形式验证**内政部：10.1145/1538788.1538814**

一个真实的编译器

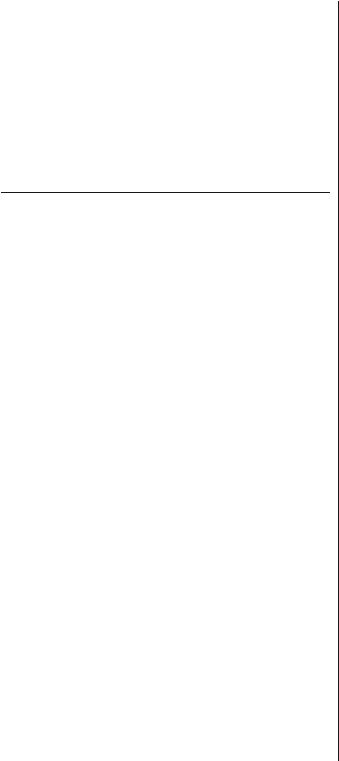
作者：Xavier Leroy

# 摘要

**本文报道了用Coq证明助手对C语言的一个大子集Clight到PowerPC汇编代码的编译器CompCert进行开发和形式化验证（语义保持证明），并证明了它的正确性。这种经过验证的编译器在关键软件及其形式验证的上下文中是有用的：编译器的验证保证了源代码上证明的安全属性也适用于可执行编译代码。**

## 1.简介

你能相信你的编译器吗？编译器通常被认为是语义透明的：编译后的代码应该按照源程序的语义来运行。然而，编译器，尤其是优化编译器是执行复杂符号转换的复杂程序。尽管进行了密集的测试，编译器中还是会出现bug，导致编译器在编译时崩溃，或者更糟糕的是，为正确的源程序生成错误的可执行文件。

对于只通过测试验证的低保证软件，编译器错误的影响很小：被测试的是编译器生成的可执行代码；严格的测试应该暴露编译器引入的错误以及源程序中已经存在的错误。然而，请注意，编译器引入的bug是出了名的难以暴露和追踪的。对于安全关键、高保证的软件来说，情况发生了巨大的变化。在这里，通过测试进行的验证达到了它的极限，需要通过使用正式的方法来补充甚至替代，例如模型检查、静态分析和程序证明。这些正式的验证工具几乎普遍应用于程序的源代码。编译器中用于将经过正式验证的源代码转换为可执行文件的bug可能会使所有通过使用正式方法而痛苦获得的保证失效。在将来，正式方法通常应用于源程序的地方，编译器可能会成为从规范到可执行文件的链条中的薄弱环节。安全关键软件行业意识到这些问题，并使用各种技术来缓解这些问题，例如在关闭所有编译器优化之后，对生成的汇编代码执行手动代码检查。这些技术不能完全解决这些问题，而且在开发时间和程序性能方面代价高昂。

一个明显更好的方法是将形式化方法应用于编译器本身，以确保它保留源程序的语义。在过去的5年里，我们一直致力于开发一个实际的、经过验证的编译器CompCert。通过验证，我们指的是一个编译器，它伴随着一个经过机器检查的语义保留属性的证明：生成的机器代码的行为符合源程序的语义规定。我们所说的“现实”是指可以在关键软件的生产环境中实际使用的编译器。也就是说，它编译了一种常用于关键嵌入式软件的语言：既不是Java，也不是ML，也不是汇编代码，而是C语言的一个很大的子集。它为嵌入式系统中常用的处理器生成代码：我们选择PowerPC是因为它在航空电子设备中很流行。最后，编译器必须生成足够高效和紧凑的代码，以满足关键嵌入式系统的需求。这意味着一个多程编译器具有良好的寄存器分配和一些基本的优化功能。

证明编译器的正确性决不是一个新的想法：第一个这样的证明是在1967年发表的（用于编译算术表达式到堆栈的机器代码），并在1972年进行了机械验证，从玩具语言的单通道编译器到复杂的代码优化。在CompCert实验中，我们将这项工作一直进行到一个完整编译链的端到端验证，从结构化命令式语言到汇编代码，再到八种中间语言。在对CompCert进行验证时，我们发现许多执行的非优化翻译虽然在编译器文献中经常被认为是显而易见的，但要正式证明其正确性却异常棘手。16178

本文简要介绍了CompCert编译器及其使用Coq证明助手的机械化验证。这个编译器经典地由两部分组成：一个前端将C的Clight子集翻译成一种称为Cminor的低级结构化中间语言，另一个后端从Cminor生成PowerPC汇编代码。克莱特的详细描述可以在布莱兹和勒罗伊中找到；Blazy等人的编译器前端设计。；以及Leroy中的编译器后端。Coq开发的完整源代码在Web上有广泛的评论。3, 75411, 1312

本文的其余部分组织如下。第2节比较并形式化了几种在编译结果中建立信任的方法。第3节

本文的前一个版本发表在第33届编程语言原理研讨会论文集上。纽约州ACM，2006年。

描述了CompCert编译器的结构、性能，以及如何使用Coq证明助手来证明其正确性，并对大部分程序进行了编程。由于篇幅不够，我们不会详细说明每个编译过程的形式验证。然而，第4节提供了编译器的一个关键步骤：寄存器分配的这种验证的技术概述。最后，第5节提出了初步结论和今后工作的方向。

## 2.可信编译方法

### 2.1. 语义保留的概念

考虑由编译器生成的源程序S和编译程序C。我们的目的是证明S的语义在编译过程中得到了保留。为了使语义保留的概念更精确，我们假设源语言和目标语言的给定语义将可观察的行为B与S和C联系起来。我们写的是⇓ 意思是程序S以可观察的行为B执行。我们在CompCert中观察到的行为包括终止、发散和“出错”（调用可能崩溃的未定义操作，例如访问超出边界的数组）。在所有情况下，行为还包括在程序执行期间执行的输入-输出操作（系统调用）的跟踪。因此，行为准确地反映了程序的用户，或者更一般地说，程序与外界交互时，可以观察到什么。

编译过程中语义保留的最强概念是源程序S和编译代码C具有完全相同的可观察行为：

                                     ∀, S⇓ ⇔ C⇓ (1)*BBB*

概念（1）太强，无法使用。如果源语言不是确定性的，编译器可以选择源程序可能的行为之一(例如，C编译器从C规范允许的几个顺序中为表达式选择一个特定的求值顺序。）在这种情况下，C的行为将比S少。此外，编译器优化可以优化“出错”行为。例如，如果S在整数被零除时出错，但编译器由于其结果未使用而消除了此计算，则C不会出错。为了说明编译器中的这些自由度，我们将定义（1）放宽如下：

S⇒ (∀, C⇓ ⇒ S⇓ ) (2)**安全的** *BBB*

（这里，S表示S的所有可能行为都不是“出错”行为）换句话说，如果S没有出错，那么C也不会出错；此外，C的所有可观察行为都是S的可接受行为。**安全的**

在CompCert实验和本文的剩余部分中，我们关注确定性的源语言和目标语言（程序仅在响应不同的输入时改变其行为，而不是因为内部选择），以及确定性的执行环境（给程序的输入由其先前的输出唯一地确定）。在这些条件下，只有一个行为B使得S⇓ B、 C也是一样。在这种情况下，很容易证明性质（2）等价于

                             ∀∉, ⇓ ⇒ C⇓ (3)*B***错***SBB*

属性（3）通常比属性（2）更容易证明，因为在执行S时可以通过归纳法进行证明。这就是我们在这项工作中采取的方法。**错**

从形式方法的角度来看，我们真正感兴趣的是编译后的代码是否满足应用程序的功能规范。假设这些规范是作为可观察行为的谓词规范（B）给出的。我们说C满足规范，然后写CSpec，如果C不能出错（C），并且B的所有行为都满足Spec(∀, C⇓ B⇒ 规范（B））。编译器的预期正确性特性是，它保留了源代码S满足规范的事实，这一事实是通过对S的形式验证分别建立的：**安全的***B*

S⇒ C*规格*(4)*规格*

很容易证明属性（2）意味着所有规范的属性（4）。因此，一次性地建立属性（2）可以避免我们为每个感兴趣的规范建立属性（4）。

财产（4）的一个特殊情况，具有相当重要的历史意义，是对类型和记忆安全的保护，我们可以概括为“如果S不出错，C也不出错”：

## 安全吗⇒ C安全（5）

再加上一个单独的检查，即S在sound-type系统中的类型是否正确，属性（5）意味着C执行时没有内存冲突。保类型编译通过不同的方法得到了这种保证：在S是好类型的假设下，在一个健全的类型系统中证明C是好类型的，确保它不会出错。已证明的属性（2）或（3）提供了相同的保证，而不必为目标语言和中间语言配备健全的类型系统，并证明编译器的类型保留。18

**2.2. 已验证、已验证、正在验证的编译器**我们现在讨论几种方法来确定编译器保留已编译程序的语义，如第2.1节所述。在下面，我们写下≈ C、 其中S是源程序，C是编译代码，表示第2.1节的语义保留属性（1）到（5）之一。

***已验证的编译器。***我们将编译器建模为从源程序到编译代码（writecomp（S）=（C））或编译时错误（writecomp（S）=）的总函数Comp。编译时错误对应于编译器无法生成代码的情况，例如源程序不正确（语法错误、类型错误等），但也超出了编译器的容量。如果编译器Comp附带以下属性的正式证明，则称其为已验证：

                        ∀, C、 （S）=（C）⇒ ≈ C（6）*S公司***好的***S*

换句话说，经过验证的编译器要么报告错误，要么生成满足所需正确性属性的代码。请注意，一个总是失败的编译器（Comp（S）=for all S）确实是经过验证的，尽管没有用。编译器是否成功地编译了感兴趣的源程序不是一个正确性问题，而是一个实现质量问题，这是由非正规方法（如测试）解决的。从形式验证的角度来看，重要的特性是编译器从不自动生成错误的代码。**错误**

从定义（6）的意义上验证编译器相当于对编译器源应用程序验证技术，使用第2.1节中定义的属性之一作为编译器的高级规范。

***使用验证的验证器进行翻译验证。***在翻译验证方法中，编译器不需要验证。相反，编译器由一个验证器来补充：一个验证属性S的布尔值函数Validate（S，C）≈ 后面的。如果Comp（S）=（C）和Validate（S，C）=，则编译的代码C被认为是可信的。验证可以通过几种方式进行，从S和C的符号解释和静态分析到生成验证条件，然后进行模型检查或自动定理证明。财产≈ C一般来说是不可判定的，验证器必然是不完整的，如果不能建立验证，则应该回答≈ C.20, 22

翻译验证对编译代码的正确性产生了额外的信心，但它本身并没有提供像经过验证的编译器所提供的那样强大的形式保证：验证程序本身可能是不正确的。为了排除这种可能性，我们说，如果验证器Validate附带以下属性的正式证明，那么它将被验证：

                    ∀, C、 验证（S，C）=⇒ ≈ C（7）*S***是的***S*

验证的验证器Validate与未验证的编译器Comp的组合确实提供了与验证的编译器提供的一样强大的形式保证。实际上，请考虑以下函数：

*公司*′（S）=（S）**匹配***公司*

## *|* 错误→错误

*|* **好的，请注意**（三）→ *验证*（S、C）（三）

此函数是定义（6）意义上的已验证编译器。因此，翻译验证程序的验证是编译器验证的一个有吸引力的替代方案，前提是验证程序比编译器更小、更简单。

***带证明的代码和证明编译器。***proof-carriering-code（PCC）方法并不试图在源程序和某些编译代码之间建立语义保留。相反，PCC侧重于生成编译代码C满足行为规范（如类型和内存安全）的可独立检查的证据。PCC使用了一个认证编译器，这是一个函数CComp，它要么失败，要么返回编译后的代码C和证明π 财产C1, 19说明。证据π, 也称为证书，可由代码用户独立检查；不需要信任代码生成器，也不需要正式验证编译器本身。基础设施中唯一需要信任的部分是客户端检查器：检查π 包含属性C规格。

与翻译验证一样，只需正式验证客户端检查器，即可获得与编译器验证属性（4）所获得的保证一样强的保证。对称地，验证编译器至少在理论上可以从验证过的编译器构造出来，前提是验证是在遵循“命题作为类型，证明作为程序”范式的逻辑中进行的。施工详见勒罗伊。11，第2节

### 2.3. 编译过程的组成

编译器自然地被分解成几个通过中间语言进行通信的过程。幸运的是，经过验证的编译器也可以用这种方式进行分解。考虑两个已验证的编译器Comp和Comp，分别来自语言L到L和L到L。假设语义保留属性≈ 是可传递的(这对于第2.1节的属性（1）到（5）是正确的。）考虑Comp和Comp的错误传播组合：12122312

*公司*（S） =（秒）**匹配***公司*1

## *|* 错误→错误

*|* **好的**（一）→*公司*2（一）

证明这个函数是一个从L到L的经过验证的编译器并不重要。13

### 2.4. 总结

这个讨论的结论很简单，定义了我们用来验证CompCert编译器后端的方法。首先，如果编译器的目标语言具有确定性语义，则编译器正确性证明的适当规范是定义（3）和（6）的组合：

∀, C、 B类∉, (S） =（C）∧ ⇓ ⇒ C⇓ *S* **错了***公司SBB*

第二，按照通常的做法，经过验证的编译器可以构造为编译过程的组合。然而，所有中间语言都必须给出适当的形式语义。

最后，对于每个过程，我们可以选择是验证实现此过程的代码，还是通过不受信任的代码执行转换，然后使用验证的验证器验证其结果。后一种方法可以减少需要验证的代码量。

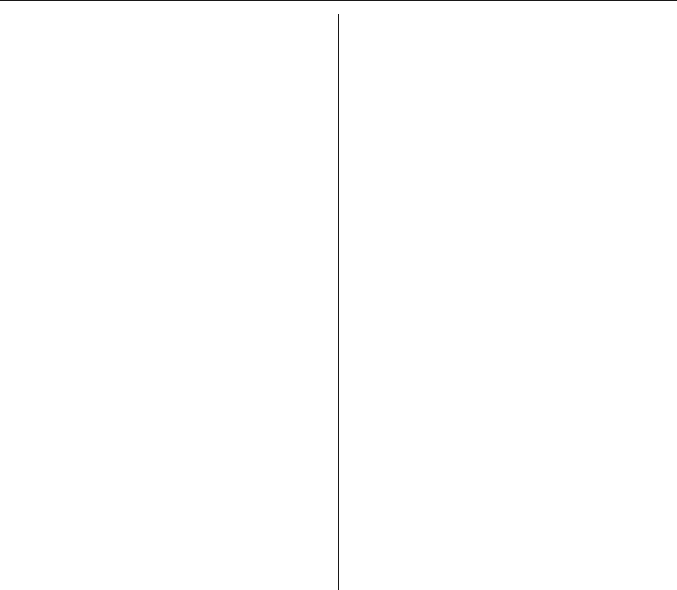
### 3.comPceRt编译器概述

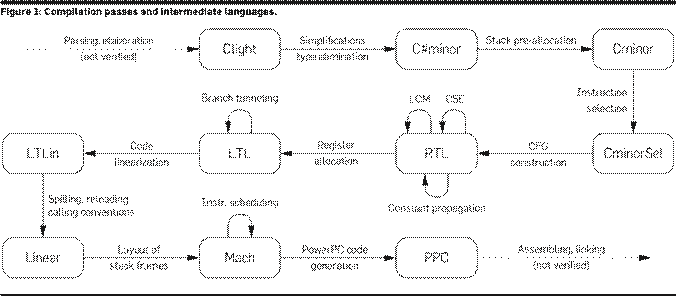
#### 3.1. 源语言

CompCert编译器的源语言Clight是C编程语言的一个很大的子集，与通常推荐用于编写关键嵌入式软件的子集相当。它支持几乎所有的C数据类型，包括指针、数组和类型；所有结构化控件（、循环、、Java样式）；函数的全部功能，包括递归函数和函数指针。主要遗漏是扩展精度算法（和）；声明；非结构化形式，如达夫装置；按值传递参数和结果；参数个数可变的函数。Clight中缺少了C的其他特性，但在解析过程中通过代码扩展（去糖化）得到了支持：表达式内的副作用（Clight表达式没有副作用）和块范围变量（Clight只有全局变量和函数局部变量）。5**structunionif/ThenBreakContinueSwitchLongDoubleGotoSwitchStructUnion**

Clight的语义是以大步操作风格形式化定义的。语义是确定性的，精确地定义了isoc标准中未定义或未定义的一些行为，如数据类型的大小、溢出时有符号算术运算的结果以及求值顺序。其他未定义的C行为始终会变成“出错”行为，例如取消对空指针的引用或访问超出界限的数组。内存被建模为不相交块的集合，每个块通过字节偏移量被访问；指针值是块标识符和字节偏移量的对。这样，即使在不兼容指针类型之间存在强制转换的情况下，指针算法也可以精确建模。

**3.2. 编译过程和中间语言**CompCert编译器的正式验证部分将Clight抽象语法转换为PPC抽象语法，PPC是PowerPC汇编语言的一个子集。如图1所示，编译器由经过8种中间语言的14个过程组成。在图1中没有详细说明的是编译器中尚未验证的部分：上游是一个解析器、类型检查器和简化器，它从C源文件生成Clight抽象语法，并基于CIL库；在下游，一个打印机为PPC抽象语法树提供具体的汇编语法，然后使用系统的汇编程序和链接器生成可执行二进制文件。21

编译器的前端通过C#minor和Cminor中间语言，分两次翻译掉C特定的特性。C#minor是Clight的一个简化的、无类型的变体，其中为整数、指针和浮点提供了不同的算术运算符，C循环被无限循环加块和封闭块的多级出口所取代。第一步相应地转换C循环并消除所有依赖类型的行为：解决了操作符重载问题；内存加载和存储以及地址计算都是显式的。下一种中间语言Cminor类似于C#minor，省略了（address of）操作符。Cminor函数局部变量不驻留在内存中，并且不能获取它们的地址。但是，Cminor支持在函数的激活记录中显式地分配数据堆栈。因此，从C#minor到Cminor的转换可以识别从未获取其地址的标量局部变量，将它们分配给Cminor局部变量，并使它们成为以后寄存器分配的候选变量；其他局部变量在激活记录中进行堆栈分配。**&**

  
编译器后端从指令选择过程开始，该过程识别使用组合算术指令（添加立即数、非和、旋转和掩码等）和目标处理器提供的寻址模式的机会。这个过程通过自下而上重写Cminor表达式来进行。目标语言是CminorSel，它是Cminor的一个依赖于处理器的变体，提供了额外的运算符、寻址模式和一类条件表达式（表达式仅针对其真值进行计算）。

下一步将CminorSel转换为RTL，RTL是一种经典的寄存器传输语言，其中控制表示为控制流图（CFG）。图的每个节点都携带一条机器级指令，该指令在tempo raries（伪寄存器）上运行。RTL是基于数据流分析进行优化的方便表示。目前实现了两种这样的优化：常量传播和公共子表达式消除，后者通过扩展基本块上的值编号来执行。第三个优化，惰性代码运动，是单独开发的，很快就会集成。与其他两种优化不同，惰性代码运动是按照验证程序方法实现的。24

在这些优化之后，通过干扰图的着色来执行寄存器分配。这个过程的输出是LTL，一种类似于RTL的语言，其中用硬件寄存器或抽象堆栈位置替换临时表。CFG然后被“线性化”，产生一个带有显式标签、条件和无条件分支的指令列表。接下来，在引用分配给堆栈位置的临时指令的指令周围插入溢出和重新加载，并在函数调用、序言和尾声周围插入移动以强制执行调用约定。最后，“stacking”过程列出函数的激活记录，将此记录中的偏移量分配给抽象堆栈位置和保存的被叫方存储寄存器，并通过相对于堆栈指针的显式内存加载和存储来替换对抽象堆栈位置的引用。6

这将我们带到Mach中间语言，它在语义上接近于PowerPC汇编语言。通过列表或跟踪调度的指令调度可以在此时执行，再次遵循验证程序方法。最后的编译过程将Mach指令扩展为PowerPC指令的罐装序列，处理特殊寄存器（如条件寄存器）和PowerPC指令集中的异常情况。目标语言PPC精确地模拟了PowerPC汇编语言的一大子集，省略了CompCert不生成的指令和特殊寄存器。23

从编译的角度来看，CompCert并不起眼：各种过程和中间表示是20世纪90年代早期的教科书式编译器技术。也许唯一令人惊讶的是中间语言的数量相对较多，但许多是彼此之间的小变体：为了验证的目的，将每个变体标识为一种不同的语言比将其标识为几个更通用的中间表示的不同子集更为方便。

#### 3.3. 证明编译器

CompCert的附加值不在于实现的编译技术，而在于源语言、中间语言和目标语言都已正式定义了语义，并且每个转换和优化过程都被证明在第2.4节的意义上保留了语义。

这些语义保存证明是使用Coq证明助手机械化的。Coq实现了归纳和共导结构的演算，这是一种强大的构造性高阶逻辑，同样支持三种常见的书写规范：通过函数和模式匹配，通过表示推理规则的归纳或共导谓词，以及通过一阶逻辑中的普通谓词。在CompCert开发中使用了这三种风格，从而产生了与编程语言研究论文中所能找到的非常接近的规范和定理语句。特别是，编译算法自然地以函数的形式出现，操作语义主要使用归纳谓词（推理规则）。Coq还具有更高级的逻辑特性，如高阶逻辑、依赖类型和ML样式的模块系统，这些我们在开发中偶尔会用到。例如，依赖类型允许我们将逻辑不变量附加到数据结构上，参数化模块允许我们重用通用的数据流方程求解器进行多个静态分析。

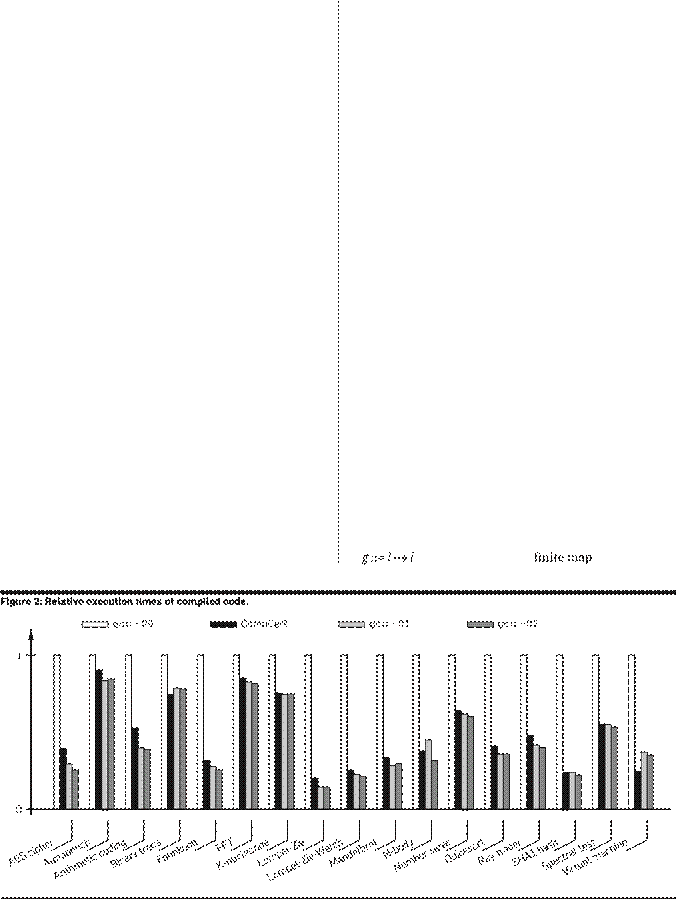
Coq中的定理证明是一个交互式的过程：例如，一些决策过程自动化了等式推理或Presburger算法，但大多数证明是由用户输入的“策略”（基本证明步骤）序列组成的，以指导Coq解决证明义务。在内部，Coq构建了证明项，这些证明项稍后将由一个小型内核验证器重新检查，从而对证明的有效性产生非常高的可信度。当以交互方式开发时，可以在批处理模式下对校对脚本进行事后复查。

整个Coq的形式化和证明代表了42000行Coq（不包括评论和空行）和大约三个人年的工作。在这42000行中，14%定义了CompCert中实现的编译算法，10%指定了所涉及语言的语义。剩下的76%对应于正确性证明本身。每个编译过程需要1500到3000行Coq作为其规范和正确性证明。同样地，每个中间语言用300到600行Coq指定，而源语言Clight需要1100行。额外的10000行对应于所有语言和过程之间共享的基础设施，例如机器整数算法和内存模型的形式化。

#### 3.4. 编程和运行编译器

我们使用Coq不仅作为验证程序进行语义保留证明，而且作为编程语言编写CompCert编译器的所有验证部分。Coq的规范语言包括一个小型的纯函数式语言，其特点是递归函数通过归纳类型（ML或Haskell风格的树型数据类型）上的模式匹配进行操作。有了一些独创性，这种语言就足以编写编译器了。编译器教科书中的高命令性算法需要用纯函数式重写。我们使用基于平衡树的持久数据结构，它支持高效的更新，而无需修改数据。同样，一元编程风格使我们能够以清晰、组合的方式对异常和状态进行编码。

与用常规命令式语言实现编译器相比，这种非常规方法的主要优点是，我们不需要程序逻辑（如Hoare逻辑）来连接编译器的代码和逻辑规范。实现编译器的Coq函数是Coq逻辑的一级公民，可以通过归纳、简化和等式推理直接进行推理。

为了获得一个可执行的编译器，我们依赖于Coq的提取工具，它可以从Coq函数规范自动生成Caml代码。将提取的代码与手工编写的编译器未经验证部分（如解析器）的Caml实现相结合，并通过Caml编译器运行所有这些，我们得到了一个具有标准样式命令行接口的编译器，该编译器在Caml支持的任何平台上运行，并生成在MacOS X下运行的PowerPC代码(其他目标平台正在开发中。）15**复写的副本**

#### 3.5. 性能

为了评估CompCert生成的代码的质量，我们在优化级别0、1和2将其与GCC4.0.1编译器进行了基准测试。由于标准基准测试套件使用了CompCert不支持的C特性，我们不得不推出自己的小套件，其中包含一些计算内核、加密原语、文本压缩器，一个虚拟机解释器和一个光线跟踪器。测试在2GHz PowerPC970“G5”处理器上运行。

|  |
| --- |
|  |
|  |  |

如图2中的计时所示，CompCert生成的代码的速度是未经优化的GCC生成的代码的两倍以上，并且在优化级别1和2上与GCC竞争。平均而言，CompCert代码只比GCC慢7%，比GCC慢12%。测试套件太小，无法得出明确的结论，但这些结果强烈表明，虽然CompCert不会在高性能计算中获奖，但其性能足以满足关键的嵌入式代码。**合同通用条款−01合同通用条款−02**

CompCert的编译时间在的2倍以内，这是合理的，并且表明为方便验证而引入的开销（许多小的过程，没有必要的数据结构等）是可以接受的。**合同通用条款−01**

### 4.寄存器分配

为了提供一个更详细的验证编译过程示例，我们现在给出CompCert的寄存器分配过程，并概述其正确性证明。

#### 4.1. RtL中间语言

寄存器分配是在RTL中间表示上执行的，它将函数表示为抽象指令的CFG，大致对应于机器指令，但在伪寄存器（也称为“临时寄存器”）上操作。每个函数都有无限量的伪寄存器，它们的值在函数调用中被保留。在下面的代码中，r范围在伪寄存器上，l范围在CFG节点的标签上。

说明：

*我*：：=（l）无运算（转到l）|（op，r，r，l）算术运算**没有**→

     |（，mode，r，r，l）内存加载**负载***k*→

     |（，mode，r，r，l）内存存储**商店***k*→

     |（sig，（r | id），r，r，l）函数调用**呼叫**→

     |（sig，（r | id），r）函数尾部调用**尾声**→

     |（cond，r，l，l）条件分支**条件**→*是的假*

     ||（r）函数返回**返回返回**

控制流图：

内部函数：F:：={=id；=；**名称** *信号*

**参数**= *r*→; 参数

**堆栈大小**= *n*; 堆栈数据块大小

**入口点**= *我*; 第一个指令的标签

**代码**= *克*}控制流程图

外部功能：

*铁*：={=id；=}**名称** *信号*

每条指令在伪寄存器r的列表中获取其参数，并将其结果（如果有的话）存储在伪寄存器r中。此外，它还带有可能的继承者的标签l。指令包括算术运算op（用一个重要的特例（，r，r′，l）表示寄存器到寄存器的拷贝）、内存加载和存储（数量）κ 通过将寻址模式应用于寄存器r）、条件分支（有两个后继者）以及函数调用、尾部调用和返回而获得的地址。→**opmove公司**→

RTL程序由一组内部或外部的命名函数组成。内部函数在RTL中由它们的CFG、CFG中的入口点和参数寄存器定义。外部函数不是定义的，只是声明的：它们为输入/输出操作和类似的系统调用建模。函数和调用指令带有sig签名，指定其参数和结果的数量和寄存器类（or）。**内部浮动**

RTL的动态语义以小步操作的方式表示，作为一个标记的转换系统。谓词gs→′ 表示从状态S到状态S′的一个执行步骤。全局环境G将函数指针和名称映射到函数定义。trace t记录这个执行步骤执行的输入-输出事件：除了对外部函数的调用之外，所有指令都为空（t=），在这种情况下，t记录函数名、参数和调用的结果。t型*Se*

执行状态S的形式为S(Σ, g、 l，R，M），其中g是当前正在执行的函数的CFG，l是该函数中的当前程序点，以及包含其激活记录的内存块。寄存器状态R将伪寄存器映射到它们的当前值（32位整数、64位浮点和指针的区分并集）。同样，内存状态M将（指针、内存量）成对映射到值，同时考虑多字节量之间的重叠。最后，Σ 为调用堆栈建模：它记录挂起的函数调用及其（g，l，R）组件。在对函数调用和返回进行建模时，会出现两种稍有不同的执行状态形式，即调用状态和返回状态，但这里将不进行描述。*ss*14*s*

为了展示RTL的语义，这里有两个定义一步转换关系的规则，分别用于算术运算和条件分支：

*克*（左）= **操作阀**（右上角）→，右，左′) （G），*s*，操作，R（R→)) = *u G S公司*(S，克，*s*，左，右，米）→e*S*(S，克，*s*，左′，R{R← u}，米）

*克*（l） =（条件，r，l，l）**条件**→*是的假*

*我*′*=* l if（条件，R（R→)) = if（cond，R））=（S，g，S，l，R，M）→e（S，g，S，l′，R，M）**evalcondtrue evalcondfalse***\_       我假是的  \_*(*r*→*G和SS*

#### 4.2. 寄存器分配算法

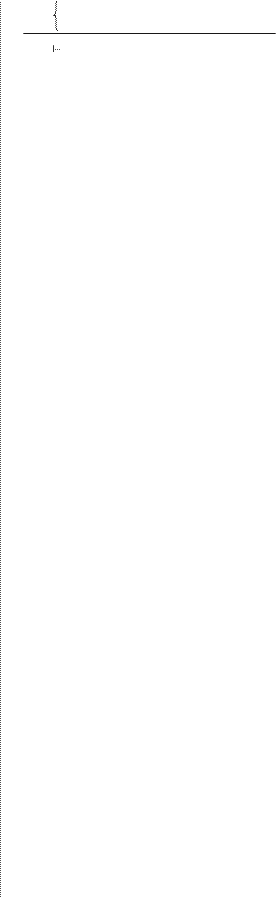
寄存器分配过程的目标是用位置l替换原始RTL代码中出现的无限数量的伪寄存器r，这些位置l是硬件寄存器（以小的、固定的数量提供）或激活记录中的抽象堆栈插槽（以无限数量提供）。由于访问硬件寄存器比访问堆栈插槽快得多，因此必须最大限度地使用硬件寄存器。寄存器分配的其他方面，如插入重载和溢出指令以访问堆栈插槽，则留给后续的过程处理。

寄存器分配从向后数据流分析执行的标准活动性分析开始。活跃度的数据流方程的形式如下

低压（l）=∪ {T（s，LV（s））| l}（8）的后继者

传输函数T（s，LV（s））计算程序点s“之前”活动的伪寄存器集，作为该点“之后”活动的伪寄存器LV（s）的函数。例如，如果s处的指令是（op，r，r，s′），则结果r将变为死，因为此时它被重新定义，但参数r变为活的。因为它们在这里使用：T（s，LV（s））=（LV（s）{r}）∪ r。但是，如果r在“之后”（r∉ L（s）），该指令是死代码，稍后将被消除，因此我们可以取T（s，LV（s））=LV（s）。**op公司**→→→

使用Kildall的worklist算法迭代求解数据流方程。CompCert提供了Kildall算法的通用实现及其正确性证明，也可用于其他优化过程。该算法的结果是从程序点到活动寄存器集的映射，证明了该映射满足正确性条件LV（l）⊇ T（s，LV（s））表示l的所有s继承者。我们只证明一个不等式而不是标准的数据流方程（8），因为我们只关心解的正确性，而不是它的最优性。

然后根据Chaitin规则构造一个以伪寄存器为节点的干涉图，并证明它包含所有必要的干涉边。通常，如果两个伪寄存器r和r′同时存在于一个程序点上，则该图必须在r和r′之间包含一条边。干扰的形式为“这两个伪寄存器干扰”或“这个伪寄存器和这个硬件寄存器干扰”，后者用于确保在函数调用中存在的伪寄存器不会分配给调用方保存寄存器。偏好边（“这两个伪寄存器最好分配到同一位置”或“这个伪寄存器最好分配到这个位置”）也被记录，尽管它们不影响寄存器分配的正确性，只影响其质量。6

寄存器分配的中心步骤是为干涉图着色，在由干涉边连接的两个节点被分配不同颜色的约束下，为每个节点r分配一个“颜色”（r），即硬件寄存器或堆栈槽。我们使用乔治和阿佩尔的着色启发法。由于这种启发式算法很难直接证明其正确性，我们将其实现为未经验证的Caml代码，然后使用一个简单的验证器对其结果进行后验验证。像许多NP难问题一样，图着色是算法的一个范例，它比直接证明正确更容易验证后验概率。着色结果的正确性条件为：*j*9*j*

1.   *j*（右）≠ *j*（右）′)如果r和r′干涉

2.   *j*（右）≠如果我和r干涉

3.   *j*（r） 和r有相同的寄存器类(**内部浮动**或者)

用Coq编写的布尔值函数对这些条件进行了检验，证明了它们是这三个条件的判定过程。如果检查失败，编译将中止，这表示外部图形着色例程中存在错误。

最后，重写原始RTL代码。对伪寄存器r的每个引用被对其位置（r）的引用替换。此外，执行合并和死代码消除。无副作用指令l:（op，r，r，l′）或*j***op公司**→

→ : (, 模式r，r，l′）被一个不可操作的l:（l′）替换，如果结果r在l（死代码消除）之后不是活的。同样，移动指令l:（，r，r，l′）替换为不可操作的l:（l′）if（r）=（r）（合并）。*我***加载nopopmovenop***ksdjdjs*

#### 4.3. 语义保持的证明

为了证明程序转换保留语义，CompCert项目中使用的一种标准技术是显示一个模拟图：原始程序中的每个转换必须对应于转换程序中具有相同可观察效果的一系列转换（相同的输入-输出操作痕迹，在我们的例子中）并保持一个给定的二元关系不变∼ 在原始程序和转换程序的执行状态之间。在寄存器分配的情况下，每个原始转换正好对应一个转换的转换，从而得到以下“锁步”模拟图：

*S*∼ tt公司*S***1**

*S***2                                   2**

（实线代表假设；虚线表示结论。）此外，如果∼ 关联初始状态和最终状态，这样的模拟图意味着原始程序的任何执行都对应于转换程序的执行，转换程序产生完全相同的可观察事件跟踪。因此，语义保留随之而来。

通过模拟证明的要点是∼ 关系。两个州的条件是什么(Σ, g、 l，R，M）和S(Σ′, g′，l′，R′，M′）有关系吗？直观地说，由于寄存器分配保留了程序结构和控制流，控制点l和l′必须是相同的，CFG g′必须是根据第4.2节所述的寄存器分配转换g的结果。同样，由于寄存器分配保留内存存储和分配，内存状态和堆栈指针必须相同：M′=M和′=。*ssjss*

不明显的关系是原始程序的寄存器状态R和转换程序的位置状态R′。假设每个伪寄存器r映射到位置（r），我们可以天真地要求r（r）=r′（（r））表示所有r。然而，这个要求太强了，因为它从本质上排除了两个活动范围不相交的伪寄存器之间的位置共享。为了获得正确的要求，我们需要考虑伪寄存器在程序点l处是活的还是死的语义含义。一个死伪寄存器r是这样的：它在点l处的值对程序的执行没有影响，因为要么r以后永远不会被读取，要么它总是在被读取之前被重新定义。因此，在设置寄存器和位置值之间的对应关系时，我们可以安全地忽略那些在当前点l处失效的寄存器。只需满足以下条件即可：*jj*

*R*（r） =r′（（r）），所有伪寄存器r在点l处活动。*j*

一旦建立了状态之间的关系，证明上面的模拟图就是对RTL语义的各种转换规则的例行案例检查。通过这样做，我们可以愉快地认识到，定义活跃度的数据流不等式，以及构造干扰图的Chaitin规则，是寄存器状态R，R′之间的不变量在所有情况下保持不变的最小充分条件。

**5.结论和观点**本文描述的CompCert实验仍在进行中，还有很多工作要做：处理C的一个较大的子集（如including）；部署并证明更多的优化是正确的；瞄准PowerPC以外的其他处理器；将语义保存证明扩展到共享内存并发等。然而，迄今为止获得的初步结果提供了强有力的证据，证明在当今证明助手的限制下，仅使用基本的语义和算法方法，就可以实现正式验证真实编译器的初始目标。我们使用的技术和工具还远远不够完善--更多的证明自动化、更高级的语义和更现代的中间表示都有可能显著减少证明工作--但足以实现目标。

回顾获得的结果，我们并没有完全排除编译器正确性的所有不确定性，而是将信任整个编译器的问题归结为信任以下部分：

1.   源语言（Clight）和目标语言（PPC）的形式语义。

2.   编译器中尚未验证的部分：

基于CIL的解析器、汇编程序和链接器。

3.   用于为编译器生成可执行文件的编译链：Coq的提取工具、Caml编译器和运行时系统(此编译链中的错误可能会使通过正确性证明获得的保证失效。）

4.   Coq证明助理本身(Coq实现中的错误或Coq逻辑中的不一致可能会伪造证据。）

问题（4）可能是最不值得关注的：正如黑尔斯所说，由证明助手机械地检查证明，生成证明项，比仔细地手工检查的数学证明更值得信赖。10

为了解决问题（3），CompCert项目中正在进行的工作研究了正式验证Coq的提取机制的可行性，以及从Mini-ML（该提取所针对的简单函数语言）到Cminor的编译器。由CompCert后端组成，这些工作最终会为用Coq编写和验证的程序生成一个可信的执行路径，就像CompCert本身一样，因此通过一种引导形式进一步增加了可信度。

问题（2）与未经验证的组件CompCert显然可以解决的重新实施和证明相应的通行证。语法分析器的语义保存很难定义，更不用说证明：如果不是语法分析产生的抽象语法树的语义，那么程序的具体语法的语义是什么？然而，CIL执行的一些解析后精化步骤可以通过正式的证明。同样地，证明汇编程序和链接器的正确性是可行的，如果没有必要的话。

也许最微妙的问题是（1）：我们如何确保形式语义符合语言标准和通用编程实践？由于所讨论的语义相对于整个编译器来说是很小的，所以专家的手动检查以及对语义的可执行形式进行的测试可以提供合理的（但不是形式上的）信心。另一种方法是证明与独立开发的替代形式语义的联系，例如为程序的演绎验证工具加下划线的公理语义（参见Appel和Blazy的示例）。此外，这种方法是朝着一个更雄心勃勃的长期目标迈出的第一步：使用正式方法认证参与关键软件开发、验证和执行的验证工具、代码生成器、编译器和运行时系统。2

# 致谢

作者感谢S。布拉齐，Z。达加耶，D。多利格斯，B。Gr公司é甲状腺肿，T。莫尼奥特，L。里多和B。感谢他们对CompCert开发的贡献。阿佩尔，Y。伯托，E。莱迪诺，P。Letouzey和G。感谢他们的建议、反馈和帮助。这项工作得到了国家研究机构（Agence Nationale de la Recherche）的支持，批准号为ANR-05-SSIA-0019。

## 工具书类

**1.Appel，A.W.基础校对代码。计算机科学逻辑2001（2001），IEEE，247–258。**

**2.阿佩尔，A.W.，布拉兹，S。小步进Cminor的分离逻辑。高阶逻辑中的定理证明，TPHOLs 2007，LNCS（2007）第4732卷，Springer，5–21。**

**3.Bertot，Y.，演员é兰，P。交互式定理证明与程序开发——Coq&apos;Art:归纳结构演算（2004），Springer。**

**4.Blazy，S.，Dargaye，z.，Leroy，X。C编译器前端的形式化验证。FM 2006：形式化方法国际研讨会，LNCS（2006）4085卷，斯普林格，460–475。**

**5.布拉齐，S.，勒罗伊，X。C语言Clight子集的机械化语义。J。自动。推理（2009）。接受出版，出现。**

**6.Chaitin，G.J.通过图着色进行寄存器分配和溢出。1982年SIGPLAN编译器构造研讨会（1982），ACM，98–105。**

**7.Coq开发团队。考证助理。网址：http://coq.inria.fr/，1989–2009。**

**8.Dave，M.A.编译器验证：参考书目。SIGSOFT软件。工程注释28，6（2003），2。**

**9.George，L.，Appel，A.W.迭代寄存器合并。ACM变速箱。掠夺。语言系统。18, 3 (1996), 300–324.**

**10.黑尔斯，T.C.正式证据。通知AMS 55，11（2008），1370–1380。**

**11.莱罗伊，X。编译器后端的正式认证，或：用证明助手编程编译器。在第33届编程语言原理研讨会（2006）上，ACM，42–54。**

**12.莱罗伊，X。CompCert验证了编译器、软件和注释证明。可在http://compcert. inria.fr/，2008年8月。**

**13.莱罗伊，X。正式验证的编译器后端。附件十四：0902.2137[cs]。提交日期：2008年7月。**

**14.Leroy，X.，Blazy，S。类C内存模型的形式化验证及其在程序转换验证中的应用。J。自动。推理41，1（2008），1–31。**

**泽维尔·勒罗伊（泽维尔。leroy@inria.fr)因里亚**

**法国，巴黎**

**© 2009 ACM 0001-0782/09/0700$10.00**

**15.Letouzey，第。Coq萃取综述。算法逻辑与理论，欧洲可计算性，CiE 2008，LNCS（2008）5028卷，斯普林格，359–369。**

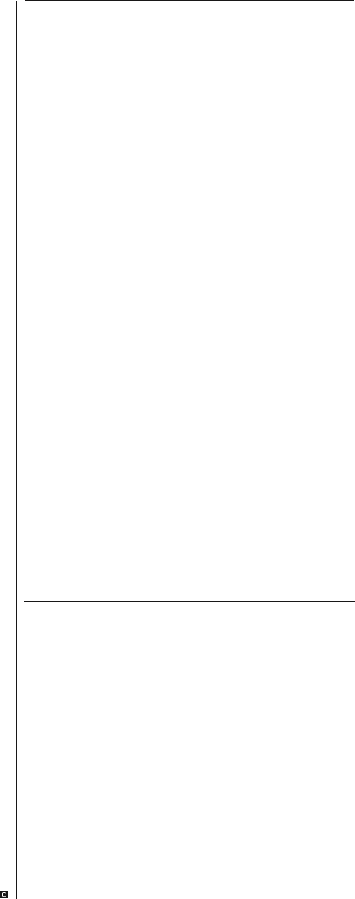
**16.麦卡锡，J.，画家，J。算术表达式编译器的正确性。在计算机科学的数学方面，《应用数学研讨会论文集》（1967）第19卷，AMS，33-41。**

**17.米尔纳，R.，韦劳赫，R。用机械化逻辑证明编译器的正确性。《第七届机器智能研讨会论文集》，机器智能第7卷（1972年），爱丁堡大学出版社，51–72。**

**18.莫里塞特，G.，沃克，D.，克雷，k.，格劳，N。从systemf到类型化汇编语言。ACM变速箱。掠夺。语言系统。21, 3 (1999), 528–569.**

**19.Necula，G.C.证明携带代码。在第24届编程语言原理研讨会（1997）上，ACM，106–119。**

**20.Necula，G.C.优化编译器的翻译验证。编程语言设计与实现2000（2000），ACM，83–95。**

**21.Necula，G.C.，McPeak，S.，Rahul，S.P.，Weimer，W。CIL：分析和转换C程序的中间语言和工具。《编译器构造》，LNCS（2002）第2304卷，Springer，213–228.22，Pnueli，A.，Siegel，M.，Singerman，E。翻译验证。系统构造和分析的工具和算法，塔卡斯&apos;98，LNCS（1998）第1384卷，斯普林格，151–166。**

**23.特里斯坦，J.-B.，勒罗伊，X。翻译验证器的形式化验证：指令调度优化案例研究。在第35届编程语言原理研讨会（2008）上，ACM，17–27。**

**24.特里斯坦，J.-B.，勒罗伊，X。已验证惰性代码运动的验证。**

**在程序设计语言中**

***设计与实施2009***