C 程序类型安全产品线的核心演算

Ferruccio Damiani
都灵大学计算机科学系（DI），意大利
都灵 ferruccio.damiani@unito.it

Daisuke Kimura东邦大学，日本千叶
kmr@is.sci.toho-u.ac.jp

Luca Paolini都灵大学计算机科学系（DI），意大利都灵
luca.paolini@unito.it

Makoto Tatsuta国立情报学研究所，日本东京
tatsuta@nii.ac.jp

本文：（1）提出轻量级 C（LC），即一种核心演算，用于形式化 ANSI C 的一个适当子集，但不包括预处理器指令；（2）定义 Colored LC（CLC），即加入 ANSI C 预处理器指令的 LC；（3）为 CLC 定义一种类型系统，以保证 C 预处理器生成的所有程序都是类型良好的 C 程序。我们认为，CLC 所提供的简明形式化也可用于教学。

Stefano Berardi 的大部分学术生涯都在都灵大学计算机科学系度过，他在那里开展了关于计算机科学的逻辑基础和基于类型的程序分析的杰出研究。多年来，他讲授过许多课程，从关于 C 语言编程的理学学士课程到关于程序分析的博士课程。因此，本文与 Stefano Berardi 的研究和教学工作密切相关。

谨以此文献给 Stefano Berardi，祝贺他迎来第 2^6 个生日。

1 引言

软件产品线（SPL）是一系列相似程序的集合，这些程序称为变体，它们具有文档详尽的共性与差异，并由共同的代码库 [2, 9, 25]生成。软件产品线的每个变体都可以由一组特征来标识，这组特征称为产品；特征则是与某项程序功能相关联的名称。

在软件产品线实现的标注式方法中，代码库中的某些代码片段会与关于特征的命题公式相关联。随后，通过从代码库中移除那些经注释的命题公式在以下条件下求值为假的代码片段，即产品中的特征取值为真、其余特征取值为假，来生成与给定产品对应的变体。C 预处理器 [6] 的 #define 和 #if 指令被广泛用于按照标注式方法编写 C 程序的软件产品线。

本文的贡献 本文的贡献有两方面：

1. 我们提出 *Lightweight C*（*LC*），一种用于 C 程序的核心演算，它对 C 的一个子集的语法和类型判定进行建模；2. 我们提出 *Colored LC*（*CLC*），一种用于 LC 程序的基于标注的软件产品线的核心演算，并配有基于产品族的类型检查系统，以保证良类型 CLC SPL 的所有变体都是良类型的 LC 程序。

T. Altenkirch、F. Barbanera、F. Damiani 和 U. de'Liguoro（编）：《逻辑与类型理论：纪念 Stefano Berardi 64 岁诞辰文集》EPTCS 441，2026 年，第 103–125 页，doi:10.4204/EPTCS.441.7

© F. Damiani等人。本作品采用知识共享署名许可协议授权。

尽管文献中已有包含用于带预处理器指令的 C 代码的类型检查器的工具链 [17, 18], ，但据我们所知，尚无针对 C 的真子集核心演算的软件产品线类型检查形式化工作。我们认为，CLC 提供的简单形式化也可用于教学。

Stefano Berardi 的大部分学术生涯都在都灵大学计算机科学系度过。在那里，他致力于计算机科学的逻辑基础和基于类型的程序分析，开展了杰出的研究，并启发和激励了学生与年轻研究人员（按时间顺序包括本文的第一作者和第三作者），还与多位研究人员合作（按时间倒序包括本文的第二作者、第四作者和第一作者）。多年来，他讲授了许多课程，涵盖了从 C 语言编程的理学学士课程到程序分析的博士课程。因此，本文的研究内容与 Stefano Berardi 的研究和教学工作密切相关。

论文组织结构 第 2 节介绍一些背景知识，并讨论软件产品线的相关工作。第 3 节通过一个示例阐述以标注式方法实现 C 程序的软件产品线；该示例将在全文中作为贯穿示例。第 4 节介绍轻量级C的语法和类型判定；第 5 节介绍 CLC 以及适用于 CLC SPL 的基于族的类型系统；第 6 节说明类型检查系统能够保证，良类型 CLC SPL 的所有变体都是良类型的轻量级C程序；第 7 节则概述未来研究的一些可能方向。

2 背景与相关工作

2.1 背景

软件产品线可划分为：(i) 特征模型（*FM*），它用特征描述各个变体，其中每个特征都是代表某种功能抽象描述的名称，而每个变体则由一组特征来标识，这组特征称为产品；(ii) 代码库（*CB*），它由依赖于语言、可复用的代码制品组成，用于构建各个变体；以及 (iii) 配置知识（*CK*），它通过指定如何根据给定的产品从代码制品中派生出相应变体，将特征模型与制品库连接起来——由此建立从产品到变体的映射，称为该软件产品线的变体生成器（*VG*）。

文献中已提出多种特征模型表示方法和软件产品线实现方法（例如，参见 [4,27]）。本文考虑特征模型的命题表示以及软件产品线实现的标注式方法。

定义 2.1（命题表示中的特征模型）。特征模型 Φ是一个二元组 (F ϕ)，其中 F 是特征的（有限）集合，ϕ 是关于 F 的命题公式（即其变量F 是 F 的元素），其语法如下：

$$\phi ::= \mathrm{F} \mid !\phi \mid \phi \&\& \phi \mid \phi \| \phi \mid 0 \mid 1$$

FM 的产品是特征集合 $p \subseteq$ F，其中 ϕ可通过将 F 中的变量赋值为 1，并将 F $\setminus p$中的变量赋值为 0 来满足。

注 2.2（关于命题公式所使用的语法）。 在定义 2.1 中，我们使用 C 编程语言语法中的逻辑运算符和常量，引入了关于特征的命题公式；这种语法也用于 #if 预处理器指令中的条件。具体来说：&& 表示合取，||表示析取，! 表示否定，0 表示假，1 表示真。

记号 2.3（命题公式的几个缩写）。 有时，为了提高可读性，我们会使用以下缩写：

- $\phi_1 \Rightarrow \phi_2$ 是 $!\phi_1 \| \phi_2$ 的简写；并且
- $\phi_1 \Leftrightarrow \phi_2$ 是 $(\phi_1 \Rightarrow \phi_2)\&\&\ (\phi_2 \Rightarrow \phi_1)$ 的简写。

根据软件产品线实现的标注式方法：

- 代码基是一个单独的程序（可能分成不同的文件），其中包含（至少）所有出现在任一变体中的代码；
- 代码知识是一种映射，它使用关于特征的命题表达式对代码基中的代码片段进行标注；并且

- 变体生成器接收一个产品 p，并从代码基中移除所有这样的代码片段：这些片段所标注的特征表达式在将 F 中 p 包含的变量赋值为 1、将 F $\setminus p$ 中包含的变量赋值为 0 时，求值结果为 0。

C 编程语言的预处理器及其指令 #define 和 #if，是软件产品线实现中标注式方法的典型代表。许多知名的软件发行版 [29]（例如 Apache、Berkeley DB、Linux、PHP、Python、SQLite、Vim 等）都可以视为 C 程序软件产品线的标注式实现 [19]。这类软件产品线通常包含数量极其庞大的产品。例如，Gentoo [12] 这一基于源代码的 Linux 发行版包含许多高度可配置的软件包；截至2019年3月1日，Gentoo 发行版包含 671617 个特性，分布于 36197 个特征模型中，因此最多可生成 2^{671617} 个产品 [21]。自然会引出一些问题，例如：

> 我们如何确保这些（最多） 2^{671617} 个变体都是类型良好的 *C* 程序？

显然，即使是特性较少的软件产品线（例如，一个包含 64 个特性的特征模型，因而最多有 2^{64} 个产品），逐个生成并独立进行类型检查也不可行。

文献中提出了若干种分析软件产品线的方法（例如， [30] 中对此进行了综述）。其中，基于族的方法 可以概括如下（参见 [30] 中的定义 4.1）：

> 如果对软件产品线的分析满足以下条件，则称其为基于族的分析：(a) 仅针对代码库运行；(b) 将特征模型和代码知识纳入分析。

基于族的方法的核心思想是将代码库与特征模型结合起来进行分析，以检查所有变体是否都满足某些预期属性，同时确保这些分析在实现上是可行的。

2.2 相关工作

我们将相关研究的讨论聚焦于本工作的两个方面：C 语言的核心语言，以及用于软件产品线标注式实现方法的核心演算。

关于 C 的核心语言 尽管 C 只有少数几个关键字，但它是一门相当微妙的语言。由于未定义行为的空间很大，鲜有程序员能详细掌握它 [13]。Chucky Ellison 在其博士论文中给出了 **C11** 语言语义的完整形式化描述 [10]（另见 [11]）；该论文回顾了为 C 提供形式语义的种种尝试。

文献中提出了多种 C 的核心语言，例如：Featherweight C [28], 、一种“简单类C语言” [15], 以及 Clight [5]。我们设计轻量级C（LC），旨在构造一种演算，使其与 C 的关系类似于 λ演算 [8]（另见 [3]）或 PCF [24] 与 ML、Haskell 等程序设计语言之间的关系，也类似于 Featherweight Java（FJ） [14] 与 Java 之间的关系。上述 C 核心语言都不满足这一要求：Featherweight C [28] 和“简单类C语言”一方面过于简单（它们不对结构体进行建模），另一方面又过于复杂（它们对函数的局部变量和语句进行建模）；而 Clight 则显然过于庞大。

值得注意的是，以下两种关系之间存在显著差异：(1) λ演算或 PCF 与 ML 或 Haskell 之间的关系；(2) FJ 与 Java 之间的关系。具体而言，λ演算和 PCF 是图灵完备的，而 FJ 的设计目标并非使其图灵完备（据我们所知，它也确实并非图灵完备）。轻量级C（假设整数大小不受限制）是图灵完备的，并且通过加入整数、对结构体成员的命令式赋值、条件表达式、表达式序列、运算符，以及堆上的动态内存分配与释放，它在一定程度上超出了 FJ 的设计理念——我们认为超出得并不多——因此我们为它选择了这个名称。

关于软件产品线实现的标注式方法的核心演算 尽管：

- 已有研究提出了一种基于最小核心演算 Colored Featherweight Java (CFJ)，对 Java 程序的基于标注的软件产品线进行基于产品族的类型检查的形式化描述；该演算形式化了对（无类型转换的）Featherweight Java (FJ) 软件产品线进行基于产品族的类型检查 [16],；并且 [14]

- TyeChef，[1] 一种基础设施，其中包含一个用于 C 程序的软件产品线的类型检查器，这些软件产品线通过 #define 和 #ifdef C 预处理器指令实现可变性。该基础设施已被提出 [18],并形式化 [17]，其基础是一个用于模块系统的核心演算 [7],，并被用作实现多种软件产品线分析的基础（例如，参见 [20, 22, 26]）；

据我们所知，尚无研究基于一个严格属于 C 子集的核心语言，对基于标注的软件产品线进行基于产品族的类型检查形式化。我们对 C 程序软件产品线实现的标注式方法进行形式化，其灵感来源于 Kastner 等人在关于 CFJ 的论文中提出的形式化方法 [16]。关于标注式方法实现软件产品线的核心演算相关工作，请参阅该论文；关于软件产品线实现方法 [27] 和软件产品线分析策略 [30]的更广泛讨论，则可参阅几篇综述论文。CLC 与 CFJ 的一个关键区别在于，LC 对命令式变量（结构体成员）赋值以及堆上的动态内存分配与释放进行了建模；因此，CLC 为研究 C 程序软件产品线的基于产品族的分析提供了适当的基础，例如堆安全性分析。

3 运行示例

作为一个运行示例，我们考虑一种由链表实现的整数先进先出队列软件产品线，称为队列软件产品线。队列软件产品线的所有变体都定义了一个名为fifo的结构体，它包含一个成员first，其类型为结构体节点*，用于保存包含队列元素的链表（即指向链表首个节点的指针；如果队列为空，则为空指针），以及一些函数。

队列 SPL 有三个特征和六个产品（因此有六个变体）。这些特征是：

[1] https://github.com/ckaestne/TypeChef

```
1struct节点{2intdata;3struct节点*next;4};5structfifo{6
st
fi
n!
fi
et
```

图 1：队列软件产品线的变体 1（对应产品 $p_1 = \varnothing$）。

- LAST，表示名为 fifo 的结构体有一个成员 last，其类型为 struct node*，用于指向包含队列元素的链表的最后一个元素（如果队列为空，则为 空指针）；

- OCCURS，表示存在一个函数 int occurs(struct node* list， int element)，用于检查给定的链表是否包含给定元素；以及
- INTERVAL，表示存在一个函数 int occurs(struct node* list， int start， intend， int element)，用于检查给定的链表在给定位置 start 和 end 之间是否包含给定元素（链表的第一个元素位于位置 0）。

队列软件产品线的六个变体（C 程序）由特征模型Φ_{queue} = (F^{queue}， ϕ_{queue})描述的六种产品来标识，其中：

- $\mathcal{F}_{队列}$ = {LAST,OCCURS,INTERVAL}，以及
- $\phi_{队列}$ = ! INTERVAL || OCCURS;

它们是 F^{queue} 中满足命题公式 ϕ_{queue} 的特性组合。具体来说，它们是：

1. $p_1 = \varnothing$，空特性集，对应图 1 中的变体；2. p_2 ={LAST}，对应图 2 中的变体；3. p_3 = {OCCURS}，对应将注释替换为变体 1 的代码后，由图 3 中的代码得到的变体；4. p_4 = {LAST,OCCURS}，对应将注释替换为变体 2 的代码后，由图 3 中的代码得到的变体；5. p_5 = {OCCURS,INTERVAL}，对应将注释替换为变体 1 的代码后，由图 4 中的代码得到的变体；6. p_6 = {LAST,OCCURS,INTERVAL}，对应将注释替换为变体 2 的代码后，由图 4 中的代码得到的变体。

队列软件产品线的代码库如图 5 所示，其中每个特性都有一个#define预处理器指令，而给定产品所标识的变体可以通过注释掉产品中未包含的特性对应的#define 预处理器指令来生成。值得注意的是，代码库中包含一些代码（即图 5 第 26 行的逗号），这些代码并非

```
1struct节点{2intdata;3struct节点*next;4};5structfi
fo
uc
uc
ur
```

图 2：变体 2（针对产品 p_2 ={LAST}）的队列软件产品线。

```
1//插入变体 1 的代码以获得变体 3；插入变体 2 的代码以获得变体 4234整型 出现 (结构体 节点 *列表 ,整型 元素 )4
{3
```

图 3：队列软件产品线的变体 3（对应 p_3 = {OCCURS}）和 4（对应 p_4 = {LAST,OCCURS}）。

在任何变体中都包含；这样做是为了确保从代码库中移除所有预处理器指令后得到的代码是格式良好的 C 程序。该程序并非队列软件产品线的变体，见图 6。

备注 3.1（若干注解式 C 软件产品线编程模式）。图 5 第 22 至 30 行的代码是以下注解式 C 软件产品线编程模式的一个实例。

模式 1 若要为一个表达式提供$n \geq 2$ 个备选版本（以纳入不同变体的代码中），请按如下方式操作。首先，给出$e1,\ldots,en$这一n个表达式的序列。然后，使用恒等于 0 的命题公式（例如，直接使用 0）注解 e_j与e_{j+1}之间的逗号$(1 \leq j \leq n-1)$。最后，使用合适的、关于特征的命题公式 ϕ_i 注解每个表达式e_i $(1 \leq i \leq n)$；也就是说，这些公式 ϕ_i 必须两两互斥（在特征模型的约束下），并且至少有一个公式始终成立（同样是在特征模型的约束下）。

我们并未用“0”标注第 26 行的逗号，而是用公式“LAST && !LAST”对其进行了标注。这使得第 22-30 行的代码也成为以下更通用的标注式 C SPL 编程模式的一个实例：

模式 2 考虑一个序列（$e1$，…，en），它由 $n \geq 2$ 个表达式组成。为每个表达式 e_i（$1 \leq i \leq n$）标注一个关于特征的命题公式 ϕ_i，使得（模特征模型 FM）这些ϕ_i’中至少有一个始终成立。标注 $n-1$ 个逗号，这些逗号分隔 n 个表达式，即序列（$e1$，…，en）中的那些表达式，并分别配以 $n-1$ 个公式 ψ_i（$1 \leq i \leq n-1$），其中 ψ_i定义为：

$$\psi_i = \phi_i \;\&\&\; (||_{i<j<n}\phi_j) = \phi_i \;\&\&\; (\phi_{i+1} \,||\cdots||\, \phi_{n-1})\,.$$

这确保了：每当 m 个表达式（$m < n$）——即 e_i（$1 \leq i \leq n$）中的表达式——被移除时，便会以相应方式移除 m 个逗号，使剩余代码形成一个良构的表达式序列。

4 轻量级 C

本节将介绍轻量级 *C*（*LC*），一种面向 C 程序的核心演算。

1 / / 插入代码：将变体 1 的代码插入变体 5，将变体 2 的代码插入变体 6 2 3 整数 occurs（结构体节点 * 列表，4 整数

图 4：队列软件产品线的变体 5（对应 $p_5 = \{\text{OCCURS},\text{INTERVAL}\}$）和变体 6（$p_6 = \{\text{LAST},\text{OCCURS},\text{INTERVAL}\}$）。

4.1 轻量级C 语法

轻量级C 的抽象语法如图 7 所示。遵循 [14, 23], , 我们采用上划线记法 $\cdot$表示（可能为空的）序列。我们使用一元函数 #(·) 表示序列的长度。具体而言，“SD”、“T m;”、“T x”和“e”分别表示结构体定义序列“$\text{SD}_1 \cdots \text{SD}_n$”（$\text{n} = \#(\text{SD})$）、结构体成员声明序列“$\text{T}_1\,\text{m}_1; \ldots \text{T}_n\,\text{m}_n;$”（$\text{n} = \#(\text{T m;})$）、函数形参声明序列“$\text{T}_1\,\text{x1}, \ldots, \text{T}_n\,\text{m}_n$”（$\text{n} = \#(\text{T x})$）以及表达式序列“$\text{e1}, \ldots, \text{e}_n$”（$\text{n} = \#(\text{e})$）。空序列用“●”表示，非空表达式序列用 ~e 表示。我们假设命名实体列表（即结构体定义、结构体成员声明、函数定义、函数形参声明）中的元素名称各不相同。

表达式“MALLOC(结构体 s)”是“(结构体 s*) malloc (sizeof (结构体 s)))”的缩写，其中 malloc 是用于内存分配的标准库函数；[2] 而MFREE 是标准库函数 free的一个变体，用于内存分配，其返回类型为“空类型*”（而非“空类型”），并且总是返回空指针。

备注 4.1 (关于轻量级C语法和 ANSI C 语法)。 图 7 中的语法是 C24 标准所定义语法的一个子集。特别要指出的是，根据 [1] 第 6.5.18 节（逗号运算符），~e可以作为单个实参，以非空的带括号表达式序列形式出现在函数调用中；例如，给定原型为“ 整数 f(整数 x) ;”的函数，我们可以这样调用它：“f((2,4))”，这等价于“f(4)”。请注意，无需预先提供结构体名称的定义，即可引用结构体名称[3], , 从而允许递归类型定义和相互递归类型定义。

程序由一系列结构体声明和一系列函数声明组成，后者始终包含一个main 函数，该函数不接受实参并返回一个整数。

为简化表述，以下我们以显而易见的方式，将结构体定义 SD视为从每个成员名称 m 到成员声明 Tm的映射；将一系列结构体定义SD 视为从每个结构体名 s 到结构体定义 SD的映射；将一系列函数定义FD 视为从函数名 f 到函数定义 FD的映射；并将程序 Prg 视为映射 SD 和FD的并集。 — —

[2]我们可以假设，每个 LC 程序都隐式地以如下指令开头，该指令定义了参数化宏：“#defineMALLOC(a) (a*) malloc(sizeof (a))”。[3]这是可行的，因为在 [1, 第 6.2 节5] 中解释道：“所有指向结构体类型的指针彼此之间都应具有相同的表示形式和对齐要求。”另一方面，不同种类的指针可能具有不同的宽度和／或对齐属性。

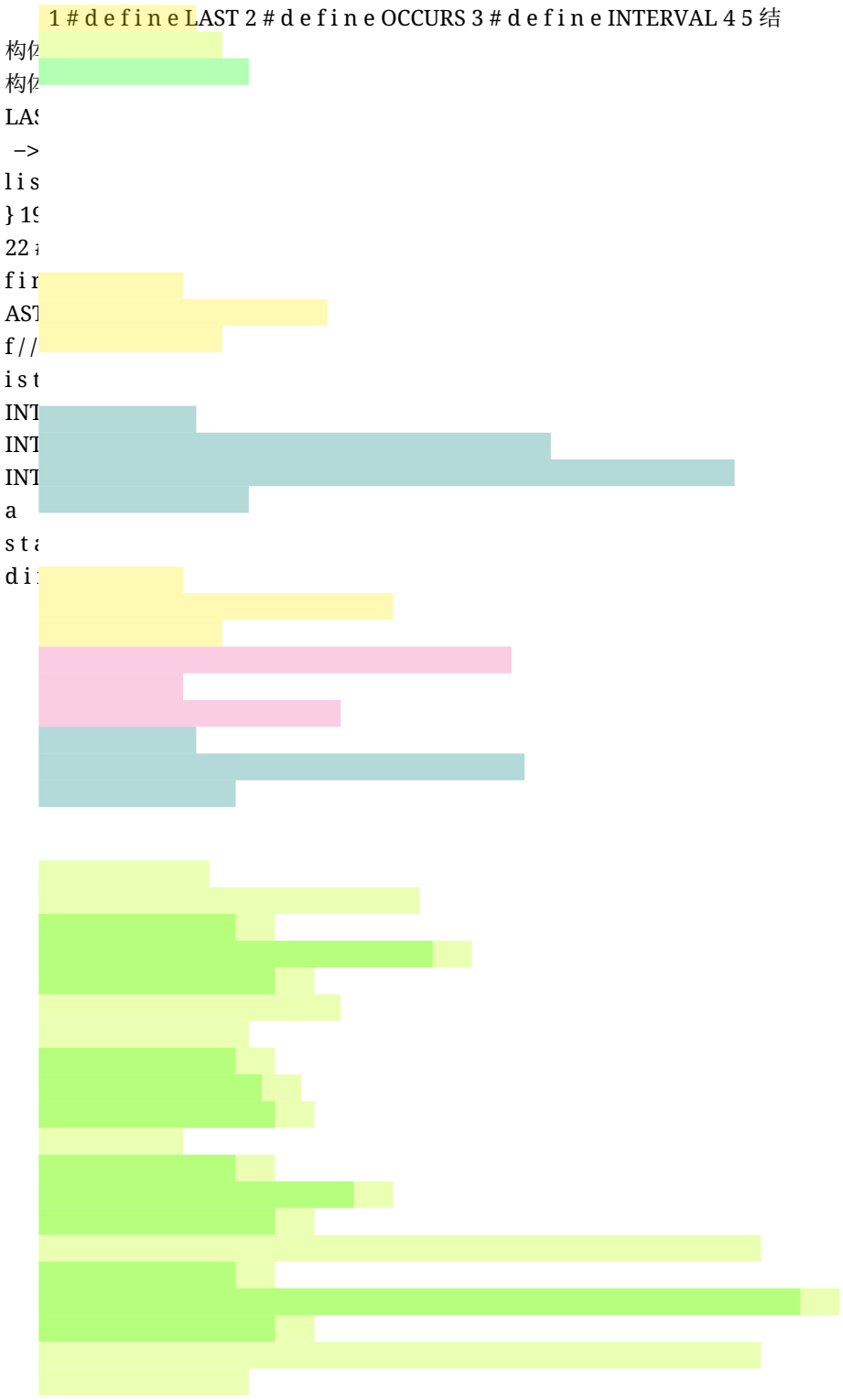

图 5：队列软件产品线的代码库，其中与以下注解相关的代码LAST 、OCCURS 、R Γ .ST （ 突出显示。

```
1  struct  节点 {
2      int  data ;
3      struct  节点 *  下一个 ;
4  };
5  结构体  菲福  {
6      结构体  节点 *  首个 ;
7      结构体  节点 *  末尾 ;
8  };
9  整数  前端 ( 结构体  信息*  队列 ) { 返回  队列 -> 首个 -> 数据; }
10 结构体  节点*  获取最后一个( 结构体  节点 *  链 表 )
11  { 返 回  !( 链 表 -> 下 一 个 ) ?  链 表  :  取 最 后 一 个 ( 链 表 -> 下 一 个 ); }
12 整型  后( 结构体  先进先出  队列 )
13 {  retur  (
14          队列 ->last ->data
15          ,
16          获取最后一个 ( 队列 -> 第一个 ) -> 数据
17          ) ;
18 }
19 整数  出现 ( 结构体  节点 *  列表 ,
20          整数  开 始 ,  整 数  end ,
21          整 数  元 素 )
22  { 返 回  !( 列 表 )
23      &&结束  >= 0
24      && ( (
25          开始  <= 0 &&
26          列表 -> 数据  ==  元素 )  ||  出现 ( 列表 -> 下一个 ,
27                                  开始 -1 , 结束 -1,
28                                  元素 )) ; }
29  整型  main ( ) { 返回  0 ; }
```

图 6：从队列软件产品线代码库中移除预处理器指令后得到的 C 程序，其中与注解 LAST 、OCCURS、INTERVAL 、!LAST 和LAST && !LA[illegible] 即[illegible]的[illegible]显示。

Prg	::=	SD FD	程序
SD	::=	结构体 s{Tm;};	结构体定义
FD	::=	T f(T x) { 返回 e; }	函数定义
T	::=	整数 | 空类型* | 结构体 s*	Type
e	::=	n | 空指针 |x | f (e) | e->m | e->m=e | e?e:e | (e) | 一元运算符 e | e 二元运算符 e | MALLOC(结构体 s) | MFREE(e)	表达式

图 7：轻量级C 抽象语法，其中 s 表示结构体名称，m 表示结构体成员名称，f 表示函数名称，x 表示函数形参名称，n 表示整数值，uop 表示一元运算符，而bop 表示二元运算符。

Name	Type	描述
一元运算符		
–	(整数)→整数	一元负号运算符
!	(T) →整数	返回其参数是否不等于 0 和空指针
二元运算符		
+, –, *, /, %	(整数,整数)→整数	算术运算符
&&, ||	(整数,整数)→整数	逻辑运算符
<, <=, >, >=	(整数,整数)→整数	整数关系运算符
==, !=	(T,T)→整数	相等与不等运算符

图 8：LC 运算符。

我们用“Prg SANE”表示程序 Prg = SD FD 满足以下健全性条件：(1) 对于 Prg 中任意位置出现的每个结构体名 s，均有 s ∈dom(SD)；(2) 对于 FD 中任意位置出现的每个函数名 f，均有 f ∈dom(FD)；以及 (3) FD(main) = int main(){ return e;}，其中 e 为某个表达式。

示例 4.2（一些轻量级C 程序）。图 1 至图 4 中展示的六个程序都是轻量级C 程序。

4.2 轻量级C 类型判定

一元和二元运算符及其类型和简要说明列于图 8 中。给定一个（一元或二元）运算符 op，我们用 type(op) 表示其类型。

按照惯例， [23], 类型环境 Γ 是从函数形参名称到类型的有限映射，记作“x :T”，通常简写为“x_1 :T1, ..., x_n :T_n”，其中 n ≥ 0。在图 9 中，我们定义了三种定型判断：用 ⊢Prg OK 表示 Prg 是良类型的；用 ⊢ FD OK 表示 dom(FD) 中定义的函数是良类型的；用 Γ ⊢ e :T 表示在环境 Γ 的假设下，表达式 e 的类型为 T。符号 ≤ 表示子类型关系，它是由 void∗ ≤ struct s∗ 所定义关系的自反闭包，其中 struct s 遍及程序中定义的所有结构体。我们用 max≤{T_1,T_2} 表示类型 T_1 和 T_2 中较大的一个。

这些类型判定规则相当标准。按照规则 T-PRG 的前提中的写法，我们用“ [23], FD OK”简写“⊢ FD_1 OK ··· ⊢ FD_n OK”；在规则 T-APP 的前提中，我们用“Γ ⊢ e :T”简写“Γ ⊢ e_1 :T_1 ··· Γ ⊢ e_n :T_n”，规则 T-SEQ 的前提“Γ ⊢ ˜e :T”也采用类似写法；而在规则 T-APP 的前提中，我们用“T′ ≤T”简写“T'_1 ≤T_1 ··· T'_n ≤T_n”。需要指出的是，T-MALLOC 要求 s ∈dom(SD)，尽管规则 T-PRG 中的合理性条件已确保这一点；这只是为了简化系统的一些表述。

例 4.3（一些类型正确的轻量级C 程序）。 图 1 至图 4 中展示的六个程序都是类型正确的轻量级C 程序。

事实 *4.1*（关于轻量级*C* 和 *ANSI C* 的类型判定）。设 *FD* =*T f(T x) {···}* 是一个函数定义，令 *prototype(FD)* 表示 *T f (T x) ;*，并令 *prototype(FD)* 表示 *prototype*(FD_1)···*prototype*(FD_n)。如果 *SD FD* 是一个类型正确的轻量级*C* 程序，即 ⊢ *SD FD OK* 成立，那么 *SD prototype(FD) FD* 是一个类型正确的 *C24* 程序 *[1]*。

程序类型判定 $\vdash$程序 确定

$$\frac{\text{程序 SANE} \qquad \text{程序 } =\overline{SD}\ FD \qquad \vdash FD \text{ 确定}}{\vdash \text{程序 OK}}\ \text{(T-PRG)}$$

函数类型判定 $\vdash FD$ 确定

$$\frac{\overline{x} : \overline{T} \vdash e : T_1 \qquad T_1 \leq T_0}{\vdash T_0\ f(\overline{T\ x})\{\ \text{返回 } e;\}\ \text{正确}}\ \text{(T-FUN)}$$

表达式类型判断 $\Gamma \vdash e : T$

$$\frac{}{\Gamma \vdash n : \text{整数}}\ \text{(T-INT)} \qquad \frac{}{\Gamma \vdash \text{空指针} : \text{空类型}*}\ \text{(T-NULL)} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{(T-PAR)}$$

$$\frac{\text{程序}(f) = T_0 f(\overline{T\ x})\{\ldots\} \qquad \Gamma \vdash \overline{e} : \overline{T'} \qquad \overline{T'} \leq \overline{T}}{\Gamma \vdash f(\overline{e}) : T_0}\ \text{(T-APP)} \qquad \frac{\Gamma \vdash e_0 : \text{结构体 } s* \qquad \text{程序}(s)(m) = T\ m}{\Gamma \vdash e_0\texttt{->}m : T}\ \text{(T-MEMBER)}$$

$$\frac{\Gamma \vdash e_0 : \text{结构体 } s* \qquad \text{程序}(s)(m) = T\ m \qquad \Gamma \vdash e_1 : T_1 \qquad T_1 \leq T}{\Gamma \vdash e_0\texttt{->}m = e_1 : T}\ \text{(T-ASSIGN)}$$

$$\frac{\Gamma \vdash e_0 : T_0 \qquad \Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2 \qquad T_3 = \text{最大}_{\leq}\{T_1, T_2\}}{\Gamma \vdash e_0 ?\ e_1 :\ e_2 : T_3}\ \text{(T-COND)}$$

$$\frac{\tilde{e} = e_1, \ldots\ldots, e_n \qquad \Gamma \vdash e_1 : T_1 \quad \ldots \quad \Gamma \vdash e_n : T_n \qquad T = T_n}{\Gamma \vdash (\tilde{e}) : T}\ \text{(T-SEQ)}$$

$$\frac{\Gamma \vdash e_0 : T_0 \qquad \text{类型}(\texttt{uop}) = (T_0) \to \text{整数}}{\Gamma \vdash \texttt{uop}(e_0) : \text{整数}}\ \text{(T-UOP)} \qquad \frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2 \qquad T_3 = \text{最大}_{\leq}\{T_1, T_2\} \qquad \text{类型}(\texttt{bop}) = (T_3, T_3) \to \text{整数}}{\Gamma \vdash e_1\ \texttt{bop}\ e_2 : \text{整数}}\ \text{(T-BOP)}$$

$$\frac{s \in \text{dom}(\overline{SD})}{\Gamma \vdash \texttt{MALLOC}(\text{结构体 } s) : \text{结构体 } s*}\ \text{(T-MALLOC)} \qquad \frac{\Gamma \vdash e_0 : \text{结构体 } s*}{\Gamma \vdash \texttt{MFREE}(e_0) : \text{空类型}*}\ \text{(T-MFREE)}$$

图 9：轻量级C 类型判定。

Prg	::=	SD F	程序
SD	::=	结构体 s{ }	结构体定义
FD	::=	T f( ) { 返回 e; }	函数定义
T	::=	整数 | 空类型* | 结构体 s*	Type
e	::=	n | 空指针 | x | f( ) | e->m | e->m=e | e?e:e | (~e)	
		| uop e | e bop e | MALLOC(结构体 s) | MFREE(e)	表达式

图10：轻量级C语法（与图 7 相同），其中 灰色高亮 的部

5 彩色轻量级C

本节我们介绍*Colored LC*（*CLC*），一种面向SPLsof (L)C程序的核心演算。为表示注解，我们采用为Colored Featherweight Java（CFJ）提出的方法 [16]。

5.1 CLC语法

C 预处理器允许删除任意代码片段（甚至是始于某个函数体、止于另一个函数体的片段），从而使该过程容易产生语法错误。与CFJ中一样 [16],，在CLC中，我们对可删除的内容采用严格的约束。具体而言，我们允许程序员删除其中的部分以灰色突出显示的图10中的元素：任意结构体定义、任意结构体成员、任意函数定义、函数定义的任意形式参数、函数调用的任意实参，以及表达式序列中的任意表达式。值得注意的是，在具体语法中，表达式链表包含可加注解的逗号，但在抽象语法中我们忽略它们的存在：可以按照（参见备注 3.1）的模式 2来处理它们，并且可以将它们统一作为链表元素进行管理。

我们通过使用注释表对代码基中的代码注释进行建模。注释表将每个可注解代码片段的出现位置（由其在代码基中的位置标识，例如在文本或语法树中的位置）[4] 映射为一个关于特征的命题公式，用以指定生成变体时是否必须保留该代码片段。因此，表示一个由 LC 程序组成的软件产品线的 CLC 程序，是一个三元组 (Φ,程序,属性变换)，其中：

- Φ 是一个特征模型（参见定义 2.1），
- 程序 是一个 LC 程序，表示该代码基，并且
- 属性变换是一个注释表，用于表达代码知识。

为简化形式化描述，下文中我们始终假定存在一个固定的软件产品线 (Φ,程序,属性变换)。

我们使用元变量 a 和 b 来表示任意可注释的代码片段。对于 CB 中的每个可注释片段 a 以及 FM 的每个产品 p，命题公式 AT(a) 满足：如果$p \vDash$ AT(a（其中 p 作为一种解释，将 p 中的特征映射为 1，并将其他特征映射为 0），那么生成p的变体时必须保留代码片段 a；否则必须将其移除。没有注释的片段默认注释为常量 1（即存在于所有变体中）。

[4]因此，同一代码片段的不同出现位置可以具有不同的注解。

注释 5.1 (关于 CLC 注释表和 #if 预处理指令)。 显然，注释表所指定的 CK 信息可以通过 #if 预处理指令实现（参见第 3 节）。

注释 5.2 （CLC 注释表的语法糖）。 在这些示例中，我们假设也可以将二元逻辑运算符的任意出现与其左操作数或右操作数之一一并注解；也就是说，每个表达式“e_1 bop e_2”中的出现（其中 bop 是二元逻辑运算符）都可以采用以下任一种形式：左注解形式“ 左注解 为 “ e_1 bop e_2” 或右注解 为“e_1 bop e_2”。值得注意的是，这类注解：

- 可以通过使用 #if 预处理器指令来表示，正如图 5 中第 40–42 行和第 44–46 行的代码所示；并且
- 可以通过在注释表的定义中引入一些复杂性来形式化。

为了使介绍更简单，我们倾向于不对二元逻辑运算符的左注解和右注解进行形式化，而是考虑：

- “e1 bop e2”中，AT(e1 bop) = ψ 可视为表达式序列“(e1 bop e2，e2)”的语法糖，其中 AT(e1 bop e2) = ψ 且 AT(e2) =!ψ；并且
- 将“e1 bop e2”及 AT(bop e2) = ψ 作为表达式序列“(e1 bop e2,e1)”的语法糖，其中 AT(e1 bop e2) = ψ 和 AT(e2) =!ψ。

示例 5.3 （队列软件产品线）。 设 (Φ_{queue},程序$_{queue}$,属性变换$_{队列}$) 为满足以下条件的 CLC 软件产品线：

- $\Phi_{队列}$ 是第 3 节开头定义的特征模型；
- 程序$_{队列}$ 是图 6 中的轻量级C 程序——忽略颜色；并且
- 属性变换$_{队列}$ 是由图 6 中的颜色表示的注释表，其中：第 23 行和第 25 行包含使用备注 5.2 中所述语法糖表示的注解；第 27 行包含以下两个注解 属性变换$_{队列}$(start−1) =INTERVAL 和 属性变换$_{队列}$(end−1) = INTERVAL。

这对 (程序$_{queue}$,属性变换$_{队列}$) 的 ANSI C 编码见图 5，其中，图 6 第 27 行的两个注解（属性变换$_{队列}$(start −1) = INTERVAL 和 属性变换$_{队列}$ (end−1) =INTERVAL ）由图 5 第 48–50 行中的一条 #ifdef 预处理指令表示。

记法 5.4（涉及注释表的命题公式的某种缩写）。

- 我们用“AT(¯a) $\odot$AT(¯b)”，其中 $\odot \in \{\Rightarrow, \Leftrightarrow\}$（参见记法 2.3），作为简写，¯这也表示序列$a$ 和b 长度相同，$(\mathrm{AT}(a_1) \odot \mathrm{AT}(b_1)) \&\& \cdots \&\& (\mathrm{AT}(a_n) \odot \mathrm{AT}(b_n))$”

- 我们用“属性变换(T)”作为简写，表示–“1”，如果 T $\in$ { 整数 ,空类型$_*$}，这表示类型 整数 和 空类型$_*$ 始终存在；以及–“属性变换(程序(s))”，如果 T = 结构体 s$_*$，这表示结构体 s 存在。
- 我们用“$\exists$(˜e)”作为“属性变换(e_1)||···||属性变换(e_n)”的简写，这表示至少有一个 e_i 属于 ˜e，使得 属性变换(e_i) 为真；也就是说，在某个变体的生成过程中，至少有一个 ˜e 表达式没有被移除。

- 我们用“neverLast(k ¯e)”作为“!属性变换(e_k)||属性变换(e_{k+1})|| . . . ||属性变换(e_n)”的简写，它表示在每个变体中，e_k 永远不会是 ¯e的最后一个（未被抑制的）表达式；也就是说，如果它在生成某个变体时未被移除，那么至少还有一个后续表达式在该变体的生成过程中未被移除。

5.2 CLC 类型判定

为了保证 CLC 软件产品线的所有变体都是类型正确的 LC 程序，我们为 CLC 软件产品线定义了一种基于族的类型系统。

一个带注释的类型环境 Δ 是从变量到类型与特性上的命题公式对的有限映射，记作$x : T$ with ψ，通常简写为$x_1 : T_1$ with $\psi_1, \ldots, \overline{x_n : T_n}$ with $\overline{\psi_n}$，其中 $n \geq 0$。基于产品族的类型判断如下：

- $\phi \vdash$ 程序 OK，读作“该软件产品线（Φ,程序,属性变换）在 $\Phi = (F \quad \phi)$ 下是良类型的”；
- $\theta \vdash$ SD OK，读作“SD 在配置假设下是良类型的[5] θ”；
- $\theta \vdash$ FD OK，读作“FD 在配置假设 θ下是良类型的”；并且
- θ； $\Delta \vdash$ 表达式 $: T$，读作“表达式 e 在配置假设 T 下具有类型 θ以及带注释的类型环境中的 e 假设 Δ”。

基于产品族的类型规则见图 11，其中我们使用记法 5.4 中引入的缩写；在定型规则 FT-PRG 的前提中，我们将“ϕ&&属性变换(SD) $\vdash$ SD OK”作为“ϕ&& 属性变换($\overline{SD_1}$) $\vdash \overline{FD_1}$ OK $\cdots$ ϕ&&属性变换(SD_n) $\vdash FD_n$ OK”的缩写，其中n是序列SD的长度；对于“ϕ&& 属性变换(FD) $\vdash$ FD OK”也是如此。通常， $\theta \vDash \theta'$表示命题公式 $\theta \Rightarrow \theta'$有效（即该公式被其变量的所有真值赋值满足）。

CLC 基于产品族的类型系统（它将整个 CLC SPL 作为整体进行类型判定）除了轻量级C 类型系统（它对单个轻量级C 程序进行类型判定）执行的标准检查之外，还会检查一些存在于变体中的 条件。例如，我们必须表达这样的条件：“只要代码片段a 存在于某个变体中，那么代码片段b 也存在于同一变体中”，或者换言之：“只要a 的注释（相对于某个产品p）求值为 1，那么b 的注释（相对于同一产品p）也求值为 1”。

也就是说，每条基于产品族的类型检查定型规则都是在一条 LC 类型判定规则（参见图 9）的基础上，通过添加涉及 [redacted] 的检查（以灰色突出显示）得到的。具体而言，每条基于产品族的定型规则 FT-<NAME> 都由定型规则 T-<NAME> 得到。

CLC SPL 基于产品族的类型检查所检查的附加条件如下。

P1（规则 FT-PRG 中）：函数 main 必须存在。S1（规则 FT-STRUCT 中）：如果结构体 s 的成员 T m 存在，则其类型 T 也必须存在。F1（规则 FT-FUN 中）：如果函数定义 T_0 f 存在，则其返回类型 T_0 也必须存在。F2（规则 FT-FUN 中）：如果某个已存在函数的实参 T x 存在，则其类型 T 也必须存在。E1（规则 FT-PAR 中）：如果某个已存在函数的函数体中出现了形式参数 x，则该函数头部中对 x 的声明必须存在。E2（规则 FT-APP 中）：如果函数调用表达式 f($\overline{e}$) 存在，则：(i) 函数 f 必须存在；(ii) 实参链表 $\overline{e}$ 中表达式的类型和数量必须与 f 的定义中的形式参数列表相匹配。E3（规则 FT-MEMBER 和 FT-ASSIGN 中）：如果表达式 e–>m 存在，则 e 的类型所对应的结构体 s（即 struct s$*$ ）必须存在，且包含成员 m。

E4（规则 FT-SEQ中）如果存在一个表达式序列 （e_1，...，e_n），则该序列中至少存在一个表达式。

[5]我们将 用于基于产品族的类型检查的命题公式称为配置假设。

软件产品线 基于产品族的类型检查

$\phi \vdash$程序 OK

程序 SANE　属性变换 (main) = 1　程序 = SD FD　ϕ && 属性变换(SD) $\vdash$ SD 正常　ϕ && 属性变换(FD) $\vdash$ FD 确定

(FT-PRG)

$\phi \vdash$程序 确定

基于结构产品族的类型检查

$\theta \vdash$SD 确定

$\theta \vDash$ 属性变换(T m) $\Rightarrow$ 属性变换($\overline{\text{T}}$)

(FT-STRUCT)

$\theta \vdash$ 结构体 s{$\overline{\text{T m}}$}OK

函数族类型规则

$\theta \vdash$FD 确定

$\theta \vDash$属性变换(T_0)　$\theta \vDash$ 属性变换(T x) $\Rightarrow$ 属性变换 $\overline{\text{(T)}}$　θ; $\overline{\text{x}}$: $\overline{\text{T}}$ $\vdash$ e : T_1　$T_1 \leq T_0$

(FT-FUN)

$\theta \vdash T_0$ f($\overline{\text{T x}}$){ 返回e;} OK

表达式族类型规则

θ; $\Delta \vdash$ e : T

(FT-INT)　θ; $\Delta \vdash$ n : 整数

(FT-NULL)　θ; $\Delta \vdash$ 空指针 : 空类型*

(FT-PAR)　x : T Δ　$\theta \vDash \psi$ / θ; $\Delta \vdash$ x : T

程序(f) = T_0 f($\overline{\text{T x}}$){...}　$\theta \vDash$属性变换(程序(f))　θ; $\Delta \vdash$ e $\overline{\text{T}}'$　$\overline{\text{T}}' \leq \overline{\text{T}}$　$\theta \vDash$ 属性变换(e) $\Leftrightarrow$ 属性变换 $\overline{\text{(T x)}}$

(FT-APP)　θ; $\Delta \vdash$ f($\overline{\text{e}}$) : T_0

程序(s)(m) = T m　θ; $\Delta \vdash e_0$: 结构体s*　$\theta \vDash$属性变换(T m)

(FT-MEMBER)　θ; $\Delta \vdash e_0$->m : T

θ; $\Delta \vdash e_0$: 结构体s*　程序(s)(m) = T m　θ; $\Delta \vdash e_1$: T_1　$T_1 \leq$ T　$\theta \vDash$属性变换(T m)

(FT-ASSIGN)　θ; $\Delta \vdash e_0$->m = e_1 : T

θ; $\Delta \vdash e_0$: T_0　T_3 = 最大≤{T_1,T_2}　θ; $\Delta \vdash e_1$: T_1　θ; $\Delta \vdash e_2$: T_2

(FT-COND)　θ; $\Delta \vdash e_0$?e_1 : e_2 : T_3

$\tilde{e}$ = e_1, ..., e_n　θ&&属性变换(e_1); $\Delta \vdash e_1$: T_1　...　θ&&属性变换(e_n); $\Delta \vdash e_n$: T_n　T = T_n

$\theta \vDash \exists(e_1,\ldots,e_n)$　$\forall i \in \{1....,n\}$, 若 ($T_i \neq$ T) 则 ($\theta \vDash$ neverLast(i ~e))

(FT-SEQ)　θ; $\Delta \vdash$ (~e) : T

θ; $\Delta \vdash e_0$: T_0 / 类型(uop) = (T_0)→整数

(FT-UOP)　θ; $\Delta \vdash$ 一元运算符 (e_0) : 整数

θ; $\Delta \vdash e_1$: T_1　T_3 = 最大≤{T_1,T_2}　θ; $\Delta \vdash e_2$: T_2 / 类型(二元运算符) = (T_3,T_3)→整数

(FT-BOP)　θ; $\Delta \vdash e_1$ bop e_2 : 整数

s $\in$ dom($\overline{\text{SD}}$)　$\theta \vDash$属性变换(程序(s))

(FT-MALLOC)　θ; $\Delta \vdash$ MALLOC(结构体 s) : 结构体 s*

θ; $\Delta \vdash e_0$: 结构体 s*

(FT-MFREE)　θ; $\Delta \vdash$ MFREE(e_0) : 空类型*

图 11：CLC SPL 的基于产品族的类型检查 （Φ，程序，属性变换），其中 Φ = （$\mathcal{F}$ ϕ）；相对于 LC 类型判定（图 9）所[illegible]出显示。

E5（规则 FT-SEQ 中）如果序列 (e1, ..., e_n) 中的表达式 e_i($1 \le i \le n$) 的类型 T_i 与序列最后一个表达式的类型 T_n 不同，那么对于某个 $j \in \{i+1, ..., n\}$，表达式 e_j 会出现（因此，在任何变体中，e_i 都不能是链表中最后一个被包含的元素）。

E6 （规则中 e FT-MALLOC）如果表达式 MALLOC(struct s) 出现，则结构体 s 出现 t.

令 （Φ、程序、属性变换）为一个软件产品线。如果p是一个有效产品，那么 $[\![$程序$]\!]_p$是一个符合图 7 文法的程序（可能不是类型正确的）。

例 5.5（队列软件产品线的基于产品族的类型检查）。 例 5.3 中给出的队列软件产品线是一个类型正确的 CLC SPL。

5.3 变体生成

为了生成程序变体，我们必须为 CLC 软件产品线定义变体生成器 VG。给定产品 p，生成器 VG 会遍历程序 的语法树（即软件产品线的代码基），并移除所有由注释表 AT 标注、且其公式 ϕ 在p所确定的解释下求值为 0 的代码片段。

给定代码库 CB 和产品p，我们用 $[\![CB]\!]_p$ 表示相应的变体。变体生成器的定义见图 12，它由两个相互递归定义的函数构成：$[\![_]\!]_p$ 用于抑制链表中那些带有在p下求值为假的公式的元素，而 $_{}_p$ 遍历语法树，并将 $[\![_]\!]_p$ 应用于带注解序列。

例 5.6 （队列软件产品线的变体）。 考虑例 5.3 中给出的类型正确的队列软件产品线。将变体生成器应用于图 6 所示的带注解代码库后，可生成其六个类型正确的 LC 程序，如例 4.3 所示。

6 性质

如果某种编程语言 PL 编写的程序的软件产品线，其所有变体按照 PL 的类型系统都是类型正确的程序，那么该软件产品线就是类型安全的。本节证明 CLC 的类型安全性，即类型正确的 CLC SPL 是类型安全的。我们首先（在第 6.1 节）证明，类型正确的 CLC SPL 的代码库是一个类型正确的 LC 程序，并给出一些辅助引理；然后（在第 6.2 节）证明关于变体生成的一些基本保证以及类型安全性结论。

6.1 类型正确的 CLC SPL 的代码库是一个类型正确的 LC 程序及一些辅助引理

定理 6.1 （类型正确的 CLC SPL 的代码库是一个类型正确的 LC 程序）。 考虑一个 *CLC SPL* (Φ, 程序, 属性变换) 且满足 Φ =(F ϕ)。如果 $\phi \vdash$ 程序 OK 那么 $\vdash$ 程序 OK。

证明。首先回顾一下，忽略灰色高亮的部分后，图 9 中的 LC 类型规则可以从图 11 中基于族的类型规则得到。具体而言，需要：(i) 忽 :结构体类型判定的前提（实际上，LC 只要求满足关于结构体的合理性条件）；以及 (ii) 忽略其他规则中涉及 $\models$ 的前提（即对注解的检查）。然后，只需对证明 $\phi \vdash$程序 OK 的推导 D 使用归纳法，即可直接得证。 □

现在我们给出一些辅助引理。

$$[\![a_1 \ldots a_n]\!]_p \quad := \quad \begin{cases} \bullet & \text{若} n == 0 \\ \langle\!\langle a_1 \rangle\!\rangle [\![a_2 \ldots a_n]\!]_p & \text{如果 } p \vDash \text{属性变换}(a_1), n \geq 1 \\ [\![a_2 \ldots a_n]\!]_p & \text{如果 } p \nvDash \text{属性变换}(a_1), n \geq 1 \end{cases} \quad \text{(VGLIST)}$$

$$\begin{array}{rcll}
\langle\!\langle \text{结构体 s}\{\overline{\text{T m}};\};\rangle\!\rangle_p & := & \text{结构体 s}\{\overline{[\![\text{T m};]\!]_p}\}; & \text{(VG0)} \\
\langle\!\langle \text{T f}(\overline{\text{Tx}})\{\text{返回 e};\}\rangle\!\rangle_p & := & \text{T f}(\overline{[\![\text{Tx}]\!]_p})\{\text{return}\langle\!\langle \text{e}\rangle\!\rangle_p;\} & \text{(VG1)} \\
\hline
\langle\!\langle \text{T m}\rangle\!\rangle_p & := & \text{T m} & \text{(VG2)} \\
\langle\!\langle \text{T x}\rangle\!\rangle_p & := & \text{T x} & \text{(VG3)} \\
\hline
\langle\!\langle \text{n}\rangle\!\rangle_p & := & \text{n} & \text{(VG4)} \\
\text{空指针}_p & := & \texttt{NULL} & \text{(变体生成器5)} \\
\langle\!\langle \text{x}\rangle\!\rangle_p & := & \text{x} & \text{(变体生成器6)} \\
\langle\!\langle \text{f}(\tilde{\text{e}})\rangle\!\rangle_p & := & \text{f}([\![\tilde{\text{e}}]\!]_p); & \text{(变体生成器7)} \\
{}_{\text{《表达式->m》}p} & := & \langle\!\langle \text{e}\rangle\!\rangle_p\text{->m} & \text{(变体生成器8)} \\
\langle\!\langle \text{e->m=e}\rangle\!\rangle_p & := & \langle\!\langle \text{e}\rangle\!\rangle_p\text{->m} = \langle\!\langle \text{e}\rangle\!\rangle_p & \text{(变体生成器9)} \\
\langle\!\langle \text{e?e:e}\rangle\!\rangle_p & := & \langle\!\langle \text{e}\rangle\!\rangle_p \ ? \ \langle\!\langle \text{e}\rangle\!\rangle_p \ : \ \langle\!\langle \text{e}\rangle\!\rangle_p & \text{(VG10)} \\
\langle\!\langle (\tilde{\text{e}})\rangle\!\rangle_p & := & ([\![\tilde{\text{e}}]\!]_p) & \text{(VG11)} \\
{}_{\text{《一元运算符 e》}p} & := & {}_{\text{一元运算符《e》}p} & \text{(VG12)} \\
{}_{\text{《e 二元运算符e》}p} & := & \langle\!\langle \text{e}\rangle\!\rangle_p \, {}_{\text{二元运算符《e》}p} & \text{(VG13)} \\
\langle\!\langle \texttt{MALLOC}(\text{结构体 s})\rangle\!\rangle_p & := & \texttt{MALLOC}(\text{结构体 s}) & \text{(VG14)} \\
\langle\!\langle \texttt{MFREE}(\text{e})\rangle\!\rangle_p & := & \texttt{MFREE}(\langle\!\langle \text{e}\rangle\!\rangle_p) & \text{(VG15)}
\end{array}$$

图 12：应用于产品 p 的 CLC SPL 变体生成器 $\overline{(\Phi, \text{SDFD}}$，属性变换）。

设 $\mathcal{D}$ 为证明 $\phi \vdash$程序 OK 的推导。值得指出的是，在定理 6.1 的证明中所构造的、用于证明 $\vdash$程序 OK 的推导 D′，大致而言，是通过从 D 中移除其规则里所有以灰色标识（依据图 11）的部分，然后将每条规则 FT-<NAME> 替换为规则 T-<NAME> 而得到的。

引理 6.2（关于基于产品族的类型检查中的子类型关系与注解）。考虑一个 *CLC SPL*（Φ，程序，属性变换）以及两个类型 T_0,T_1 出现在 程序 中。若 $\theta \vDash$ 属性变换（T_1） 且 $T_0 \leq T_1$，则 $\theta \vDash$ 属性变换（T_0）。

证明。首先注意到，若 $T_0 \leq T_1$，则可能的情形如下：（i）若 T_0 = 整数，则 T_1 = 整数；（ii）若对某个结构体 s 有 T_0 = 结构体 s∗，则 T_1 = 结构体 s∗；以及（iii）若 T_0 = 空类型∗，则要么 T_1 =空类型∗，要么对某个结构体有 T_1 = 结构体 s∗。在所有情形中，依据记法 5.4，证明都是直接的。 □

设 J 和 J' 是推导 $\mathcal{D}$ 中 $\phi \vdash$程序 OK 的两个判断出现。若语句J' 出现在J 的（子）推导中，则称其为J 的后代（J'是J' 的祖先）。

引理 6.3（关于基于产品族的类型判断左侧的公式）。考虑一个 *CLC SPL*(Φ, 程序, 属性变换)，其中包含 $\Phi = (\mathcal{F}\ \phi)$。设 J 和 J' 是 $\phi \vdash$ 程序 OK 的推导中的两个判断出现，且 J' 是 J 的后代。若 θ_J 是判断 J 的最左公式，而 θ'_J 是 J' 的最左公式，则 $\theta'_J \vDash \theta_J$。

证明。通过对 $\phi \vdash$程序 OK 的推导进行归纳即可得证。显然，前提中的每个最左公式 θ' 都是：(i) 规则结论中判断的相同最左公式 θ；或 (ii) θ 与另一公式的合取。因此，若 θ_J 是某个判断 J 的最左公式，且J' 是其后代（最左公式标注为 θ'_J），则 $\theta'_J \vDash \theta_J$。 □

引理 6.4 (关于表达式的基于产品族的类型检查)。考虑一个 *CLC SPL* (Φ，程序，属性变换)，其中 $\Phi = (\mathcal{F}\ \phi)$。若 θ；$\Delta \vdash e : T$ 包含在证明 $\mathcal{D}^{Prg}$ 该判断式 $\phi \vdash$ 程序 良构 的推导中，则 $\theta \vDash$属性变换(T)。

证明。若 $\mathcal{D}$是证明 θ；$\Delta \vdash$ e :T的子推导，则根据 $\mathcal{D}$ 中最后使用的规则进行归纳，即可得证。注意，$\mathcal{D}$ 是通过唯一一条 FT-$\mathcal{D}^{Prg}$ 中包含的子推导，该规则紧接在唯一一次 FT-函数 应用之上使用，即 FT-PRG；判断式 θ；$\Delta \vdash$ e :T涉及函数 $f_{\mathcal{D}}$ 函数体的一部分。我们将这条唯一的父规则应用称为 D 的 FT-FUN 父规则应用。

- FT-INT、FT-空指针、FT-BOP、FT-UOP 和 FT-MFREE 的情形显然成立，因为我们假设 属性变换(整数) = 属性变换 (空类型 ∗) = 1。
- FT-PAR 情形成立，因为它验证了 $\theta \vDash \psi$，更准确地说，是某个 Tx 的 $\theta \vDash$ AT(Tx)。事实上，ψ是 x :T 的带标注公式，其中 AT(Tx) 由 D 的 FT-FUN 父规则应用加入类型环境。该父规则的前提中包含 $\theta' \vDash$AT(Tx) ⇒AT(T)，其中 $\theta' = \phi$&&AT(Prg(f))，因为 FT-FUN 紧接在规则 FT-PRG 之上。由于 θ 是在标注一个后代判断式，根据引理 6.3 可得 $\theta \vDash \theta'$。因此，$\theta \vDash$ AT(Tx) ⇒AT(T)，我们由此得出 $\theta \vDash$AT(T)。
- 在 FT-APP 的情形中，表达式 e 必须具有 f(e) 的形式，且其前提为 $\theta \vDash$AT(Prg(f))，这确保了只要 θ成立，就存在 Prg(f) =T0 f (T x){. ..}（其中 T0 是 f(e) 的类型）。注意，D 中的 FT-FUN 父规则应用用于对函数 fD 进行类型判定，而 fD 可能与 f 不同。尽管如此，对于 Prg 中定义的每个函数，推导 D Prg 都包含一个根规则为 FT-FUN 的子推导。特别地，其中存在一个针对 f 的子推导，其前提中包括：$\theta' \vDash$AT(T0)，其中 $\theta' = \phi$&&AT(Prg(f))，因为 FT-FUN 紧接在规则 FT-PRG 之上。显然 $\theta \vDash \theta'$，因此可得 $\theta \vDash$ AT(T0)。

- FT-MEMBER 和 FT-ASSIGN 的情形类似。它们的前提是 $\theta \vDash$AT(T m)，涉及结构体 s 的一个成员。尽管如此，FT-PROG 纳入 D Prg 的健全性假设确保必定存在一个从规则 FT-STRUCT 开始的子推导，用于验证结构体 s 的类型判定。后者的前提为 $\theta' \vDash$AT(T m) $\Rightarrow$AT(T)，其中 $\theta' = \phi$&&AT(Prg(struct s))。因此，$\theta \vDash \theta'$，证明便可轻松得出。
- 其余情况由归纳法得出。

□

6.2 CLC变体生成和类型安全的基本保证

下列定理保证，SPL 的类型判定确保所有有效变体的一致性。

到目前为止，我们将 程序 用作映射，因此，dom(程序) 表示其定义域。我们将该记号扩展为包含基类型；具体来说，我们用dom^{int}(程序) 表示 dom(程序) $\cup$ {整数 ,空类型$_*$}。

定理 *6.5*（带注释结构体的安全性）。设 *(Φ,Prg, AT)* 是一个 *SPL*，且满足 $\Phi = (F \quad \phi)$，其中 p 是一个有效产品，并且 $struct\ s\{T_1 m_1 \ldots T_n m_n\}$ 属于 *Prg*。则有：*(i)* $p \vDash AT(\ struct\ s)$ 当且仅当 $struct\ s \in dom([\![Prg]\!]_p)$；*(ii)* 若 $p \vDash AT(\ struct\ s)$，则：$p \vDash AT(T_i m_i)$ 当且仅当 $T_i m_i$ 包含于《$struct\ s\{\overline{T\ m}\}$》$_p$；*(iii)* 若 $\phi \vdash Prg\ OK$，$p \vDash AT(\ struct\ s)$ 且 $p \vDash AT(T_i m_i)$，则对所有 i，$T_i \in dom^{int}([\![Prg]\!]_p)$。

证明。第 (i) 点成立，因为按照变体生成规则 VGLIST，$[\![_]\!]_p$ 应用于结构体定义链表，并且我们知道 p $\vDash$ AT(结构体 s)。同理，$T_i m_i$ 不会从 结构体 s{T m}$_p$ 中移除，因为规则 VG2 $[\![_]\!]_p$ 由 p $\vDash$ AT($T_i m_i$)驱动：因此，(ii) 得证。第 (iii) 点成立，因为在 $\phi \vdash$Prg OK 的推导中，结论必然由规则 FT-PRG 得出。其前提中有 ϕ&& AT(SD) $\vdash$ SD OK，其中包含 ϕ&& AT(结构体 s) $\vdash$ 结构体 s{T m} OK 这一实例。反过来，这个前提由规则 FT-STRUCT 证明，其前提为 ϕ && AT(结构体 s) $\vDash$AT(T m) $\Rightarrow$ AT(T)。因此，该证明由关于 p 的假设得出。 □

我们用 $e_0 \Subset e_1$ 表示位置 e_0 包含在 e_1 中。设 e_f 为函数 f 的函数体。由有效产品 p 生成的变体中位置 e′ 是否存在，取决于许多语法元素的注解：(i) 程序(f) 的注解；(ii) 属性变换(e′) 的注解；以及 (iii) e′ 所嵌套于其中的表达式列表的注解。更精确地说，我们将 nested_f(e′) 记为如下可注解位置集合 {(˜e) $\Subset$ e f | e′ $\Subset$ (˜e)} $\cup$ {g(˜e) $\Subset$ e f | e′ $\Subset$ g(˜e)}。

引理 *6.6*（函数生成）。设 *(Φ, 程序, 属性变换)* 为一个软件产品线，使得 p 是一个有效产品，且 $Tf\ (T_1 x_1 \ldots T_n x_n)\ \{\ return\ e;\}$ 属于程序。则有：*(i)* $p \vDash$属性变换*(f)* 当且仅当 $f \in dom([\![程序]\!]_p)$；*(ii)* 若 $p \vDash$属性变换*(f)*，则：$p \vDash$属性变换$(T_i x_i)$ 当且仅当 $T_i x_i$ 被包含（未被移除）于《$Tf\ (T_1 x_1 \ldots T_n x_n)\{\ return\ e;\}$》$_p$；并且，*(iii)* 若 $p \vDash$属性变换*(f)* 且 $nested\ f\ (e') = \{e1, \ldots, e_n\}$，则：$p \vDash$(属性变换$(e_1)$&& $\ldots$ &&属性变换(e_n)) 当且仅当 e' 被包含于《$Tf\ (T_1 x_1 \ldots T_n x_n)\{\ return\ e;\}$》$_p$。

证明。 第 (i) 点和第 (ii) 点与定理 6.5 类似。第 (iii) 点通过对变体生成器的（相互递归）定义进行标准（相互）归纳法得出。 □

设 Δ 为 $\overline{\text{x :T}}^{\text{with}}$ 属性变换(T x)，则我们将 $[\![\Delta]\!]_p$ = {x:T | x:T 带有 属性变换(Tx) $\in \Delta$, p $\vDash$ 属性变换(Tx)} 记为：产品 p 中所使用的序对 x:T 可作为所考虑函数的参数。

定理 *6.7*（带注解表达式的安全性）。设 *(Φ,Prg, AT)* 是一个软件产品线（*SPL*），其中 p 是有效产品，且 D^{Prg} 是证明 $\phi \vdash Prg\ OK$ 的推导。若 $p \vDash \theta$ 且 $\theta; \Delta \vdash e : T$ 出现在 D^{Prg} 中，则 $[\![\Delta]\!]_p \vdash$《e》$_p : T$。此外，若推导中使用了结构体 s，并由此得出 $[\![\Delta]\!]_p \vdash$《e》$_p : T$，则 $p \vDash AT(s)$。

证明。若$\mathcal{D}$是证明 θ； $\Delta \vdash$ e :T那么，只需对$\mathcal{D}$中最后使用的规则进行归纳即可得证。（事实上，$[\![\Delta]\!]_p \vdash \langle\!\langle e \rangle\!\rangle_p \vdash$:T的证明可以由 θ； $\Delta \vdash$ e :T的证明得到：删除有关从变体生成器中移除的子表达式的子推导，然后擦除剩余推导中的注解即可。）

• FT-INT、FT-NULL、FT-MALLOC 这几种情况，分别可通过使用规则 T-INT、T-NULL、T-MALLOC，在变体$[\![$程序$]\!]_p$的类型推导中进行转换。

• 如果最后一条规则是 FT-PAR，那么它的前提是 x :T，且$\psi \in \Delta$和 $\theta \vDash \psi$。容易看出， Δ在FUN 规则下的（唯一）祖先中，$\mathcal{D}$ 中的$\mathcal{D}_{程序}$已预先确定，因此没有任何针对表达式的类型判断会修改它。规则 FT-FUN 在 Δ中为f 的每个形式参数加入一对变量—类型及其对应的注解。因此，证明成立，因为$p \vDash \theta$。

• 考虑 FT-APP 的情况，并令f 为被调用函数。前提 $\theta \vDash$AT(程序(f)) 和引理 6.6.i 确保f 的定义包含在变体 $[\![$程序$]\!]_p$中，满足 T-APP的前提要求。引理 6.1 确保，在软件产品线代码库程序中，f 的形式参数数量始终与调用中提供的实参数量相同，因此，前提 $\theta \vDash$AT(e) $\Leftrightarrow$ AT(Tx) 应用于长度相同的链表。后一条件与引理 6.6.ii 确保，在变体 $e\rangle\!\rangle_p$中，f 未被移除的形式参数数量与提供给它的未被移除的参数数量完全匹配。T-APP 隐含地要求这种匹配。因此，我们可以通过归纳法轻松得出结论。

• FT-MEMBER 情形可通过归纳法得出，只需注意到 $\theta \vDash$属性变换(T m) 可确保 T m 不会从《e_p 中移除，正如 T-MEMBER 所要求的（在 $[\![$程序$]\!]_p$ 的类型推导中）。

• FT-ASSIGN 情形与前一个情形类似，只需在 $[\![$程序$]\!]_p$ 的推导中用 T-ASSIGN 规则替换它。

• FT-COND、FT-UOP、FT-BOP 和 FT-MFREE 情形可通过归纳法得出。

• FT-SEQ 情形。软件产品线的文法确保每个表达式列表至少包含一个表达式，而前提 $\theta \vDash \exists$(e1, . . . , en) 确保在任何变体中都不会将列表中的所有表达式移除。此外，前提 $\forall$i $\in$ {1...., n}，若 (Ti $\neq$ T)，则 ($\theta \vDash$ neverLast(i ˜e)) 确保在某个变体中结束列表表达式的表达式，其类型始终与软件产品线代码基中最后一个表达式的类型相同。每当 p $\vDash \theta$&&属性变换(ei) 成立时，根据前提θ&&属性变换(ei);$\Delta \vdash$ ei :Ti 的归纳假设，我们有 $[\![\Delta]\!]$p $\vdash$ eip :Ti。因此得证。

□

现在我们可以证明类型安全性结论。

定理6.8 (CLC 的类型安全性)。 设(Φ,程序, 属性变换) 是一个软件产品线，且 $\Phi = (\mathcal{F}\ \phi)$ 满足 $p \vDash \phi$，即 p 是一个有效产品。如果 $\phi \vdash$程序 OK，那么 $\vdash [\![$程序$]\!]_p$ OK，即由 p 标识的变体是类型正确的。

证明。 设 D^{Prg} 是证明 $\phi \vdash$ 程序 OK 的推导。该推导以规则 FT-PRG 结束。我们需要验证能否构造出以规则 T-PRG 结束的变体 $[\![$程序$]\!]_p$ 的推导。

• 前提 程序SANE 并不能立即推出其健全性，对象为$\langle\!\langle$程序$\rangle\!\rangle_p$，因为 first 涉及整个软件产品线的代码基，而后者涉及一个已移除某些代码片段的特定变体。健全性需要 3 个条件。first 要求对于每个结构体名 s在各处出现于 程序 我们有 s $\in$dom(SD)。这由引理 6.2、6.5.iii 和 6.7 得出。所出现函数的安全性可立即得出，因为唯一考虑某个

已定义函数 f（在其定义之外）是 FT-APP，其前提中包含 $\theta \vDash$属性变换(程序(f))：它确保若该应用未被从某个变体（由 p 标识）中移除，则 f 的定义也会被包含。f 的存在由 FT-PRG 的前提保证。

• T-PRG 的其余前提涉及 ⟦Prg⟧p 中的函数定义。VG 的定义确保，如果函数 f 的定义被包含在变体中，那么 p ⊨ AT(Prg(f))，因此对于 ⟦Prg⟧p 中包含的每个函数，都有 p ⊨ ϕ && AT(Prg(f))。对于 SPL 中具有定义的每个函数 Prg(f) =Tf (T1x1 . . .Tnxn){ return e;}，规则 FT-PRG 都包含一个前提 ϕ && AT(Prg(f)) ⊢Prg(f) OK。反过来，该前提是规则 FT-FUN 的结论；该规则包含足够的前提，可用于对所包含的函数应用规则 T-FUN。具体而言，引理 6.7 确保我们能够为 f 函数体中表达式未被移除的部分构建类型推导。

□

7 结论与未来工作

本文介绍了：（1）轻量级C 的语法和类型判定，它是用于 C 程序的核心演算；（2）CLC 的语法和生成映射，它是用于（L）C 软件产品线的核心演算；以及（3）CLC 的类型系统，该系统对基于产品族的类型检查进行了形式化，并保证良类型 CLC SPL 的所有变体都是良类型的（L）C 程序。

未来的工作中，我们计划为轻量级C 提供小步操作语义，证明其类型健全性，并对 C 程序的其他分析以及 C 程序软件产品线的基于产品族的分析（例如堆安全性）进行形式化。

此外，我们希望将 CLC 扩展到更大的 C 语言子集，并为其配备一套用于分析 C 程序和 C 程序软件产品线的工具链。众所周知，分析 C 代码极其困难。因此，我们不打算涵盖完整的 C 语法，也不打算提供用于分析现有 C 程序软件产品线的工具链，而是致力于为以下方面提供支持：

• 开发以 C 子集编写的程序的新软件产品线；以及
• 将现有的 C 程序软件产品线重构为以工具链所支持的 C 子集编写的软件产品线。

参考文献

[1] ANSI（2024）：当前 *C* 编程语言标准——*ISO/IEC 9899:2024*（*C24*）。网址：https://blog.ansi.org/c-programming-language-standard-iso-iec-9899-2024/。最后访问日期：2024-02-03。[2] Sven Apel、Don S. Batory、Christian Kästner 和 Gunter Saake（2013）：面向特征的软件产品线：概念与实现。Springer，doi:10.1007/978-3-642-37521-7。[3] Hendrik Pieter Barendregt（1985）：λ 演算——其语法与语义。逻辑与数学基础研究103，North-Holland。[4] Don Batory（2005）：特征模型、文法与命题公式。载于：国际软件产品线会议（*SPLC*）论文集，*LNCS* 3714，Springer，第 7–20 页，doi:10.1007/11554844_3。[5] Sandrine Blazy 和 Xavier Leroy（2009）：*C* 语言 *Clight* 子集的机械化语义。*J.Autom. Reason.*43(3)，第 263–288 页，doi:10.1007/S10817-009-9148-3。

[6] Dennis M. Ritchie、Brian W. Kernighan（1988）：《ANSI C 程序设计语言》，第 2 版。Prentice Hall。[7] Luca Cardelli（1997）：程序片段、链接与模块化。载于：第 24 届 ACM SIGPLAN-SIGACT 程序设计语言原理研讨会论文集，POPL '97，计算机协会，纽约，美国，页码 266–277，doi:10.1145/263699.263735。[8] Alonzo Church（1932）：《逻辑基础的一组公设》。《数学年刊》33(2)，第 346–366 页。网址：http://www.jstor.org/stable/1968337。[9] P. Clements 和 L. Northrop（2001）：软件产品线：实践与模式。Addison Wesley Longman。[10] Chucky Ellison（2012）：C 的形式语义及其应用。博士论文，伊利诺伊大学厄巴纳-香槟分校，美国。网址：https://hdl.handle.net/2142/34297。[11] Chucky Ellison 和 Grigore Rosu（2012）：C 的可执行形式语义及其应用。载于：第 39 届 ACM SIGPLAN-SIGACT 程序设计语言原理研讨会论文集，POPL '12，计算机协会，纽约，美国，页码 533–544，doi:10.1145/2103656.2103719。[12] Gentoo 基金会（2019）：Gentoo Linux。网址：https://gentoo.org。最后访问日期：2019-08-20。[13] Chris Hathhorn、Chucky Ellison 和 Grigore Ro̧su（2015）：定义 C 的未定义性。载于：第 36 届 ACM SIGPLAN 编程语言设计与实现会议论文集，PLDI '15，计算机协会，纽约，美国，页码 336–345，doi:10.1145/2737924.2737979。[14] Atsushi Igarashi、Benjamin C. Pierce 和 Philip Wadler（2001）：Featherweight Java：Java 和 GJ 的最小核心演算。ACM Trans. Program. Lang. Syst. 23(3)，页码 396–450，doi:10.1145/503502.503505。[15] Jeehoon Kang、Chung-Kil Hur、William Mansky、Dmitri Garbuzov、Steve Zdancewic 和 Viktor Vafeiadis（2015）：支持整数与指针类型转换的 C 形式化内存模型。载于：第 36 届 ACM SIGPLAN 编程语言设计与实现会议论文集，PLDI '15，计算机协会，纽约，美国，页码 326–335，doi:10.1145/2737924.2738005。[16] Christian Kästner、Sven Apel、Thomas Thüm 和 Gunter Saake（2012）：基于注解的软件产品线类型检查。ACM Trans. Softw. Eng. Methodol. 21(3)，doi:10.1145/2211616.2211617。[17] Christian Kästner、Klaus Ostermann 和 Sebastian Erdweg（2012）：一种感知变异性的模块系统。载于：第 27 届 ACM SIGPLAN 面向对象程序设计、系统、语言与应用年会（OOPSLA）论文集，ACM 出版社，纽约，美国，页码 773–792，doi:10.1145/2384616.2384673。[18] Andy Kenner、Christian Kästner、Steffen Haase 和 Thomas Leich（2010）：TypeChef：面向 C 语言中 #ifdef 变异性的类型检查。载于：第 2 届面向特征的软件开发国际研讨会论文集，FOSD '10，计算机协会，纽约，美国，页码 25–32，doi:10.1145/1868688.1868693。[19] Jörg Liebig、Sven Apel、Christian Lengauer、Christian Kästner 和 Michael Schulze（2010）：对 40 个基于预处理器的软件产品线中的变异性进行分析。载于：第 32 届 ACM/IEEE 国际软件工程会议论文集——第 1 卷，ICSE '10，计算机协会，纽约，美国，页码 105–114，doi:10.1145/1806799.1806819。[20] Jörg Liebig、Alexander von Rhein、Christian Kästner、Sven Apel、Jens Dörre 和 Christian Lengauer（2013）：《可扩展的可变软件分析》。载于：欧洲软件工程会议暨 ACM SIGSOFT 软件工程基础研讨会（ESEC/FSE）论文集，ACM 出版社，纽约，美国，页码 81–91，doi:10.1145/2491411.2491437。[21] Michael Lienhardt、Ferruccio Damiani、Einar Broch Johnsen 和 Jacopo Mauro（2020）：超大配置空间中的惰性产品发现。载于：ACM/IEEE 第 42 届国际软件工程会议论文集，ICSE '20，计算机协会，纽约，美国，页码 1509–1521，doi:10.1145/3377811.3380372。[22] Flávio Medeiros、Márcio Ribeiro 和 Rohit Gheyi（2013）：调查基于预处理器的语法错误，第 75–84 页。doi:10.1145/2517208.2517221。

[23] Benjamin C. Pierce（2002）：类型与程序设计语言，第1版。麻省理工学院出版社。[24] G.D. Plotkin（1977）：将 *LCF* 视为一种程序设计语言。理论计算机科学 5(3)，第223–255页，doi:10.1016/0304-3975(77)90044-5。[25] K. Pohl、G. Böckle 和 F. van der Linden（2005）：软件产品线工程——基础、原则与技术。Springer。[26] Alexander von Rhein、Jörg Liebig、Andreas Janker、Christian Kästner 和 Sven Apel（2018）：大规模变异性感知静态分析：一项实证研究。*ACM* 软件工程与方法学汇刊27(4)，第18号文章，doi:10.1145/3280986。[27] Ina Schaefer、Rick Rabiser、Dave Clarke、Lorenzo Bettini、David Benavides、Goetz Botterweck、Animesh Pathak、Salvador Trujillo 和 Karina Villela（2012）：软件多样性：现状与展望。《软件工具与技术转移国际期刊》14(5)，第477–495页，doi:10.1007/s10009- 012-0253-y。[28] Jeremy Siek（2012）：一种熟悉语言的语义：*Featherweight C*。可访问 http://siek.blogspot.com/2012/07/the-semantics-of-familiar-language.html。最后访问日期：2024-02-03。[29] SourceForge（1999）：开源 *Linux* 软件开发软件。可访问 http://freshmeat. net/。 [在线；访问日期：12月05日-2025]。[30] Thomas Thüm、Sven Apel、Christian Kästner、Ina Schaefer 和 Gunter Saake（2014）：软件产品线分析策略的分类与综述。*ACM* 计算调查 47(1)，第6:1–6:45页，doi:10.1145/2580950。