计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.191000173

在可信编译器设计中实践 CompCert 编译器的语法分析器形式化验证过程

李 凌 李璜华 王生原
清华大学计算机科学与技术系 北京 100084
(liling14@tsinghua.org.cn)

摘 要 Jourdan 等在其 2012 年发表的论文“Validating LR(1) Parsers”中提出了一种形式化验证语法分析器的方法,并将其成功地应用于 CompCert 编译器(2.3 以上版本)的语法分析器验证中。借助这种方法,文中完成了 L2C 项目中的 Lustre* 语言语法分析器的形式化验证,实现了开源 L2C 编译器前端语法分析器的两个选项之一。首先对这一语法分析器的实现进行了论述,其中包括有参考价值的技术细节;随后分析了该语法分析器的运行性能及正确性;最后对如何将这一方法推广至更一般的应用场景进行了总结。

关键词:语法分析;LR(1)分析器;形式化验证;Lustre * 语言;CompCert;Coq
中图法分类号 TP314

Experiment on Formal Verification Process of Parser of CompCert Compiler in Trusted Compiler Design

LI Ling,LI Huang-hua and WANG Sheng-yuan
Department of Computer Science and Technology,Tsinghua University,Beijing 100084,China

Abstract Jourdan and others presented a method to formally verify a parser in their paper Validating LR(1) Parsers published in 2012,and successfully applied it to the parser verification of CompCert compiler(version 2.3 and above). With this method,formal validation of the Lustre* parser is completed,which is a part of the Open L2C project,and one of the two options of the front-end parser of the Open L2C compiler is implemented. Firstly,this paper discussesthe implementation of the parser,including some valuable technical details. Thenit analyzes the running performance and correctness of the parser. And finally,how to apply this method tomore general parsersis summarized.

Keywords Syntax parsing,LR(1) parser,Formal verification,Lustre* language,CompCert,Coq

1 引言

“误编译”问题是编译器中常见的错误之一。对于许多领域来说,误编译一般不会引发本质的问题,因此其带来的影响往往被忽视。然而,对于安全攸关系统[1]而言,必须考虑编译器引入的错误,否则在源程序级进行的验证工作可能在目标程序级失效。实际上,如航空领域的 RTCA DO-178B/C 标准中,编译器属于需要鉴定的工具类软件,需要按照机载软件的要求进行对待。

为保证编译器的正确性,传统上一般采用大量的测试以及严格的软件过程管理,但这并不能杜绝“误编译”的发生。对编译器进行正确性验证,而不仅仅是测试,是解决问题的根本途径,而最严格的验证手段莫过于采用形式化方法。近年来,有关编译器形式化验证的研究工作取得了长足的进步,已达到了实用化水平,为未来制定新的工业标准奠定了坚实的基础。CompCert 编译器[2]是经过形式化验证的可信编译器的杰出代表。该编译器将 C 语言的一个子集 Clight 翻译为汇编代码(目前支持 PowerPC,IA32,ARM 以及 RISC-V 等主流体系结构,近期已扩展到支持 64 位结构),其编译过程中各个阶段的翻译正确性都借助交互式定理证明工具 Coq[3]进行了证明,且这些证明均可由独立的证明检查器进行检查。这是迄今最强的形式化验证手段,达到了人们期望的最高可信程度[4],已被应用于航空等安全攸关系统[1]。

语法分析的理论和技术已经相当成熟,并且已有多种语法分析器自动构造工具。然而,作为编译器的重要组成部分,语法分析器的形式化验证工作是获得编译器正确性证据链的重要一环。

提高语法分析器的可信度有多种途径,Jourdan 等[5]总结了 3 种常见的方法。第 1 种是对未经验证的语法分析器进行介入,使编译器在每一次运行时都对其生成的完整语法分析

到稿日期:2019-10-28 返修日期:2020-03-20 本文已加入开放科学计划(OSID),请扫描上方二维码获取补充信息。
基金项目:核高基重大专项(2017ZX01030-301-003)
This work was supported by the National Science and Technology Major Project (2017ZX01030-301-003).
通信作者:王生原(wwssyy@tsinghua.edu.cn)

树进行检查，判断它是否符合文法的定义。这种方法实现简单，但并不能证明语法分析器的完备性（所有合法的输入都应当被接受）和无二义性（对于任一输入，最多生成一棵语法分析树）。第 2 种是对手工编写或程序生成的语法分析器直接进行程序证明。这样的证明不仅复杂冗长（特别是针对自动生成的语法分析器），而且语法分析器在每次被修改后都要重新进行一次证明。第 3 种是 Barthwal 等提出的方法[6]，即仅针对语法分析器的生成器（Parser Generator）进行一次形式化验证，以确保由它生成的全部语法分析器相对于输入文法而言是正确的。Mcpeak 等[7]也曾提出过类似的方法。然而，该方法目前仅对 SLR 文法有效，若要使其适用于应用更加广泛的文法类（如 LALR 文法），则必须对其进行实质性的改进。

Jourdan 等[5]还提出了第 4 种方法，即针对未经验证的语法分析器的生成器所生成的 LR(1) 自动机进行确认，但确认程序（validator）是经过形式化验证的。这种方法比直接验证语法分析器的生成器更加简单，而且适用于各类 LR(1) 自动机，包括 LR(0)，SLR 以及 LALR 等[5]。Jourdan 等将该方法成功地应用于构造 CompCert 编译器[2]的语法分析器，并将其集成到 2014 年 5 月发布的 CompCert 2.3 版本中[8]。之后，Jourdan 等还对满足 C11 标准的语法分析器的正确性开展了深入的研究[9]，但目前尚未完成严格的形式化验证。

笔者所在课题组的 L2C 项目源于国内某安全攸关领域的实际需求[10]，源语言为一个单时钟的核心 Lustre 子集。L2C 可信编译器的一个完整企业版[11]的源语言为一个面向领域的类 Lustre 同步数据流语言。L2C 项目开展之后，合作企业的建模与代码生成工具从原 Scade[12]的用户走向自主研发之路，目前已投入到实际应用中。近年来，L2C 项目组主要致力于开源 L2C 可信编译器的研发工作[13-15]。源语言 Lustre* 的语法定义参见文献[16]。

本文主要是从技术环节上重现 CompCert 编译器中语法分析器的形式化验证过程，即将 Jourdan 等提出的方法应用于 L2C 项目中 Lustre* 语言语法分析器的形式化验证中，实现了开源版 L2C 编译器[14]前端语法分析器的一个选项，并将其集成到现有版本中。文中对这一语法分析器的实现全过程进行了比较详细的论述，包括一些有参考价值的技术细节。

本文第 2 节介绍了 Jourdan 等的语法分析器形式化验证的思路；第 3 节论述了 Lustre* 语言语法分析器形式化验证的实现过程；第 4 节简述了具体的实现情况；最后总结全文并进行展望。

2　Jourdan 等的语法分析器形式化验证思路

图 1[5]展示了生成语法分析器并对其进行形式化验证的全过程。在“编译－编译期”，即语法分析器的生成器（图中的 Instrumented parser generator）根据给出的语法分析规范生成语法分析器时，原文法 G（图中的 Grammar）、所生成的自动机 A（图中 LR(1) Automaton）连同将被用作证书的辅助信息（图中 Certificate）作为确认程序（图中 Validator）的输入，确认程序检查 A 所识别的语言和 G 的语言的等价性。

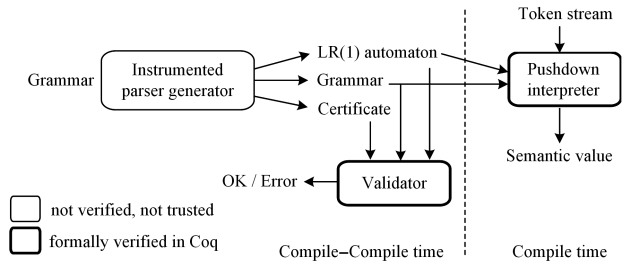

图 1　对语法分析器进行形式化验证的总体流程

Fig. 1　Overall flow of formal verification of parser

如果生成的自动机 A 被确认程序验证为正确无误的，就得到了经过形式化验证的语法分析器。它能够对词法分析器输出的单词序列进行解析，输出经过结构化组织的语义值，即“编译期”所发生的事件。综上所述，使用一个未经形式化验证的语法分析器的生成器，根据给定的文法可以生成一个经过验证的语法分析器，而所生成的语法分析器的正确性由以下确认程序的形式化验证提供保证。

Jourdan 等对语法分析器进行形式化验证的主要目的是证明语法分析器拥有某些特定的性质，如可靠性（soundness）、安全性（safety）、完备性（completeness）和无二义性（unambiguity）。这 4 种性质保证了语法分析器的正确性。其中，在不考虑发生内部错误的情况下，可靠性对文献[5]中所定义的自动机都成立；无二义性在安全性与完备性成立的条件下也是自动成立的。而安全性和完备性仅对某些自动机成立，因此对所构造的自动机证明其安全性与完备性是 Jourdan 方法最核心的部分。Jourdan 等的证明思路是：将安全性和完备性转换为能够确保这两个性质的若干充分条件，然后使用相应的确认程序进行自动检查，以确保这些条件成立。所有相关的定义以及这些确认程序的正确性证明均在 Coq 中实现。

可靠性是指语法分析器只接受正确的输入。根据文献[5]中描述的可靠性定理，如果语法分析器接受了某个给定的输入，并输出语义值 v，则一定能够根据文法从起始符号 S 推导出 ω，同时也会产生语义值 v，记作 $S \xrightarrow{v} w$。

Jourdan 等证明，如果不出现内部错误，可靠性对文献[5]中定义的所有 LR(1) 自动机都能够成立。对于发生内部错误的情形，可靠性定理未考虑，而是将这一负担转嫁到安全性定理中。

安全性是指语法分析器在运行过程中不会产生内部错误。根据文献[5]中描述的安全性定理，如果能够通过负责安全性的确认程序检查，那么，对于任何给定的输入，语法分析器一定能够返回运行结果（得到语义值或解析失败、超时等），即不会发生内部错误。

Jourdan 等将内部错误总结为以下 4 类。

(1)对于规约动作 $A \to X_1 \cdots X_n \{f\}$，下推栈中将弹出 n 个元素 $(\sigma_1, v_1) \cdots (\sigma_n, v_n)$，此时若不满足对任意的 $i \in \{1 \cdots n\}$，$incoming(\sigma_i) = X_i$，则语法分析器将产生错误。这里，σ_i 对应状态栈，v_i 对应语义栈，$incoming(\sigma_i)$ 代表自动机中进入状态 σ_i 的所有转移边所标记的全部终结符或非终结符。

(2)试图从空的下推栈中弹出元素。

(3)试图执行 goto 表中不存在的表项。

(4)试图在下推栈非空时执行接受动作。

以上的错误可全部归结为与下推栈相关的错误。它们或者是由于下推栈中的元素数量不匹配(过多或过少),或者是由于下推栈中的元素内容不匹配(类型或状态不正确)而造成的。因此,要防止内部错误,只需保证下推栈在语法分析的任意时间点正确即可。为此,Jourdan 等提出需要安全性确认程序检查的一组性质,可作为足以保证上述安全性的一种不变量条件。

Jourdan 等对语法分析器中的每个状态都构建了两个属性:pastSymbols 和 pastStates[5]。用 pastSymbols(σ)和 pastStates(σ)分别表示与状态 σ 的入栈相关联的靠近栈顶的符号集合序列和状态集合序列。安全性确认程序检查的一组性质可描述如下。

(1)对任意从状态 σ 到状态 σ′的转移,满足:

1) $pastSymbols(\sigma')$ 是 $pastSymbols(\sigma)incoming(\sigma)$ 的后缀(*incoming* 的含义如前);

2) $pastStates(\sigma')$ 是 $pastStates(\sigma)\{\sigma\}$ 的后缀。

(2)对任意具有归约 $A\rightarrow\alpha\{f\}$ 动作的状态 σ,满足:

1) α 是 $pastSymbols(\sigma)incoming(\sigma)$ 的后缀;

2)设 $pastStates(\sigma)\{\sigma\}$ 为 $\Sigma_n\cdots\Sigma_0$,且 α 的长度为 k,那么对任意状态 $\sigma'\in\Sigma_k$,goto 表在 (σ',A) 上有定义。

(3)对任意具有接受动作的状态 σ,满足:

1) $\sigma\neq init$(初态);

2) $incoming(\sigma)=S$(文法开始符号);

3) $pastStates(\sigma)=\{init\}$。

文法符号集合是有限的,故这 3 条性质显然可判定。

完备性是指语法分析器能够接受所有正确的输入,同时,对于所有正确的输入,语法分析器或者由于内部错误而终止,或者输出相应的解析结果并得到相应的语义值。由于在证明安全性的过程中已经证明了语法分析器不会产生内部错误,因此能够通过完备性定理保证语法分析器对于正确的输入一定能够输出对应的语义值。

类似地,根据文献[5]中描述的完备性定理,如果能够通过负责完备性的确认程序检查,就能够确保语法分析器满足上述完备性。同样地,Jourdan 等提出了一组足以确保完备性的可判定条件。

(1)为计算 *first* 集和 *nullable* 集(可致空集)所用到的标准定义方程组可完成不动点求解。

(2)对于任何状态 σ,其项目集 $items(\sigma)$ 一定是闭包的。

(3)对于任何状态 σ,若项目 $A\rightarrow\alpha\cdot[a]\in items(\sigma)$,那么动作表中 (σ,a) 所对应的表项一定为归约动作 $reduceA\rightarrow\alpha\{f\}$。

(4)对于任何状态 σ,若项目 $A\rightarrow\alpha_1\cdot a\alpha_2[a']\in items(\sigma)$,那么动作表中 (σ,a) 所对应的表项一定为 $shift\sigma'$。其中 σ′是满足 $A\rightarrow\alpha_1 a\cdot\alpha_2[a']\in items(\sigma')$ 的某个状态。

(5)对于任何状态 σ,若项目 $A\rightarrow\alpha_1\cdot A'\alpha_2[a']\in items(\sigma)$,那么 *goto* 表中 (σ,A') 所对应的表项或者没有定义,或者一定为 σ′。其中 σ′是满足 $A\rightarrow\alpha_1 A'\cdot\alpha_2[a']\in items(\sigma')$ 的某个状态。

(6)对于每个终结符 a,有 $S'\rightarrow\cdot S[a']\in items(init)$。注意:$S$ 为文法开始符号,S' 为增广文法开始符号,文献[5]中所有讨论均以假定输入为无穷序列为前提。

(7)对于任何状态 σ,如果有 $S'\rightarrow S\cdot[a']\in items(\sigma)$,那么其中一定包含有 *accept* 动作。

无二义性是指语法分析器对同一个输入只输出一种结果。Jourdan 等证明了具有安全性和完备性的语法分析器同时具有无二义性。假设对于某个 w,同时有 $S\xrightarrow{v_1}w$,$S\xrightarrow{v_2}w$,根据安全性假设,语法分析器不会遇到内部错误,因此根据完备性定理,给定足够的运行步数,语法分析器对于输入 $w\omega$(其中 ω 是任意的)一定能够输出 v_1,类似地也能够输出 v_2。然而,根据定义,自动机是一个函数,它应当有着确定的执行结果。因此,必然有 $v_1=v_2$。

以上 4 个性质的证明,相当于完成了对语法分析器的形式化验证,更具体的验证原理可参见文献[5],以及相关确认程序及其验证的 Coq 代码(已集成到 CompCert[8])。

3 形式化验证语法分析器的具体实现

3.1 语法分析器的总体结构

图 2 展示了构建一个语法分析器所需的全部步骤和总体流程。

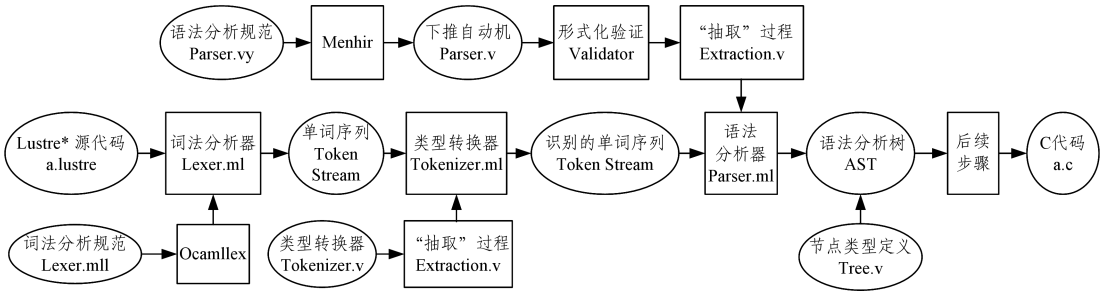

图 2 构建经过形式化验证的语法分析器的具体步骤

Fig. 2 Steps of constructing a formally verified parser

源代码被语法分析器接收后,首先经过词法分析器的处理,转换为单词(Token)序列。这里的词法分析器是由 Ocamllex 根据给定的词法分析器规范自动生成的(见 3.2.2 节)。在得到词法分析的结果后,由于词法分析所生成的单词序列的类型不能被语法分析器正确识别,因此需要编写一个类型转换器进行类型转换(详细的叙述见 3.2.2 节 3.2.3 节)。转换后的单词序列经过语法分析器的后,形成一棵语法分析树。这里的语法分析器是由 Menhir[17] 生成器依照给定的语法分析规范生成的(见 3.3 节),并且经过了 Jourdan 等编写的确认程序的验证,完成了形式化验证的过程(见 3.4.1

节）。同时，语法分析树的节点类型是预先定义的（见 3.2.1 节）。类型转换器、语法分析器等由 Coq 代码编写的组件还需经过“抽取”生成 Ocaml 代码才能被集成至已有的编译器中（见 3.4.2 节）。在得到语法分析树后，经过编译器的后续步骤处理即可得到目标语言的输出。

3.2　实现语法分析器前的准备工作

在着手实现语法分析器之前，我们要先完成一些准备工作：(1)定义抽象语法树的节点类型；(2)实现词法分析器。后者为语法分析提供输入，而前者定义了语法分析器的输出结构。

3.2.1　抽象语法树的节点定义

抽象语法树（Abstract Syntax Tree，AST）是沟通语法分析和后续步骤之间的桥梁。它既是语法分析的结果，又是后续步骤的输入和处理对象。由于本课题是开源 L2C 项目的一部分，为了在不更改其他阶段代码的基础上保证后续步骤的顺利进行，对抽象语法树的定义必须与开源 L2C 项目中原有的定义[18]一致，以确保新的语法分析器的输出在类型上与原有的输出一致，从而使其结果可作为后续步骤的正确输入。我们将抽象语法树的 Coq 文件记作 Tree.v。Tree.v 中对标识符的表示是以 Record 类型实现的：Record ident：＝{name：str；key：BinNums.positive}。Coq 中的 Record 类型类似于 C 语言中的结构体，这段代码定义了一个名为 ident 的类型，用以表示解析过程中出现的标识符。其中，name 成员是一个字符串，对于用户自定义的变量名、节点名和函数名等名称，name 即为该名称本身；对于整形和浮点数等常量，name 是该常量的字符串表示。key 成员是每个标识符在编译器中的整数表示，它以 Coq 中二进制的方式表示。

Tree.v 中大量使用了枚举类型和复合类型来表示抽象语法树的各类节点，其中每一个类型都与语法树的一类节点具有一一对应的关系。下面以 Lustre* 中的类型声明部分为例进行说明。设类型声明语句的抽象语法规则（使用扩展巴科斯范式）为：

〈type_decl〉::＝(IDENT，〈kind〉)

其中，〈type_decl〉表示一个类型声明节点，它由两部分构成：一部分是标识符（IDENT）；另一部分是一个〈kind〉节点，表示这个标识符所代表的类型。在 Tree.v 中，类型声明节点的定义如下：

Inductive typeStmt：＝｜TypeStmt：ident－＞kind－＞typeStmt

类型 typeStmt 的构造子 TypeStmt 接受两个参数，其类型分别为 ident 与 kind。ident 即上文提到的用于表示标识符的 Record 类型，kind 是一个用来表示语法树中〈kind〉节点的复合类型。利用 TypeStmt 构造子就能将一个具有 ident 类型的表达式 p 和具有 kind 类型的表达式 q 通过 TypeStmt p q 得到一个具有 typeStmt 类型的表达式。

3.2.2　词法分析

词法分析是整个语法分析乃至整个编译过程的第一个步骤。L2C 编译器的实现语言是 Ocaml。与 Jourdan 等在 CompCert 编译器的做法一样，本文使用词法分析器构造工具 Ocamllex 来实现所需要的词法分析。Ocamllex 的使用方式与其他 lex 实现类似，即先写出词法分析器的规范文件，然后用生成器自动生成相应的词法分析器代码。对于 Ocamllex，规范文件的后缀名一般为.mll，Ocamllex 会将其转换为后缀名为.ml 的 Ocaml 代码，词法分析过程作为其中的一个函数，供上层 Ocaml 代码调用。在本文工作之前，Open L2C 已有一个待测试的前端，并且也使用了 Ocamllex 来自动生成词法分析器代码。然而实践中发现，因本文使用了语法分析器自动生成工具 Menhir[17] 的 Coq 后端，已有的这个词法分析器代码不能直接使用（参见下文的解释）。因此，本文开发了新的词法分析模块。

本文编写了一个名为 Lexer.mll 的词法分析器规范文件，通过 Ocamllex 生成了名为 Lexer.ml 的文件，词法分析器即为 Lexer.ml 中的一个函数。这个规范文件包括两个主要的规则：第一个规则 token 负责解析各种终结符并执行对应的语义动作；第二个规则 comment 负责解析代码中的注释并跳过它们。规范文件的主体结构如下：

```
rule token＝parse
  ｜ regexp
     { action }
  …
and comment＝parse
  ｜ regexp
     { action }
  …
```

本节侧重于对终结符进行词法分析。对不含任何语义值的终结符进行词法分析最为简单，它们有着相同的结构。例如，对于“function”关键字，有如下的匹配规则：

```
｜“function”
     { get_token FUNCTION′token }
```

第一行的正则表达式表示完全匹配“function”关键字，第二行的语义动作表示在解析至“function”关键字时返回 get_token FUNCTION′token 函数执行的结果（3.2.3 节将详细叙述该函数）。

通常情况下，如果使用的是 Ocamlyacc 或 Menhir 的 Ocaml 端来生成 Ocaml 版本的语法分析器，那么词法分析器中的语义动作应当是该终结符在上文提到的 terminal 类型中对应的构造子。举例来说，在原本的开源 L2C 项目中，编译器使用的是由 Ocamlyacc 生成的语法分析器。同样，对于“function”关键字，词法分析规范中的匹配规则如下：

```
｜“function”
     { FUNCTION}
```

其直接返回了 FUNCTION 构造子。注意“′t”后缀只有在使用 Coq 后端时才会被自动添加，因此这里的构造子没有“′t”后缀。

在使用 Menhir 的 Coq 后端时，语义动作的写法变得繁琐，其原因有两个。首先，Menhir 的 Coq 后端在将终结符转换为 terminal′的构造子时，自动地为每一个构造子添加了“′t”后缀。因此，构造子 FUNCTION 在语法分析器 Parser.v 中会变为 FUNCTION′t。如果在词法分析的语义动作中写作 FUNCTION，则编译时会因找不到对应的构造子而提示错误。其次，Parser.v 中将终结符和非终结符的定义置于模块 Gram 下。由于在词法分析器中没有 Gram 模块的作用域，因

此访问不到 Gram 模块中的终结符定义，也就不能简单地将语义动作写作构造子 FUNCTION't。

这里，我们定义了一个 Coq 类型 token_id。对于每一个终结符（如“function”）及其构造子（如“FUNCTION”），在 token_id 中均有一个对应的以“'token”为后缀的构造子（如“FUNCTION'token”）。token_id 类型的形式如下：

```
Inductive token_id:=
  | COMMA'token
  | COLON'token
  | SEMICOLON'token
  ...
```

它们分别对应终结符“,”“:”“;”和构造子 COMMA't，COLON't，SEMICOLON't。此外，还需要一个函数 get_token，其作用是根据 token_id 的不同构造子，在 Gram 作用域中寻找对应的终结符。例如，对于 COMMA'token，get_token COMMA'token 返回 Gram. COMMA't 对应的语义值。

回到词法分析中的例子，当匹配到“function”关键字时，语义动作 get_token FUNCTION'token 将返回 Gram 中对应的终结符 FUNCTION't。经过这样的转换后，才能得到正确的构造子。

对于不含语义值的终结符，其词法分析规则与上述例子类似。

接下来，需要对含有语义值的终结符进行词法分析。在本课题中，由于含有语义值的终结符的语义值类型均为 string，因此对它们解析规则具有相同的结构。以整型常量为例，其词法解析规则如下：

```
| ['0'-'9']+ as lxm
    {get_token(CONST_INT'token lxm)}
```

在正规表达式之后出现了“as lxm”运算，表示将正规表达式匹配得到的结果字符串作为变量 lxm 在语义动作中使用。对于带有语义值的终结符，其解析规则与上述例子类似。这里的语义动作同样是使用 get_token 函数来返回对应的构造子，只是形式上略有不同。

3.2.3 类型转换

如 3.2.2 节所述，为了让词法分析器执行的语义动作能够返回正确的终结符和语义值，需要编写一个类型转换函数来完成这一任务，这一类型转换函数位于源码的 Tokenizer. v 文件中。

Tokenizer. v 文件中的内容分为两个部分。第一部分是上文提到的 token_id 类型的定义，每一个终结符在 token_id 中都有一个对应的构造子。不含语义值的终结符在 token_id 中对应的构造子是裸构造子，如 FUNCTION'token；反之，在 token_id 中有一个接受该语义值类型（在本课题中仅有一种类型，即 string）作为参数的构造子，如 CONST_INT'token，它接受一个类型为 string 的参数。

Tokenizer. v 文件内容的另一部分是转换函数 get_token，它接受一个 token_id 类型的参数，返回 Gram 中定义的终结符。它有一个辅助函数 get_type，get_type 接受一个 Gram. terminal 类型的参数和一个 Gram. symbol_semantic_type (Gram. T t)类型的参数。get_token 在调用 get_type 时传入的第一个参数是 Gram. terminal 中对应的构造子；如果要转换的终结符不含语义值，则第二个参数为空，否则该参数为终结符所携带的语义值。

3.3 使用 Jourdan 提出的方法构建语法分析器

Ocamlyacc 和 Menhir 是 Ocaml 中最常用的两种语法分析器的生成器。其中，Menhir 由于能生成可读性更好的错误提示，同时生成的语法分析器的性能和易用性更优秀，因此被更多的编译器开发者所使用和推荐。Menhir 生成的语法分析器原本是由 Ocaml 语言编写的，Jourdan 等在实现 CompCert 的语法分析器时对 Menhir 进行了修改，使其能够生成由 Coq 代码实现的语法分析器，以供确认程序进行形式化验证。本文使用的生成器便是 Jourdan 等改进的 Menhir[19]。

我们从语法分析的起始符号（或者抽象语法树的根节点）开始进行说明。在 Lustre* 的语法规范文档中，每一个输入的 Lustre* 文件都是一段程序（Program），语法分析的起始符号是〈program〉。其语法规范的扩展巴科斯范式如下：

```
〈program〉::={〈decls〉}
〈decls〉::=〈type_decl〉
|〈const_decl〉
|〈node_decl〉
|SEMICOLON'token
```

由上面的扩展巴科斯范式可知，一个程序由零到多个声明组成，每个声明可以是类型声明，也可是常量声明或节点声明。上述语法在语法分析规范中的写法如下：

```
p_Program:
  |v1=p_Block_List;EOF
    {Programv1}
p_Block_List:
  |v1=p_Block;v2=p_Block_List
    {v1::v2}
  |(*empty*)
    {[]}
p_Block:
  |v1=p_Type_Block
    {v1}
  |v1=p_Const_Block
    {v1}
  |v1=p_Function_Block
    {v1}
```

每一个语法分析规则都以一个非终结符加一个冒号开始，用以说明如何对这一非终结符进行解析。每个以“|”起始的语句都代表了对该非终结符的一种解析规则。大括号内的 Coq 代码则是当解析到该规则时所要执行的语义动作。

语法分析从非终结符 p_Program 开始。紧接的规则 v1=p_Block_List 表示在对应的语义动作中使用 v1 变量代表被解析的非终结符 p_Block_List，直至遇到终止符号 EOF。由于 p_Block_List 具有类型为 list nodeBlk 的语义值，因此在语义动作中调用构造子 Program v1 就能得到一个类型为 program 的语义值。

由上面的扩展巴科斯范式可知，〈program〉非终结符可以被解析为零到多个声明，即一个含有零到多个声明的列表。在语法分析规范中，非终结符 p_Block_List 对应语法规范中

的{〈decls〉}，它具有类型为 list nodeBlk 的语义值，其中 nodeBlk 为〈decls〉非终结符所携带的语义值的类型。对 p_Block_List 的解析，实际上是通过递归方式实现的。对于第一条规则，它将 p_Block_List 所对应的单词序列的头部归约成非终结符 p_Block，而将剩余部分归约成一个规模较小的 p_Block_List，这个 p_Block_List 也将按照同样的方式推导出一个 p_Block 和一个规模更小的 p_Block_List。如此递归执行，直至所有的单词序列都被归约为 p_Block，不再有多余的单词。这时执行第二条规则的语义动作，返回一个元素类型为 nodeBlk 的空列表“[]”。递归过程开始回溯，每次回溯都执行第一条规则的语义动作，即将每个 p_Block 携带的类型为 nodeBlk 的语义值加入对应的 p_Block_List 所携带的类型为 list nodeBlk 的列表中，直到回溯过程结束。在语法分析规范中，含有列表类型语义值的非终结符基本都是按照这种方式来解析的。

最后，p_Block 对应语法规范中的〈decls〉非终结符，它可以被推导为类型声明 p_Type_Block、常量声明 p_Const_Block 或节点声明 p_Function_Block，这 3 个非终结符携带的语义值都具有 nodeBlk 类型。

在定义语法规则时，需要解决的一个主要问题是如何确定表达式中运算符的优先级。如果使用 Ocamlyacc 或 Menhir 的 Ocaml 后端，可以通过%left，%right，%noassoc 等标志及声明的先后顺序来规定运算符的优先级和结合性，生成器则会利用这些声明尽可能地解决文法冲突。但是，Menhir 的官方文档中明确说明，在使用 Coq 后端时不能使用上述方法来解决冲突，因为经过形式化验证的使用运算符结合性消除冲突的方法是不存在的[17]。因此，我们必须对常量表达式的解析规则进行详细的编写，以消去文法中的冲突。

参考 ISO C99 文法规范中对表达式的文法定义和 CompCert 中对表达式及常量表达式的解析方法，本文以下列二元常量表达式为例说明消去冲突的方法：

```
p_Const_Binary_Expression:
  |v1=p_Const_Or_Expression
    {v1}
p_Const_Or_Expression:
  |v1=p_Const_And_Expression
    {v1}
  |v1=p_Const_Or_Expression;OR;v2=p_Const_And
  _Expression
    {CEBinOpExpr OR v1 v2}
  |v1=p_Const_Or_Expression;XOR;v2=p_Const_
  And_Expression
    {CEBinOpExpr XOR v1 v2}
```

运算符“or”和“xor”是所有一元和二元运算符中优先级最低的。运算符的优先级越低，它所对应的运算被归约的顺序就越靠后，这一点从直观上很容易理解（如取反运算“－10”就先于乘法运算“10＊20”被规约）。因此，“or”和“xor”运算应当是最后被归约的非终结符。当它们被归约时，唯一可能尚未被归约的非终结符只能是〈p_Const_Binary_Expression〉（囊括了所有运算的非终结符），因为此时不可能再出现其他的运算。因此，“or”和“xor”必然是最先被推导的运算。

在推导完全部的“or”“xor”运算后，优先级更高一级的运算是与运算“and”。在该非终结符里，自动机将推导出所有“and”运算，并将优先级更高的运算符留待其他非终结符进行推导。

```
p_Const_And_Expression:
  |v1=p_Const_Compare_Expression
    {v1}
  |v1=p_Const_And_Expression;AND;v2=p_Const_
   Compare_Expression
    { CEBinOpExpr AND v1 v2}
```

综上所述，消除文法冲突的思路是用不同的非终结符表示具有不同优先级的运算表达式，优先级越低的运算表达式最先被推导，最后被归约；反之最后被推导，最先被归约。对于每一个对应了运算表达式的非终结符，其第一个匹配规则用于推导优先级高一级的运算表达式所对应的非终结符，第二个匹配规则用于推导运算表达式。对于左结合的运算符，运算符左边的非终结符是自身，右边的非终结符是优先级高一级的运算表达式所对应的非终结符，反之亦然。这样，在解析的过程中，某一个非终结符或者遇到它所对应的运算从而根据第二条规则进行推导，或者根据第一条规则推导至优先级更高的运算所对应的非终结符。这样，就能够通过文法自身来表示不同运算符之间的优先级和运算符本身的结合性。

在节点声明中，对表达式〈expr〉进行冲突消除也采用了同样的方法，区别仅在于表达式支持的运算符比常量表达式更多，且还引入了三目运算等特殊的运算符。其对表达式进行冲突消除的复杂程度比常量表达式更高，难度更大，且需要一些技巧，但其基本思路和原理与上述例子是共通的。

3.4　完成语法分析器后的剩余工作

完成语法分析器之后，还有一些后续任务。首先，需要使用确认程序对生成的语法分析器进行形式化验证。在形式化验证通过之后，需要利用 Coq 的“抽取”机制将语法分析器转变为可运行的 Ocaml 函数。最后需要改写 L2C 项目中原有的顶层驱动，将本文所实现的词法分析器和语法分析器等组件集成至 L2C 中。

3.4.1　对语法分析器进行形式化验证

打开生成的语法分析器的 Coq 代码进行查看，会发现在语法分析器的末尾有两个由 Menhir 自动添加的定理，分别是 p_Program_correct 和 p_Program_complete。它们是用于证明语法分析器正确性的两条性质，即安全性和完备性，也就是对语法分析器进行形式化验证所需的证明入口。具体来说，首先要使用 coqc 编译器对 CompCert 中提供的确认程序进行编译，得到后缀名为“.vo”的二进制版本；然后调用 coqc 对所生成的语法分析器（在本课题中是 parser/Parser.v）进行编译运行。如果没有提示错误，则表示证明完成。

3.4.2　抽取 Ocaml 代码

在 L2C 项目中，顶层驱动和文件读写等部分都是使用 Ocaml 代码进行编写的，而本文生成的语法分析器以及抽象语法树的定义文件、词法分析部分的类型转换器等都是在 Coq 环境下编写、生成的。因此，我们要利用抽取机制将它们转换为 Ocaml 文件和函数。

首先，对抽象语法树的类型定义进行抽取工作。由于在

Tree.v 中只有类型定义而没有需要运行的函数，因此我们选择使用“Library”的策略。

Extract Constant Tree.str=>“string”

Extraction Library Tree

然后，抽取语法分析器和词法分析所需的类型转换器，并将它们抽取到同一个文件中：

Extract Constant Parser.intern_string=>“Obj.magic (fun s-> Camlcoq.intern_string s)”

Extract Constant Parser.ocaml_string =>“(fun s -> Camlcoq.camlstring_of_coqstring s)” Extraction “Parser.ml” Parser.p_Program Tokenizer.get_token

以上两个“Constant”策略用于在抽取后的 Ocaml 代码中调用已有的 Ocaml 函数，我们使用匿名函数的方式来实现。其中的 intern_string 函数由 Ocaml 编写，它在 Coq 代码中以 Parameter 的形式出现，在抽取时以匿名函数的形式完成对其的调用。

执行上述抽取代码就能得到两个文件 Tree.ml 和 Parser.ml。其中，Tree.ml 是对抽象语法树节点类型的定义，Parser.ml 中则包括语法分析器（函数名 p_Program，即编写语法分析规范时使用的起始符号）和类型转换器（函数 get_token）。这样就从 Coq 代码中抽取出了 Ocaml 函数和文件。

3.4.3 修改顶层驱动

至此，已经得到了所需的所有部件，即词法分析、语法分析和抽象语法树定义，并且通过 Coq 的抽取机制得到了这些部件的 Ocaml 代码。最后的工作就是修改顶层驱动，使之调用我们所编写的语法分析器进行编译。相关内容并非本文的主题，这里不做进一步阐述。

4 实现

本文工作主要包括以下几个部分：用于定义语法分析树的库文件 Tree.v、用于进行词法分析的 Lexer.mll、用于对单词序列进行类型转换的 Tokenizer.v，以及用于进行语法分析的 Parser.vy。主要使用 Coq 和 Ocaml 两种语言进行开发。确认程序提取自 CompCert 项目。此外，本文还对抽取函数 extraction.v 及顶层驱动 driver.ml 的一部分进行了修改。

形式化验证能够保证所生成的语法分析器代码的正确性。但是，人为编写的部分仍然有发生错误的可能，例如拼写错误、逻辑错误，以及文法改写过程中产生的不符合 Lustre* 原有语法规范定义的规则等。我们能够确保在不发生这种人为错误的前提（即所编写的文法与源文法的语法和语义能够完全对应）下，经过形式化验证的语法分析器是正确的（即满足第 1 节所述的 4 条性质）。要保证这一前提成立，则必须进行较全面的覆盖范围测试。因此，我们对本文实现的语法分析器进行了覆盖范围测试，以进一步保证其符合实际需求的广义正确性，而不仅仅是语法分析过程本身的正确性。

本文利用开源 L2C 项目中目前提供的近 400 个开放测例进行测试，分别使用未经验证语法分析器的原 L2C 编译器和本课题所实现的语法分析器的新版 L2 编译器对这些测例进行编译，并对编译产生的后缀名为“.c”的文件进行比对，若比对结果一致，则测试通过，否则需要对两个编译器所产生的不一致结果进行检查，以便定位问题，进行双向互测。具体过程是：首先，在 L2C 项目中有一个对于给定测试集已确认生成结果的正确性的语法分析器，它能够将语法分析生成的节点树以给定的格式打印出来。例如，对于下面这段代码：

```
function fun1(var1:int)
returns(y1:int)
let
y1=var1+1;
tel
```

语法分析器生成的语法分析树打印为：

```
Function
    (fun1,37)
    Params
        (var1,25):int
    Return
        (y1,31):int
    Equations
        y1=(var1+1)
```

两个语法分析器的打印函数是完全一致的。因此，对于已有的语法分析器和经过了形式化验证的新型语法分析器，只需分别打印出它们所生成的节点树，再使用文本对照工具进行对比，即可验证新的语法分析器的正确性。当前开源版 L2C 的语法分析器是不久前才开发出来的（领域版 L2C 的语法分析器基于图形化的 IDE，因版权限制无法开源），可与本文的形式化验证的语法分析器实现双向互测试。目前，双向互测工作已经结束，两个前端选项的测试结果完全一致。

另外，我们分别使用原版和新版编译器对上面提到的近 400 个开放测例进行编译，并在命令行下统计编译所需的总时间，得到的结果为：原版 L2C 总编译用时 1.768 s，新版 L2C 总编译用时 3.174 s。可以看到，经过形式化验证的语法分析器的运行时间大致是原有语法分析器的 1.8～2 倍。这个数值低于 Jourdan 等给出的数值（大约为原有语法分析器的 5 倍）[5]。据 Jourdan 等分析，性能损耗主要来自两个方面：1）新的语法分析器所引入的预语法分析阶段（pre-parsing），这一部分并没有被引入 L2C 项目中；2）由于新的语法分析器是由 Coq 编写并抽取成为 Ocaml 代码的，因此需要执行一些额外的运行以便检查，而 OcamlYacc 的执行引擎是以 C 编写的，因此没有这一部分的性能开销，这一点对于所有以 Jourdan 等的方法实现的语法编译器而言都是不可避免的。我们还进行了一个简单的性能剖析，发现在词法分析步骤中引入的类型转换（Token.v）函数也会带来一定的性能损失。因此，这样的性能损耗与我们的预期大致相符。

结束语 本文参考 CompCert 编译器语法分析程序的实现，将 Jourdan 等提出的形式化验证语法分析器的方法实际应用于 L2C 可信编译器，验证了这种方法的可用性。目前，形式化验证的 Lustre* 语法分析器已经集成至开源 L2C 编译器[14]的新版本中，可以通过选项“-coq-parser”来决定是否使用该语法分析器。新的 Lustre* 语法分析器虽然比另一个未经验证的语法分析器运行时间长，但这仅发生在编译期间，对所产生的目标代码没有丝毫影响。

本文所展示的实现过程具有很好的可复用性，读者可以根据文中所提到的步骤，以较小的修改代价构建出其他语言

的语法分析器。本文所提到的实现步骤中，语法分析器的生成器和验证器都是 Inria 研究院开发的可复用工具。读者只需根据目标语言的文法编写语法分析树的节点定义、词法分析器与语法分析器的说明文件，即可采用同样的步骤生成经过了形式化验证的语法分析器。此外，由于所生成的语法分析器具有统一的接口，读者可通过修改上层驱动将其集成至其他编译器中。

本文的侧重点在于技术环节，相关原理可参考文献[5]。直接验证 LR(1)分析过程是一项非常困难的工作，即便完成验证，其扩展性也非常差。Jourdan 所提出的后验确认(posterioriverified validation)的方法具有很好的扩展性，它并不直接验证 LR(1)自动机本身，而是对其执行结果与原始文法在语义上(能够识别的单词序列以及输出的语义值)的一致性进行确认(validation)。

在集成了 Jourdan 等的语法分析器[8]之后，CompCert 又陆续对其类型检查等前端过程实现了形式化验证。目前，CompCert 编译器已完成了除词法分析器之外，编译全链条的形式化验证。开源 L2C 可信编译器也在向这一目标发展，目前仅剩前端的某些变换尚未完成形式化验证。CompCert 和 L2C 的词法分析器都是由 Ocamllex 自动生成的，对其进行形式化验证的必要性及可行性是未来需要考虑的事项之一。前端本身还有许多需要探索的问题，如 Pottier[20]更进一步地提出了一种能够检测语法分析器本身错误的算法，并将其应用于 CompCert 的“pre-parser”阶段，而这一阶段的形式化验证工作比”parser”阶段更加复杂。如果不考虑在实际可信编译器项目中的成功应用，与形式化验证的语法分析器的基础性研究相关的工作[7,9,20-21]还有很多。

本文采用 Jourdan 等实现的形式化验证的 Lustre* 语法分析器的出发点在于它在 CompCert 项目中已经得到成功应用，这一点至关重要。其成功的实践经验使我们避开了讨论可行性的问题。我们期待本文的工作能在一定程度上对同行实践者起到类似的作用。

CompCert 编译器是目前最具影响且经过形式化验证的 C 编译器，已经在安全攸关领域得到了实际应用。同时，CompCert 编译器也是语法分析程序唯一经过形式化验证的实用 C 编译器。当前，国内已有多个团队启动了有关编译器形式化验证的研究课题，本文的工作可供国内同行参考。

参 考 文 献

[1] KNIGHT J C. Safety critical systems: challenges and directions [C]// Proceedings of the 24th International Conference on Software Engineering. 2002: 547-550.

[2] LEROYX. Formal verification of a realistic compiler[J]. Communications of the ACM, 2009, 52(7): 107-115.

[3] BERTOT Y, CASTÉRAN P. Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions[M]. Springer Science & Business Media, 2013.

[4] MORRISETT G. Technical Perspective A Compiler's Story[J]. Communications of the ACM, 2009, 52(7): 106-106.

[5] JOURDAN J H, POTTIER F, LEROY X. Validating LR (1) parsers[C] // European Symposium on Programming. Berlin: Springer, 2012: 397-416.

[6] BARTHWAL A, NORRISH M. Verified, executable parsing [C]// European Symposium on Programming. Berlin: Springer, 2009: 160-174.

[7] MCPEAK S, NECULAG C. Elkhound: A fast, practical GLR parser generator [C] // International Conference on Compiler Construction. Berlin: Springer, 2004: 73-88.

[8] CompCert home[OL]. http://compcert.inria.fr/.

[9] JOURDAN J H, POTTIER F. A simple, possibly correct LR parser for C11[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 2017, 39(4): 1-36.

[10] SHI G, WANG SY, DONG Y, et al. Construction for the trustworthy compiler of a synchronous data-flow language[J]. Ruan Jian Xue Bao, 2014, 25(2): 341-356.

[11] LIU Y, GAN Y K, WANG S Y, et al. Trustworthy translation for eliminating high-order operation of a synchronous dataflow language[J]. Ruan Jian Xue Bao, 2015, 26(2): 332-347.

[12] Scade home[OL]. http://www.ansys.com/products/embedded-software/ansys-scade-suite.

[13] SHANG S, GAN Y K, SHI G, et al. Key translations of the trustworthy compiler L2C and its design and implementation [J]. Ruan Jian Xue Bao, 2017, 28(5): 1233-1246.

[14] Open L2C home[OL]. http://soft.cs.tsinghua.edu.cn/l2c.

[15] KANG Y X, GAN Y K, WANG S Y. Comparison of two trustworthy compilers Vélus and L2C for synchronous languages[J]. Ruan Jian Xue Bao, 2019, 30(7): 2003-2018.

[16] Syntax of Lustre* for the Open Source L2C Compiler[OL]. http://soft.cs.tsinghua.edu.cn/~wang/projects/L2C/Languages/LustreStar-v6.pdf.

[17] POTTIER F, RÉGIS-GIANAS Y. Menhir Reference Manual [OL]. http://gallium.inria.fr/~fpottier/menhir/manual.pdf.

[18] Abstract Syntax of Lustre* for the Open Source L2C Compiler [OL]. http://soft.cs.tsinghua.edu.cn/~wang/projects/L2C/Languages/LustreStar-AST.pdf.

[19] POTTIER F, RÉGIS-GIANASY. The Menhir parser generator [OL]. http://gallium.inria.fr/fpottier/menhir, 2016.

[20] POTTIER F. Reachability and error diagnosis in LR (1) parsers [C]// Proceedings of the 25th International Conference on Compiler Construction. 2016: 88-98.

[21] RIDGE T. Simple, functional, sound and complete parsing for all context-free grammars[C] // International Conference on Certified Programs and Proofs. Berlin: Springer, 2011: 103-118.

LI Ling, born in 1995, undergraduate student. His main research interests include formal verification and so on.

WANG Sheng-yuan, born in 1964, Ph.D, associate professor, is a senior member of China Computer Federation. His research interests include programming languages and systems, and formal verification.