Advances in Applied Mathematics 应用数学进展, 2024, 13(8), 3666-3676
Published Online August 2024 in Hans. https://www.hanspub.org/journal/aam
https://doi.org/10.12677/aam.2024.138349

Hans汉斯

基于指针操作的乘法器验证程序优化

王晨瑞，江建国

辽宁师范大学数学学院，辽宁 大连

收稿日期：2024年7月5日；录用日期：2024年7月29日；发布日期：2024年8月6日

摘　要

乘法器电路验证是算术电路验证领域内的一个重大难题。Gröbner基方法是其中目前最为有效的验证方法之一。基于此方法开发的Amulet程序通过减少中间变量数量提高了验证效率，但是对于大型乘法器，验证速度慢的问题仍存在。本文对Amulet的关键算法进行了进一步优化，通过指针操作对函数进行重写，缩短了验证的时间，并根据实验数据体现了其在大型乘法器验证中的应用优势，为形式化验证技术的未来研究提供了参考。

关键词

乘法器验证，Gröbner基方法，C语言指针

Optimization of Multiplication Circuit Verification Program Based on Pointer Operations

Chenrui Wang, Jianguo Jiang

School of Mathematics, Liaoning Normal University, Dalian Liaoning

Received: Jul. 5th, 2024; accepted: Jul. 29th, 2024; published: Aug. 6th, 2024

Abstract

The verification of multiplier circuits is a significant challenge in the field of arithmetic circuit verification. The Gröbner basis method is currently one of the most effective verification methods available. The Amulet program, developed based on this method, improves verification efficiency by reducing the number of intermediate variables. However, for large multipliers, the verification

文章引用: 王晨瑞, 江建国. 基于指针操作的乘法器验证程序优化[J]. 应用数学进展, 2024, 13(8): 3666-3676.
DOI: 10.12677/aam.2024.138349

speed remains an issue. This paper further optimizes the key algorithms of Amulet, by rewriting functions through pointer operations, reduces verification time. Experimental results demonstrate its advantages in the verification of large multipliers. It provides a reference for future research in formal verification techniques.

Keywords

Verification of Multipliers, Gröbner Basis Method, C Pointers

1. 引言

数字电路执行逻辑运算，因此成为计算机和数字系统的核心部分。特别是执行特定算术运算的算术电路，如加法器和乘法器电路，它们是处理器的关键组件。因此，保证这些电路的正确性至关重要，以避免类似于 1994 年奔腾 FDIV 错误的问题再次出现[1]。尤其是乘法器电路的正确性验证，至今仍充满挑战。

迄今为止，验证乘法器的方法多种多样。最初发现奔腾错误的方法依赖于二叉判定图[2]，这需要对乘法器的内部结构有所了解[3] [4]。另一种常见的方法是将问题建模为可满足性(SAT)问题，并将电路转换为合取范式(CNF)。此类方法在 2016 年的 SAT 竞赛中得到了大量应用[5]。然而，验证乘法器的 CNF 公式需要指数级的消解证明[6]，导致 CDCLSAT 求解器的运行时间呈指数增长。在此基础上，结合定理证明器如 ACL2 [7]的方法也被用于证明工业级乘法器的正确性[8]，但这通常不是完全自动化的，并且需要专业知识。

目前，对于平面化乘法器的自动验证中最有效的方法是应用计算机代数，将电路的每一个门及其规范用多项式表示，并按其拓扑结构排序成 Gröbner 基[9]。通过对规范使用电路的 Gröbner 基进行约简，可以验证乘法器的正确性；如果约简结果为零，则乘法器被认为是正确的。然而，传统的代数方法面临着约简的中间变量数量急剧膨胀的问题，为解决这一问题，近年来开发了多种预处理技术和约化方法[10]-[12]。

Kaufmann 团队针对 Gröbner 基方法进行了创新性改进，并开发了一款基于 C 语言的程序——Amulet [13]。该程序在 Linux 中运行，通过对乘法器的重写、替换和切片操作，有效地减少了在约简过程中生成的中间变量数量，从而提高了形式化验证的效率。尽管 Amulet 已显示出显著的性能提升，但程序本身的优化潜力仍然存在。

本文对 Amulet 的关键算法进行了深入改进，通过指针操作对其进行优化，进一步提升了验证效率，扩大了其在大型乘法器验证中的应用优势。本研究的成果不仅加深了我们对形式化验证技术的理解，也为未来在该领域的研究提供了参考。

2. Gröbner 基法

2.1. Gröbner 基理论

本节内容仅介绍 Gröbner 基法应用于电路验证中的预备知识，想要了解更加系统的内容请查阅[14]。

假设 k 是一个数域，$k[x_1,\cdots,x_n]$ 是数域 k 上以 $x_1,\cdots,x_n$ 为变量的 n 元多项式环。在其中的单项式可以定义一定的序关系，得益于此每个多项式能够定义首项，记为 lm。同时根据首项，对多项式进行排序，以保证除法运算中余式的唯一性。

定义 2.1

设 I 是 $k[x_1,\cdots,x_n]$ 的一个非空子集，且：

1) $0\in I$；

2) 对任意的多项式 $f,g\in I$，都有 $af+bg\in I$，其中 $a,b\in k[x_1,\cdots,x_n]$；

则称 I 是一个多项式理想。

定义 2.2

设多项式 $f_1,\cdots f_s\in k[x_1,\cdots,x_n]$，若理想 $I=\langle f_1,\cdots,f_s\rangle=\left\{\sum_{i=1}^{s}h_i f_i \middle| h_i\in k[x_1,\cdots,x_n]\right\}$，则称 $\{f_1,\cdots,f_s\}$ 是 I 的一组基。

一个理想可以有很多组基。在定义一定序关系后，我们可以根据基中元素首项之间的关系找出一种特殊的理想基。

定义 2.3

对于数域 $\mathbb{R}$ 上的多项式环，理想 $I(C)$ 的一个基 P 若满足 $\forall q\in I,\exists p\in P:\ lm(p)\mid lm(q)$，则称 P 为 I 的一个 D-Gröbner 基。

利用 D-Gröbner 基的某种良好性质，我们可以解决理想成员问题，即给定 $y\in k[x_1,\cdots,x_n]$，$I=\langle f_1,\cdots,f_s\rangle$，判断 y 是否属于 I。具体方法将在稍后进行介绍。

定义 2.4

对于每个布尔值变量 $x\in\{x_1,\cdots,x_n\}$，$x(1-x)=0$ 表示其布尔值约束；对于布尔值变量集 $Y\subseteq\{x_1,\cdots,x_n\}$，记 $B(Y)=\{y(y-1)=0\mid y\in Y\}$ 为 Y 的布尔值约束集，其中 $B(Y)\subseteq\mathbb{R}[x_1,\cdots,x_n]$。

定义 2.5

$G(C)\subseteq\mathbb{R}[x_1,\cdots,x_n]$ 表示电路 C 中所有门多项式集。设 $P\subseteq\mathbb{R}[x_1,\cdots,x_n]$。如果对于某个规定的序关系，$P$ 的所有首项仅由指数为 1 的单个变量组成并且是唯一的，且对于所有 $p\in P$ 满足 $lc(p)\in\mathbb{R}^{\times}$，那么我们说 P 具有唯一的单项首项，记作 UMLT。令 $X_0(P)\subseteq X$ 为 P 中不为首项的所有变量的集合，并记为

$$B_0(P)=B(X_0(P))。$$

定理 2.1

如果 C 是无环电路，$l\in\mathbb{N}$。那么集合 $G(C)\cup B_0(C)\cup\{l\}$ 是理想 $I(C)+\langle l\rangle\subseteq\mathbb{Z}[x_1,\cdots,x_n]$ 的一个 D-Gröbner 基。

定理 2.2 (约简算法)

给定 p,q,r 属于 $\mathbb{R}[x_1,\cdots,x_n]$。如果存在 q 中的单项式 m'，满足 $m'=m\cdot lm(p)$ 并且 $r=q-mp$，那么我们称 q 关于 p 被约简为 r [15]。使用此算法对 D-Gröbner 基进行运算可以解决理想成员问题：

定理 2.3

设 G 是理想 I 的一组 D-Gröbner 基，多项式 $q\in\mathbb{R}[x_1,\cdots,x_n]$；$q$ 被 G 约简的余式 r 满足 $q-r\in G$；$q\in I$ 当且仅当 $r=0$

2.2. 电路代数模型

先前的 Gröbner 基法想要应用于电路，需要对电路建立代数模型，将电路中的逻辑门结构抽象成为

门多项式。一个输入、输出位数均为 $2n$ 的无环电路，设其输入变量为 $a_1,\cdots,a_n,b_1,\cdots,b_n$，输出变量为 $s_0,\cdots,s_{2n-1}$，并设其内部逻辑门变量为 $g_0,\cdots,g_{i-1}$ [16]，则可以定义多项式环 $\mathbb{R}\left[a_0,\cdots,a_{n-1},b_0,\cdots,b_{n-1},g_1,\cdots,g_k,s_0,\cdots,s_{2n-1}\right]$，定义布尔值变量 $a_0,\cdots,a_{n-1},b_0,\cdots,b_{n-1},g_1,\cdots,g_k,s_0,\cdots,s_{2n-1}$

并根据逆拓扑序定义字典序关系 $\succ$：

$$s_{2n-1}\succ\cdots\succ s_0\succ b_n\succ\cdots\succ b_0\succ a_n\succ\cdots a_0$$

因此电路的逻辑门中变量的关系可以通过多项式形式表达：

$$\begin{aligned} u=\neg v &\Rightarrow 0=-u+1-v\\ u=v\wedge w &\Rightarrow 0=-u+vw\\ u=v\vee w &\Rightarrow 0=-u+v+w-vw\\ u=v\oplus w &\Rightarrow 0=-u+v+w-2vw \end{aligned}$$

定义 2.6

设 C 是一个电路，$p\in\mathbb{R}\left[x_1,\cdots,x_n\right]$ 是多项式。将 C 中所有输入变量、逻辑门变量和输出变量的任一组赋值代入到 p 中，若结果为 0，则称 p 是电路 C 的一个多项式电路约束，记为 PCC，电路 C 是无环电路，那么 C 的 PCC 实际上就是 $G(C)\cup B_0(C)$，根据定理 2.1 的内容，电路 C 中所有 PCC 构成的集合就构成电路的 D-Gröbner 基，记为 $I(C)$。

定义 2.7

设 C 是一个电路，如果多项式

$$\sum_{i=0}^{2n-1}2^i s_i-\left(\sum_{i=0}^{n-1}2^i a_i\right)\left(\sum_{i=0}^{n-1}2^i b_i\right)$$

是一个 PCC，则称电路 C 是一个乘法器电路，称该多项式为 C 的规范多项式 L。

定理 2.4

无环电路 C 满足规范 L 当且仅当 $L\in I(C)$。

总结上面的定义定理，我们可以得到如下结论：

验证乘法器电路 C，只需使用定理 2.3 来验证 L 是否属于理想 $I(C)$。当且仅当 L 属于 $I(C)$，乘法器电路 C 正确。为此我们利用定理 2.3 对电路规范进行约简，查看最终结果 r 的取值，进而对电路正确性做出判断。

3. 电路验证程序与优化

3.1. 电路验证程序 Amulet

在最近的研究中，Linz 大学的 Kaufman 团队研发的 *Amulet* 验证程序为电路验证领域做出了重大贡献[13]。*Amulet* 是一个先进的验证工具，它利用 C 语言编写，并在 Linux 环境下运行。该工具的开发集成了乘法器的识别、重写、切片、约简和验证等多个关键步骤，提供了一个全面的解决方案来处理电路验证难题。同时，*Amulet* 还能够根据用户的需求，生成相应的证书证明，以支撑验证结果的透明度和可靠性。

本研究旨在基于 *Amulet* 验证器的现有架构和功能进行进一步的优化，目标是加速大型乘法器的验证过程，从而显著提高整体的验证效率。我的研究初步分析了使用 *Amulet* 进行电路验证时所产生的日志文件，发现在整个验证过程中，多项式的约简操作占据了大部分的计算时间。这一发现指出了一个有待提

升的性能瓶颈，即多项式运算函数的效率直接影响到验证器的运行速度。

针对上述问题，本文的核心工作集中于探讨和改进 *Amulet* 中的多项式运算函数。通过对这些核心函数进行优化，减少多项式约简步骤所需的时间，从而在不牺牲验证准确性的前提下，提升整个验证流程的效率。

验证器运行过程中，电路以 *aig* (And-InverterGraph)格式输入[17]，并在验证器中被识别。

电路中的变量均以结构体 *Var* 形式储存变量名称，变量序，所属链表等形式的信息。电路中通过访问 *Var* 指针对变量进行访问赋值。

电路中门多项式的项利用结构体指针 *Term**表示，其中主要包含：变量的结构体指针 *Var*variable*，储存对应变量的地址；指向相连接的下一个链表的指针 *Term* rest*，利用链表的特性，变量联系起来作为项的整体(见图 1)。

单项式 *Monomial**，多项式 *Polynomial**的数据结构与项的结构大概一致，都使用了单向链表相互进行联系，方便进行访问和赋值。

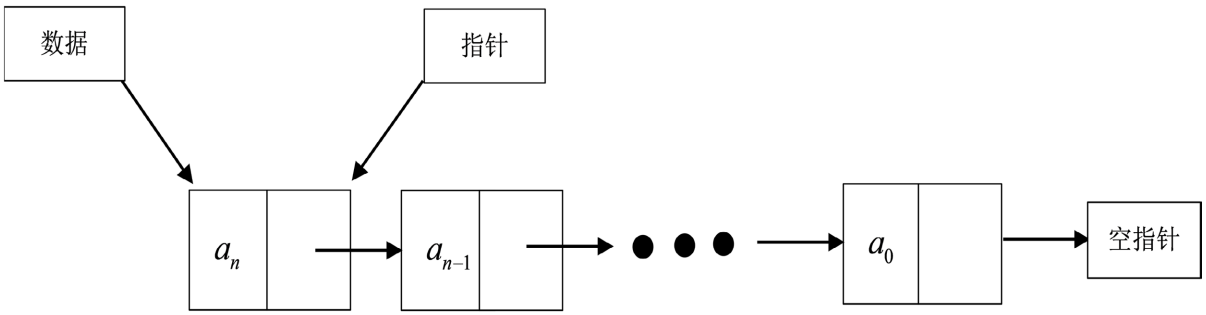

Figure 1. Linked list structure diagram
图 1. 链表结构示意图

代码 1：单向链表的储存逻辑

```
struct listnode
{
    int data;
char* name;//储存的数据
    listnode* rest;//指向下一个节点地址的结构体指针
};
```

切片保存为结构体 *Slice* 形式，它是程序通过识别输出变量，标记输入变量父节点，分割输入锥后生成的。其中保存了切片的变量集合，门多项式集合，以便稍后的按列约简操作。程序根据输出的输入锥定义切片，并根据从电路输入读取的 *level* 对切片中的变量进行排序。这确保了变量是拓扑排序的，并且相应的多项式具有 UMLT，从而形成 D-Gröbner 基。

在读取，切片，重写电路后，函数 *init_slices()*对各个切片所有变量进行排序，将参与切片的变量存入变量集，并根据变量间的关系生成门多项式存入多项式集中。

根据 *D-reduction* 算法，验证电路需要用每个门多项式对电路规范进行约简，函数 *reducing_non_inc()* 实现了这一功能，它遍历所有生成的切片，并利用循环，用切片中的多项式集对电路规范约简。其中调用了函数 *generate_non_inc_spec()*根据稍前读取的电路信息生成电路规范 *spec*；使用变量 *tmp* 把约简的结

果储存起来，每次循环后更新 spec，在下一个循环中用新的 *spec* 进行约简；调用函数 *reduce_by_one_poly()* 进行两个多项式的约简运算；在最后判断 *spec* 是否被约简为 *0*，并输出对应信息：

算法：约简过程

输入：指向 *Polynomial* 结构的指针 *p1*，指向 *Polynomial* 结构的指针 *p2*
输出：指向 *Polynomial* 结构的指针

1. 计算多项式 *p1* 和 *p2* 的首项之间的除法，结果存储于 *negfactor*；
2. 将 *negfactor* 取反；
3. 将 *negfactor* 与多项式 *p2* 相乘，结果存储于 *mult*；
4. 将多项式 *p1* 与多项式 *mult* 相加，结果存储于 *rem*；
5. 返回多项式 *rem*；

程序中约简算法的核心是函数 *reduce_by_one_poly()*，它遵循约简原则对两个多项式在既定的序关系下进行约简。它首先调用函数 *divide_lm()*，用除式 *p2* 的首项对被除式 *p1* 作用除法，获得商 *negfactor*，最后调用多项式乘法函数 *multiply_poly()*和加法函数 *add_poly()*返回余式 *rem=p1-p2*negfactor*。函数 *add_poly()*将符合条件的单项式放入堆栈中，利用栈操作构建新的多项式，算法如下：

算法：函数 *add_poly* 的算法

输入：指向 *Polynomial* 结构的指针 *p1*，指向 *Polynomial* 结构的指针 *p2*
输出：指向 *Polynomial* 结构的指针

1. 当 *p1* 和 *p2* 都非空时循环：
2. a. 如果 *p1* 的首项大于 *p2* 的首项：
3. 将 *p1* 的首项复制入堆栈；
4. 将 *p1* 更新为下一个节点；
5. b. 如果 *p1* 的首项小于 *p2* 的首项：
6. 将 *p2* 的首项复制入堆栈；
7. 将 *p2* 更新为下一个节点；
8. c. 否则：
9. 复制 *p1* 的首项到 *m*；
10. 将 *p1*，*p2* 的首项系数相加并赋值给 *m* 的系数；
11. 如果 *m* 的系数为 *0*，则释放 *m*；
12. 否则将 *m* 放入堆栈；
13. 将 *p1* 和 *p2* 分别更新为各自的下一个节点；
14. 当 *p1* 非空时循环：
15. a. 将 *p1* 的首项复制并按排序规则压入堆栈；
16. b. 将 *p1* 更新为下一个节点；
17. 当 *p2* 非空时循环：
18. a. 将 *p2* 的首项复制并按排序规则压入堆栈；
19. b. 将 *p2* 更新为下一个节点；
20. 从堆栈构建新的多项式 *p*；
21. 返回 *p*；

函数 *contianed()*被函数 *divide_lm()*调用作为条件判断，其功能是判断项 *p2* 是否包含 *p1* 全部变量；它使用了循环嵌套，对于每个 *p1* 中的元素，遍历 *p2*，检查是否在其中；具体算法如下：

算法：函数 *contained* 的算法

输入：指向 *Term* 结构的指针 *m1*，指向 *Term* 结构的指针 *m2*
输出：整型数字

1. 初始化变量 *found* 为 *0*；
2. 将指针 *tmp1* 初始化为 *m1*；
3. 当 *tmp1* 不为空时循环：
4. a. 将指针 *tmp2* 初始化为 *m2*；
5. b. 当 *tmp2* 不为空时循环：
6. 如果 *tmp1* 等于 *tmp2*，将 *found* 设为 *1* 并退出循环；
7. 否则，将 *tmp2* 更新为下一个节点；
8. c. 如果 *found* 等于 *0*，则返回 *0*；
9. d. 将 *found* 重置为 *0*；
10. e. 将 *tmp1* 更新为下一个节点；
11. 返回 *1*；

3.2. 分析与优化

对 *add_poly* 函数的原始实现进行分析，我们发现其采用了一种基于全局栈的方法来处理多项式加法操作。在此方法中，多项式中的每个单项式首先被复制，随后推入一个全局栈中。待所有相关操作完成之后，栈中的元素被逐一取出，用以构建出最终的多项式。虽然这种方法在某些情况下比较有效，但它也带来了几个潜在的问题和挑战。

首先，该函数强烈依赖于全局栈的状态，这种依赖，尤其是在复杂的系统环境中，经常会导致数据竞争和同步问题，增加了软件的不稳定性。

其次，该函数中每次的调用都需要将单项式推入栈中，然后再将它们从栈中取出以构建最终的多项式。这种做法引入了栈操作的中间步骤，增加了执行过程的时间开销，也提高了处理每个单项式所需的计算资源需求。这种额外的时间和资源开销可能成为性能瓶颈，影响整体应用的响应速度和效率。

最后，使用栈作为中间数据结构增加了代码的复杂性，增加了程序的理解、维护和扩展的难度。对于维护者和开发者而言，去理解栈的操作逻辑及其与多项式处理之间的关联变得更加不易，这不仅降低了代码的可读性，也增加了开发和维护的工作量。

在分析 *contained()*函数时，我们发现了一个效率问题，即该函数依赖于嵌套循环。为了确定一个多项式的所有项是否完全包含于另一个多项式中，该函数逐一比较第一个多项式(记为 *m1*)中的每个项与第二个多项式(记为 *m2*)中的所有项。这种方法在 *m1* 和 *m2* 都具有大量项时会显著增加计算的时间复杂度，尤其是每个项都必须与另一个多项式中的每个项进行比较的场景下。这种比较方法可能引发性能瓶颈，影响验证过程的效率。

此外，*found* 变量的使用进一步增加了代码的复杂性。该变量的目的是标记在遍历 *m2* 时是否找到了与 *m1* 中当前项匹配的项。然而，*found* 需要在每次外层循环迭代后重置，使得函数的逻辑变得难以理解，从而降低了代码的可读性和可维护性。这种管理机制不仅对于开发者来说增加了理解和维护代码的难度，也为优化和功能扩展设置了障碍。

在处理复杂且数量庞大的多项式时，上述函数逻辑存在明显的局限性，影响了程序的执行效率。本文提出了一种新的优化方案，利用 C 语言中指针的特性，通过指针操作优化多项式处理函数。

指针是 C 语言中一种强大的工具，允许程序直接访问内存地址，从而操作存储在相关位置的数据。通过指针，开发者可以实现更为高效和灵活的内存管理。尤其是在构建数据结构、实现函数参数的传递

和操作等方面，指针操作展现出了无可比拟的优势：不仅提升了程序的执行效率，也极大地增强了代码的灵活性和复用性。

C 语言的指针在软件开发和系统编程中占有重要地位。指针直接操作内存，可以创建更为高效的数据处理逻辑，如链表、树等动态数据结构的实现，这些结构在处理大量数据和实现复杂算法时表现出极高的效率和灵活性。此外，指针的使用减少了对额外内存的需求，直接操作内存地址以传递复杂的数据结构，避免了数据的不必要复制，进而降低了程序的内存消耗同时提升了性能[18]。

我们针对 *add_poly* 函数进行优化，对原始函数的实现逻辑进行了重新设计，并提出了一个新的实现方案，该方案通过指针操作，利用链表的结构特点，直接构建新的多项式链表来替代原始版本的栈操作进行加法，从而避免了对全局栈的依赖。

算法：优化函数 *add_poly*

输入：指向 *Polynomial* 结构的指针 *p1*,指向 *Polynomial* 结构的指针 *p2*
输出：指向 *Polynomial* 结构的指针

1. 1. 初始化指向 *Polynomial* 的指针 *result* 为空；
2. 2. 创建一个指针 *ptr*，指向 *result* 的地址；
3. 3. 当 *p1* 和 *p2* 都非空时循环：
4. a. 比较 *p1* 和 *p2* 当前首项的大小；
5. b. 如果 *p1* 的首项较大：
6. 在 *ptr* 指向的地址创建一个新 *Polynomial* 节点；
7. 复制 *p1* 的首项到新节点的首项；
8. 将 *p1* 更新为下一节点；
9. c. 如果 *p1* 的首项较小：
10. 在 *ptr* 的地址创建一个新的 *Polynomial* 节点；
11. 复制 *p2* 的首项到新节点的首项；
12. 将 *p2* 更新为下一节点；
13. d. 如果 *p1* 和 *p2* 的首项相同：
14. 复制 *p1* 的首项到临时单项式 *m*；
15. 将 *p1* 和 *p2* 的首项系数相加;如果不为 *0*，则在 *ptr* 地址创建一个新的 *Polynomial* 节点，并将 *m* 设为该节点的首项；
16. 如果结果为 *0*，则释放 *m*；
17. 更新 *p1* 和 *p2* 为各自的下一节点；
18. e. 如果 *ptr* 非空，则更新 *ptr* 为该节点的下一个地址；
19. 4. 如果 *p1* 非空，则将 *p1* 剩余的项添加到结果多项式中；
20. 5. 如果 *p2* 非空，则将 *p2* 剩余的项添加到结果多项式中；
21. 6. 返回指向结果多项式的指针 *result*；

重构后的 *add_poly* 函数彻底消除了全局变量的使用，显著提高了函数的重入性和线程安全性。在并发执行环境下，该函数展现出更为稳定的性能，减少了数据竞争的风险，保证了算法在高并发场景下的可靠性。

新的实现减少了不必要的中间步骤，直接在遍历输入多项式的过程中构建结果多项式，从而提升了整体操作的效率。这种效率的提升突出体现在处理包含大量项的复杂多项式时，能够加速如大型电路验证这样高复杂度的过程。

优化后的方案简化了加法逻辑，使得代码更加直观易懂，提高了程序的可读性和可维护性。此外，减少对全局状态的依赖不仅提升了代码的模块化程度，也为功能扩展和优化提供了便利。这种设计思路

符合现代软件工程的实践逻辑。

算法：优化函数 *contained*

输入：指向 *Term* 结构的指针 *m1*，指向 *Term* 结构的指针 *m2*
输出：整型数字

```
初始化指针 tmp1 为 m1；
初始化指针 tmp2 为 m2；
当 tmp1 和 tmp2 都非空时循环：
    a. 如果 tmp2 当前变量的级别小于 tmp1 当前变量的级别，则
        更新 tmp2 为下一个节点；
    b. 如果 tmp2 当前变量的级别等于 tmp1 当前变量的级别，则
        更新 tmp1 为下一个节点；
        更新 tmp2 为下一个节点；
    c. 如果 tmp2 当前变量的级别大于 tmp1 当前变量的级别，则
        返回 0；
如果 tmp1 仍然非空，则
    a. 返回 0；
返回 1；
```

由于在 *init_slices()*函数调用期间，每个变量的 *level* 值被更新为独一无二的值，所以允许我们使用这些 *level* 值作为索引进行直接比较，来确定变量的等价性。根据这个前提，我们使用双指针方法代替原来的循环嵌套优化函数 *contained*：通过双指针同时遍历参数项，比较变量 *level* 值进行判断。修改后的 *contained* 函数的判断逻辑更为高效、更为清晰。与原始基于嵌套循环的实现相比，新方法能够以线性时间而非二次的时间复杂度完成相同的任务。

新函数通过比较变量的 *level* 值，更加充分地结合程序中的变量构造，允许函数在更抽象的层次上进行比较，提高了运用过程的灵活性，也为程序的功能扩展提供基础。如果有支持更复杂的多项式结构或者引入新的比较逻辑的需求，优化后的函数可以更容易地适应这些变化。同时，优化的函数移除 *found* 变量简化状态管理，避免了原有函数中的复杂逻辑，使得代码更加直观和易于维护。

4. 对比实验

实验对比分析了原始的 *y* 验证程序与其改进版本的性能。由于对任何结构的乘法器，程序在进行约简操作时采用的约简算法均相同，因此，本研究选择以 btor 乘法器和 Kojevnikov 乘法器作为案例来进行分析对比。二者的基准测试由 Boolector 工具生成，并以 AIG 格式进行存储[19]。

本文的实验分析将以表格形式阐述两种程序在处理乘法器时的效率与效果对比，以评估改进后的验证程序在实际应用中的性能表现。

本研究在配备 AMDRyzen 9 5900HX 处理器(主频为 3293.816 MHz)和 8 GB 内存的 Ubuntu 23.04 操作系统环境下的虚拟机中进行了实验，通过图表对实验结果进行对比展示，以便评估优化后的验证程序对乘法器电路验证的性能提升。具体结果如表 1。

根据以下数据我们发现，与原始程序相比，改进后的程序在执行效率上有所提升。特别是在处理大规模乘法器电路时，这种性能优势更为显著。

优化后的程序提高了验证程序处理复杂乘法器电路的能力，展示了其在大规模数字电路验证领域的潜在应用价值。因此，我们认为本文的优化不仅改善了程序的总体性能，而且对于那些涉及大量计算和数据处理的高复杂度电路验证任务，尤其有利。

Table 1. Multiplier experiment data
表 1. 乘法器实验数据

乘法器	位数	原程序(秒)	优化程序(秒)
btor	128	6.18	6.09
btor	256	58.39	58.26
btor	512	616.39	575.57
Kojevnikov	128	5.67	4.91
Kojevnikov	256	46.84	44.96
Kojevnikov	512	454.42	443.18

5. 结语

本文通过对 Amulet 多项式算法的优化，特别是利用指针操作快捷清晰的优点对函数进行重写，将原算法中通过栈操作构建多项式的步骤转换为指针操作，成功缩短了验证时间，并通过实验数据验证了其在大型乘法器验证中的实际优势。

就目前而言，优化的效率提升在 btor 乘法器上更为明显，其他更为复杂的乘法器验证还有很大优化空间。在未来工作中，有希望通过重构多项式结构的形式，重写程序中的结构体，对程序中的多项式运算算法进行进一步优化。

参考文献

[1] Sharangpani, H. and Barton, M.L. (1994) Statistical Analysis of Floating Point Flaw in the Pentium Processor. Intel Corporation.

[2] Bryant, R.E. (1986) Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, **35**, 677-691. https://doi.org/10.1109/tc.1986.1676819

[3] Bryant, R.E. and Chen, Y. (2001) Verification of Arithmetic Circuits Using Binary Moment Diagrams. *International Journal on Software Tools for Technology Transfer*, **3**, 137-155. https://doi.org/10.1007/s100090100037

[4] Chen, Y. and Bryant, R.E. (1995) Verification of Arithmetic Circuits with Binary Moment Diagrams. 32*nd Design Automation Conference*, San Francisco, 12-16 June 1995, 535-541. https://doi.org/10.1109/DAC.1995.250005

[5] Biere, A. (2016) Collection of Combinational Arithmetic Miters Submitted to the SAT Competition 2016. *SAT Competition* 2016, 65-66.

[6] Biere, A. (2016) Weaknesses of CDCL Solvers, August 2016. Fields Institute Workshop on Theoretical Foundations of SAT Solving. http://www.fields.utoronto.ca/talks/weaknesses-cdcl-solvers

[7] Kaufmann, M. and Moore, J.S. (2019) ACL2 Version 8.2. http://www.cs.utexas.edu/users/moore/acl2/

[8] Kaufmann, D. (2020) AMulet 1.5. https://github.com/d-kfmnn/amulet

[9] Buchberger, B. (1965) Ein Algorithmuszum Auffinden der Basiselemente des Restklassenringesnacheinemnulldimensiona-lenPolynomideal. Ph.D. Thesis, University of Innsbruck.

[10] Ciesielski, M.J., Su, T., Yasin, A. and Yu, C. (2019) Understanding Algebraic Rewriting for Arithmetic Circuit Verification: A Bit-Flow Model. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **39**, 1346-1357.

[11] Kaufmann, D. (2020) Formal Verification of Multiplier Circuits using Computer Algebra. Ph.D. Thesis, Johannes Kepler University Linz.

[12] Mahzoon, A., Große, D. and Drechsler, R. (2019) RevSCA: Using Reverse Engineering to Bring Light into Backward Rewriting for Big and Dirty Multipliers. *Proceedings of the* 56*th Annual Design Automation Conference* 2019, Las Vegas, 2-6 June 2019, 1-6. https://doi.org/10.1145/3316781.3317898

[13] Kaufmann, D., Biere, A. and Kauers, M. (2019) Verifying Large Multipliers by Combining SAT and Computer Alge-

bra. 2019 *Formal Methods in Computer Aided Design* (*FMCAD*), San Jose, 22-25 October 2019, 28-36 https://doi.org/10.23919/FMCAD.2019.8894250

[14] Cox, D., Little, J. and O'Shea, D. (2015) Ideals, Varieties, and Algorithms. 4th Edition, Springer-Verlag.

[15] Becker, T., Weispfenning, V. and Kredel, H. (1993) Gröbner Bases, volume141 of Graduate Texts in Mathematics. Springer. https://doi.org/10.1007/978-1-4612-0913-3_5

[16] Sayed-Ahmed, A., Groβe, D., Kühne, U., Soeken, M. and Drechsler, R. (2016) Formal Verification of integer Multipliersby Combining Gröbner basis with Logic Reduction. *DATE* 2016: 2016 *Design, Automation & Test in Europe Conference & Exhibition*, Dresden, 14-18 March 2016, 1048-1053. https://doi.org/10.3850/9783981537079_0248

[17] Biere, A. (2017) The AIGER And-Inverter Graph (AIG) Format, 20071012. Institute for Formal Models and Verification.

[18] Kernighan, B.W. and Ritchie, D.M. (1988) The C Programming Language. 2nd Edition, Prentice Hall.

[19] Niemetz, A., Preiner, M. and Biere, A. (2015) Boolector 2.0 System Description. *Journal on Satisfiability, Boolean Modeling and Computation*, **9**, 53-58 https://doi.org/10.3233/SAT190101