cond, R(\vec{r}))=\text{true} \\ l_{false} & \text{if } \mathtt{eval\_cond}(cond, R(\vec{r}))=\text{false} \end{cases}}{G\vdash S(\sum, g, \sigma, l, R, M) \xrightarrow{\varepsilon} S(\sum, g, \sigma, l', R, M)}$$

## 寄存器分配算法

寄存器分配遍的目标是将在原始RTL代码中标为$l$处出现的可以无限使用的伪寄存器$r$替换掉，或者将其替换为硬件寄存器（可用数量不多，个数固定）或者替换为活动记录中的抽象栈单元（无个数限制）。由于硬件寄存器的访问速度要远远高于栈单元的访问速度，因此必须最大限度地使用硬件寄存器。寄存器分配需要考虑的其他方面，比如插入栈单元操作的溢出和重取指令，则留给其他后续遍处理。

寄存器分配开始于采用逆向数据流分析的标准活跃分析，针对活跃信息的数据流方程形如：

$$LV(l) = \cup\{T(s, LV(s)) | s \text{ successor of } l\} \qquad (8)$$

转换函数$T(s, LV(s))$算是程序点$s$之前的活跃伪寄存器集合，描述为该点之后活跃伪寄存器集合$LV(s)$的函数。例如，假定$s$点的指令为$op(op, \vec{r}, r, s')$，由于$r$在当前点重新定值，因此$r$的结果将不再活跃，而由于$\vec{r}$为当前点所引用，有$T(s, LV(s))=(LV(s)\{r\})\cup\vec{r}$，因此参数$\vec{r}$开始活跃。不过，如果之后的$r$非活跃($r \notin L(s)$)，则该指令是死代码并将在稍后删除，因此我们可以采用$T(s, LV(s))=LV(s)$来代替上式。

可以用基尔代尔（Kildall）工作表算法来迭代求解该数据流方程，CompCert给出了基尔代尔算法的一种通用实现及其正确性证明，该实现也适用于其他相关优化遍。这一算法的结果是一个由程序点到活跃寄存器集合的映射$LV$，已经证明对于$l$的所有后继指令标号$s$，该映射均满足正确性条件$LV(l) \supseteq T(s, LV(s))$。我们只证明不等式，而不证明标准数据流方程（8），原因是我们所感兴趣的只是解的正确性而不是解的最优性。

接着采用柴廷（Chaitin）规则[6]来建立以伪寄存器为节点的冲突图，并证明了图中包含所有必须的冲突边。典型情况是，如果在同一个程序点上两个伪寄存器$r$和$r'$同时活跃，则图中必定包含连接$r$和$r'$的边。冲突的表现形式为“某两个伪寄存器冲突”或者“此伪寄存器和彼硬件寄存器冲突”，后者用来保证跨函数调用活跃的伪寄存器不被分配为调用者保存的硬件寄存器。首选边（“这两个伪寄存器最好分配为同一个位置”或者“这个伪寄存器最好分配为这个位置”）也记录下来。虽说这些不影响寄存器分配的正确性，但影响到寄存器分配的质量。

寄存器分配的核心工作是基于有关约束的冲突图着色，为每个节点$r$指派一种“颜色”$\varphi(r)$。这里的颜色要么是一个硬件寄存器，要么是一个栈单元，约束条件是必须为任何两个由冲突边连接的节点指派不同的颜色。

我们采用乔治（George）和阿贝尔（Appel）[9]的启发着色方法。由于该启发式方法的正确性很难直接证明，因此我们先采用未经验证的Caml代码实现该方法，然后用一个由Coq编写并证明的简单验证器来对其结果进行事后确认。与很多NP难问题一样，图着色是一个典型例子，事后确认比直接证明其正确性简单得多。着色结果$\varphi$的正确性条件为：

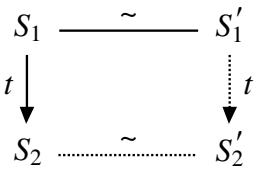
1. $\varphi(r)\neq\varphi(r')$若$r$和$r'$冲突
2. $\varphi(r)\neq l$若$r$和$l$冲突
3. $\varphi(r)$和$r$拥有相同的寄存器类别(整型或浮点)

这些条件都用由Coq书写的布尔值函数来检查，并且已经证明这些函数是上述条件的判定过程。如果检查出错，就表明外部图着色函数存在错误，此时将终止编译。

最后的工作是重写原始的RTL代码，每一个伪寄存器$r$的引用都将替换为该寄存器位置$\varphi(r)$的引用。同时还将执行变量合并和死代码删除优化：如果结果$r$在$l$之后不再活跃，则无副作用的指令$l$:op($op$, $\vec{r}$, $r$, $l'$)或$l$:load($k$, $mode$, $\vec{r}$, $r$, $l'$)将替换为空指令$l$:nop($l'$)（死代码删除）；类似地，如果$\varphi(r_d)=\varphi(r_s)$，则寄存器移动指令$l$:op($move$, $r_s$, $r_d$, $l'$)将替换为空指令$l$:nop($l'$)（变量合并）。

## 证明语义保持性

为了证明程序变换的语义保持特性，贯穿CompCert项目的标准技术是建立如下所示的模拟关系图：原始程序的每一个单步转移一定对应于变换后程序的一个具有相同观察效果（在这里是指具有相同的输入-输出操作轨迹）的转移序列，并保持以不变量形式给出的原始程序和变换后程序的执行状态之间的一个二元关系 ~ 。在寄存器分配中，每一个原始转移步恰好对应于变换后的一个转移步，其结果就得到如下的“定步长(Lock-step)”模拟关系图。




(实线表示假设，而虚线表示结论。)进而，如过将不变量 ~ 看做初始状态和终止状态之间的关系，则本模拟关系图表示：原始程序的任何执行都对应着变换后程序的一个执行，二者具有完全相同的可观察行为的轨迹。语义保持特性因而得证。

证明模拟关系的要点是 ~ 关系的定义。那么，两个状态$S(\sum, g, s, l, R, M)$和$S(\sum', g', s', l', R', M')$相关的条件是什么呢？直观来讲，由于寄存器分配过程中程序结构和控制流保持不变，因此控制点$l$和$l'$一定完全相等，同时控制流图$g'$一定是$g$经过上一节所述某种寄存器分配$\varphi$的变换结果。同样地，因为寄存器分配过程中内存存储和分配保持不变，因此内存状态和栈指针一定保持不变，即$M'=M$且$s'=s$。

原始程序的寄存器状态$R$和变换后程序的存储状态$R'$之间的关系并不那么明显，假定每个伪寄存器$r$映射到$\varphi(r)$，我们可以简单地要求对所有$r$有$R(r)=R'(\varphi(r))$。但是这个要求太强了，它从根本上排除了两个活跃范围无重叠的伪寄存器之间共享同

一个寄存器分配单元的可能性。为了得到正确的要求条件，我们需要认真考虑一个伪寄存器在程序点$l$处活跃和非活跃在语义上的意义。对于在$l$点非活跃的伪寄存器，要么在此之后不会读取$r$的值，要么$r$必定在下一次读取之前重新定值，因此$r$在$l$处的值不会影响程序的执行。所以，在设定寄存器及其分配单元之间的对应关系时，我们可以安全地忽略在当前点$l$非活跃的所有寄存器。要求下述条件就足够了：

对所有在$l$点活跃的伪寄存器$r$（而不是像上面要求的对所有$r$），有$R(r)=R'(\varphi(r))$。

一旦建立状态之间的模拟关系，证明上面的模拟关系图就变成了检查各种RTL语义转移规则的程序性工作。在检查过程中，将会惊喜地发现定义活跃性的数据流不等式和构建冲突图所遵循的柴廷规则，也就是任意情况下寄存器状态$R$、$R'$之间不变量保持的最小充分条件。

## 结论与展望

本文所述的CompCert实验仍在进行中，还有很多亟待解决的问题：比如处理更大的C语言子集，例如将goto纳入其中，加入更多的优化方法并对其加以证明，以PowerPC之外的其他处理器作为目标器件，扩展语言保持证明以支持共享存储式的并发等等。尽管如此，我们的初步结论雄辩地证明，在今天很有限的证明工具的条件下，采用基本的语义和算法，形式化验证真实可用编译器的初步目标是完全可以实现的。我们所用的技术和工具远非完善，但已经足够实现本项目的目标，而更好的自动证明工具、更高级的语义以及更为现代的中间表示必将极大地减少证明的工作量。

回首本项目的成果，可以看到，我们并没有彻底解决所有与编译器正确性相关的问题，而仅仅是将从原来需要信任整个编译器的问题归结到目前只要信任以下的几个小的部分：

1. 源（Clight）和目标（PPC）语言的形式语义。

2. 编译器中未获验证的部分，包括基于CIL的解析器、汇编器和链接器。

3. 用于生成编译器的执行代码的编译工具链，包括：Coq中的代码提取工具、Caml语言的编译器和运行时系统（这个工具链中的任何错误都将彻底摧毁正确性证明得到的正确性保证）。

4. Coq辅助证明工具本身（Coq实现的错误或者是Coq逻辑的不一致性同样会毁掉所有证明）。

问题（4）可能是最不需要担心的，正如哈尔斯（Hales）所言[10]，证明辅助工具机械化检查过的证明所生成证明项的可信程度，要比哪怕是最仔细的手工检查数学证明都要高好几个数量级。

针对第（3）个问题，CompCert项目中正在进行的工作将研究“从Coq的代码提取工具和Mini-ML语言（专门用于代码提取的简单函数式语言）到Cminor编译器的形式化验证”的可行性。毫无疑问地，这些工作和CompCert后端一起，将最终构成Coq中编码和验证的程序的可信任执行路径，类似于CompCert自身，进一步的自展技术将大大提升其可信程度。

问题（2）关注的是未经验证的CompCert模块，这些毫无疑问可以通过重新实现和证明相关编译遍的方式来解决。解析器的语义保持特性很难定义，更不用说去证明：如果说解析所生成的抽象语法树的语义，怎么去说明程序的具体词法的语义呢？不过，在解析遍之后基于CIL的几个加工步骤是适合于形式化证明的。同理，即便不那么令人振奋，汇编器和链接器的正确性证明也是可行的。

或许最令人头疼的是问题（1），如何才能确保形式化语义符合语言标准和编程常规？由于这种问题中的语义和完整编译器的关系并不大，所以由专家做的人工审查，以及在语义的可执行形态上实施的测试将能提供合理（而非形式化）的信心。另外一种方式是证明与另外独立开发的形式语义之间的联系，如作为程序的演绎证明工具的公理语义（一个例子见文献[2]）。进一步说，本方法只是向着一个更为宏伟和长远的目标的第一步，这个目标就是采用形式化方法认证开发、验证和运行关键软件相关的验证工具、代码生成器、编译器和运行时

系统。■

ACM（Association for Computint Machinery）是国际著名的由计算机教育人士、研究者、专业人士和学生所组成的专业组织。加入ACM（http://china.acm.org）可以获取ACM的期刊与杂志、访问ACM数字图书馆以及在线书籍，并在注册ACM国际会议的时候获得注册费优惠。ACM特别为中国的会员制订了特殊优惠的会费以及获得高级会员资格的优惠条件。ACM和CCF有密切的合作关系。

## 致谢

作者感谢S. Blazy, Z. Dargaye, D. Doligez, B. Grégoire, T. Moniot, L.Rideau,和B. Serpette等人为CompCert开发所作出的贡献，感谢A. Appel, Y. Bertot,E. Ledinot, P. Letouzey,和G. Necula等人的建议、反馈和帮助。本文工作得到Agence Nationale de la Recherche, 编号ANR-05-SSIA-0019的资助。

作者：

**泽维尔·乐华**

法国国家信息自动化研究所

**译者 董 渊**

CCF系统软件专委会委员。清华大学计算机系副教授。主要研究领域为操作系统，编译系统，基于语言的可信软件。dongyuan@tsinghua.edu.cn

**译校 王生原**

CCF高级会员。清华大学计算机系副教授，主要研究领域为程序设计语言与系统，Petri网应用。
wwssyy@tsinghua.edu.cn

**译校 郭 宇**

中科大－耶鲁高可信软件联合研究中心博士后研究人员。主要研究领域为程序验证与软件安全。
guoyu@mail.ustc.edu.cn

## 参考文献

[1] Appel, A.W. Foundational proofcarrying code. In Logic in Computer Science 2001 (2001), IEEE, 247~258

[2] Appel, A.W., Blazy, S. Separation logic for small-step Cminor. In Theorem Proving in Higher Order Logics, TPHOLs 2007, volume 4732 of LNCS (2007), Springer, 5~21

[3] Bertot, Y., Castéran, P. Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions (2004), Springer

[4] Blazy, S., Dargaye, Z., Leroy, X. Formal verification of a C compiler front-end. In FM 2006: International Symposium on Formal Methods, volume 4085 of LNCS (2006), Springer, 460~475

[5] Blazy, S., Leroy, X. Mechanized semantics for the Clight subset of the C language. J. Autom. Reasoning (2009). Accepted for publication, to appear

[6] Chaitin, G.J. Register allocation and spilling via graph coloring. In 1982 SIGPLAN Symposium on Compiler Construction (1982), ACM, 98~105

由于版面原因，查阅参考文献[7～24]请访问：www.ccf.org.cn

## 中国计算机学会YOCSEF2010年4月活动计划

**4月10日 郑州**
**论坛：企业信息化建设管理与策略**

**4月10日 上海**
**学术报告会：无线传感器网络（二）**

**4月17日 济南**
**论坛：高校实践教学探索**

**4月17日 武汉**
**学术报告会：对等计算**

**4月23日 北京**
**学术报告会：移动搜索技术与趋势**