
上海交通大学学士学位论文

# VST-IDE：交互式程序验证工具（分离逻辑求解）

**姓　　名**：唐亚周

**学　　号**：519021910804

**导　　师**：曹钦翔教授

**院　　系**：电子信息与电气工程学院

**学 科/专 业**：计算机科学与技术

**2023 年 5 月**

**A Dissertation Submitted to**

**Shanghai Jiao Tong University for Bachelor Degree**

# VST-IDE: INTERACTIVE PROGRAM VERIFICATION TOOL (SEPARATION LOGIC SOLVING)

**Author:** Tang Yazhou

**Supervisor:** Prof. Cao Qinxiang

School of Electronic, Information and Electrical Engineering

Shanghai Jiao Tong University

Shanghai, P.R. China

May $7^{th}$, 2023

# 上海交通大学

# 学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全知晓本声明的法律后果由本人承担。

学位论文作者签名：唐亚周

日期： 2023 年 5 月 7 日

# 上海交通大学

# 学位论文使用授权书

本人同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。

本学位论文属于 ：

☑**公开论文**

☐**内部论文**，保密☐1 年/☐2 年/☐3 年，过保密期后适用本授权书。

☐**秘密论文**，保密____年（不超过 10 年），过保密期后适用本授权书。

☐**机密论文**，保密____年（不超过 20 年），过保密期后适用本授权书。

（请在以上方框内选择打“√”）

学位论文作者签名：唐亚周　　指导教师签名：

日期：2023 年 5 月 7 日　　日期：2023 年 5 月 7 日

# 摘 要

形式化验证是保障软件正确性的一种手段，该方法将程序通过一定的逻辑规则抽象为数学命题并对其进行证明，从而保证程序的完全正确。形式化验证在航空航天等对安全性要求极高的领域有着广泛的应用，但其对验证人员的专业知识要求较高，验证成本也较高。本项目的主要目的是降低 C 语言程序验证的门槛和成本，并且尝试将验证结合到开发过程中，帮助软件开发者提高验证效率，实现“开发即安全”。

VST-IDE 项目分为编译前端、符号执行、蕴含关系求解三个部分，本文主要介绍蕴含关系求解，该部分的目的是判断符号执行得到的断言是否能够推导出用户在注释中定义的断言。本项目对于断言中的空间断言部分进行了模块化的设计，通过编写不同的子模块，可以支持不同数据结构的求解。在求解过程中，本项目还进行了一些特殊的操作，从而能够处理一些特殊情况。最后，本项目会将求解过程以证明规则和 VST 证明代码的形式输出，以便用户验证求解过程的正确性。

本项目目前实现了对非递归数据结构、单链表和双链表的常见操作的验证支持，对于二叉树的操作验证支持正在完善中。

**关键词：**程序验证，定理证明，分离逻辑，VST，Coq

# ABSTRACT

Formal verification is a means of ensuring software correctness, whereby a program is abstracted into mathematical propositions and verified through logical rules, ensuring complete program accuracy. Formal verification finds extensive applications in fields such as aviation and space industry where safety is crucial. However, it requires a high level of expertise and verification cost is also high. The main objective of this project is to reduce the threshold and cost of C program verification, and merge verification with the development process, thus helping software developers to improve verification efficiency and achieve "safety in development".

The VST-IDE project consists of three parts: compilation front-end, symbolic execution, and entailment relation solving. This article focuses on entailment relation solving, which aims to determine whether the assertion obtained from symbolic execution can infer the assertion defined by the user in the comments. This project has modularized the spatial assertion section of the assertion, and by implementing different sub-modules, it can support the solving of different data structures. During the solving process, this project also performs some special operations to handle specific cases. Finally, this project outputs the solving process in the form of proof rules and VST proof code for users to verify the correctness of the solving process.

Currently, this project has realized the verification support for common operations of non-recursive data structures, single-linked lists, and double-linked lists. Verification support for binary tree operations is still under development.

**Key words:** Program Verification, Theorem Proving, Separation Logic, VST, Coq

# 目 录

# 插 图

# 表 格

# 算 法

# 符号对照表

| | |
|---|---|
| `Res` | 蕴含关系左边的断言 |
| `Tar` | 蕴含关系右边的断言 |
| `Res.Prop` | `Res` 的 PROP 部分 |
| `Res.Local` | `Res` 的 LOCAL 部分 |
| `Res.Sep` | `Res` 的 SEP 部分 |
| `Res.Exist` | `Res` 的 EX 部分 |
| `Tar.Prop` | `Tar` 的 PROP 部分 |
| `Tar.Local` | `Tar` 的 LOCAL 部分 |
| `Tar.Sep` | `Tar` 的 SEP 部分 |
| `Tar.Exist` | `Tar` 的 EX 部分 |
| [] | 空列表 |
| :: | 单个元素添加到列表的头部 |
| ++ | 两个列表的连接 |

# 第一章 绪论

## 1.1 研究背景

随着计算机科学的不断发展，软件在人们生活中的重要性日益增加。软件的漏洞可能会带来重大的生命安全和财产损失，因此保障软件的正确性和安全性是软件开发过程中的重要环节。在软件工程领域，软件测试（software testing）是保障软件正确性的重要手段之一。软件测试有着许多有点，比如效率高、易维护等，但其缺点在于无法覆盖到全部情况。在一些安全攸关（safety critical）的场景中，比如航空航天、医疗设备、自动驾驶等领域，我们对软件正确性的要求极高，这时就需要采用其它方式。

形式化验证（formal verification）是指通过使用形式化方法来证明软件系统符合其规范或需求的技术。在软件工程领域，形式化验证是提高软件系统可靠性和安全性的重要手段之一。定理证明（theorem proving）是形式化验证的一种方法，它将程序和系统的正确性表达为数学命题, 然后使用逻辑推导的方式证明正确性。不同于基于程序测试的技术, 定理证明方法能保证覆盖所有边缘情况，完全排除一些特定类型的错误。[1-2]

## 1.2 相关工作介绍

自从分离逻辑[3]被提出以来，已经有许多基于分离逻辑的程序验证工作发表，这里我们介绍一些比较有代表性的工作。[4-5]

基于分离逻辑的程序验证相关工作最早可以追溯到 Berdine 等人在 2005 年提出的 Smallfoot 项目[6-7]。该工作采用了符号执行与蕴含检查交替进行的方法，在符号执行过程中更新抽象程序状态。Smallfoot 的输入语言是为匹配该工作而设计的，并非日常中广泛使用到的编程语言。该工作对分离逻辑自动化验证的模块化进行了尝试，给出了几种链表和树结构的硬编码（hardwired）谓词，但并没有给出对于任意数据结构的归纳定义方式。

2006 年，Distefano 等人[8]将形状分析（shape analysis）与分离逻辑相结合，提出了局部形状分析（local shape analysis）。该工作提出了基于抽象解释的不动点求解方法进行了循环不变量的推导，从而提高了程序验证的自动化程度。该工作的仅支持链表数据结构的验证。

2008 年，Yang 等人开发了 SpaceInvader 工具[9]，并利用其完成了对 1 万行以上的设备驱动代码的自动验证。该工作首次讲自动分析与验证技术应用到工业级代码中。但该工作所处理的程序仅限于链表数据结构。

2009 年，在 SpaceInvader 工作的基础上，Calcagno 等人提出了基于双向诱导（bi-abduction）技术的可组合式形状分析系统 Abductor[10-11]。该工作对程序的每个过程进行单独分析，然后根据过程调用关系进行组合。

2010 年，Jacobs 等人发表了 VeriFast 工具[12-13]，用于对带前后置条件的 C 语言和 Java 语言程序进行验证。

2011 年，Appel[14]提出了经验证的软件工具链（Verified Software Toolchain, VST），这也是本项目的前置工作之一，具体内容会在2.4小节中介绍。

2016 年，Leroy 等人[15]发表了 Compcert，一个经过形式化验证的 C 语言优化编译器。Compcert 保证了编译后的代码与源代码的语义等价，且编译器本身不包含错误。

2018 年，Cao 等人[16]提出了 VST-Floyd 验证助手。该工具基于 Veifiable C 程序逻辑[17]，提供了一套半自动化的证明策略，用户可以在 Coq 中使用这些证明测罗来完成 C 语言程序的功能正确性验证。

## 1.3 本文解决的主要问题

本项目验证的对象是 C 语言程序。C 语言是一种通用的程序设计语言，广泛应用于系统软件、应用软件、嵌入式软件等领域。C 语言的语法简洁，易于理解，但其语义复杂，且易出错。因此，对 C 语言程序进行形式化验证是一项具有挑战性的工作。

本项目验证的重点是内存安全性（memory safety），比如对于内存泄漏、空指针引用、非法内存访问等问题的检测。本项目所验证的程序包含函数调用，因此是一种跨过程分析。

本项目的主要目的是降低 C 语言程序验证的门槛和成本，并且尝试将验证结合到开发过程中，帮助软件开发者提高验证效率，实现“开发即安全”。

## 1.4 本文结构

第二章将介绍本文所涉及的相关技术背景。第三章将介绍本项目的整体框架。第四章将介绍求解器的具体实现。第五章将介绍求解器的测试样例，并通过一个例子来

进行详细介绍。第六章将介绍本文的总结与展望。

# 第二章 相关技术介绍

## 2.1 霍尔逻辑

1967 年，Robert W. Floyd 最早将逻辑断言引入程序验证[18]，提出了一种基于流程图的表示方法。1969 年，C. A. R. Hoare 基于 Floyd 的工作提出了一套形式系统[19]，使用一套逻辑规则来严格推理计算机程序的正确性。这套形式系统被称作霍尔逻辑（Hoare Logic），又称弗洛伊德-霍尔逻辑（Floyd-Hoare Logic）。

霍尔逻辑的中心特征是霍尔三元组（Hoare triple），其基本结构如下：

$$\{P\}C\{Q\},$$

其中，$P$ 和 $Q$ 是断言，$C$ 是程序指令。霍尔三元组的意义是：如果断言 $P$ 在程序 $C$ 执行前为真，那么程序 $C$ 执行后断言 $Q$ 为真。这里我们也把 $P$ 称为前置条件（Precondition），$Q$ 称为后置条件（Postcondition）。

霍尔逻辑有以下几条推理规则：

$$\textbf{Skip}\quad \frac{}{\{P\}\texttt{skip}\{P\}} \qquad \text{空语句规则} \qquad (2\text{–}1)$$

$$\textbf{Assign}\quad \frac{}{\{P[e/x]\}\texttt{x:=e}\{P\}} \qquad \text{赋值语句规则} \qquad (2\text{–}2)$$

$$\textbf{Seq}\quad \frac{\{P\}\texttt{C1}\{Q\}\quad \{Q\}\texttt{C2}\{R\}}{\{P\}\texttt{C1;C2}\{R\}} \qquad \text{顺序语句规则} \qquad (2\text{–}3)$$

$$\textbf{If}\quad \frac{\{P\wedge B\}\texttt{C1}\{Q\}\quad \{P\wedge\neg B\}\texttt{C2}\{Q\}}{\{P\}\texttt{if(B)then\{C1\}else\{C2\}}\{Q\}} \qquad \text{条件语句规则} \qquad (2\text{–}4)$$

$$\textbf{While}\quad \frac{\{P\wedge B\}\texttt{C}\{P\}}{\{P\}\texttt{while(B)do\{C\}}\{P\wedge\neg B\}} \qquad \text{循环语句规则} \qquad (2\text{–}5)$$

$$\textbf{Pre}\quad \frac{P\Rightarrow P'\quad \{P'\}\texttt{C}\{Q\}}{\{P\}\texttt{C}\{Q\}} \qquad \text{前置条件强化规则} \qquad (2\text{–}6)$$

$$\textbf{Post}\quad \frac{\{P\}\texttt{C}\{Q'\}\quad Q'\Rightarrow Q}{\{P\}\texttt{C}\{Q\}} \qquad \text{后置条件弱化规则} \qquad (2\text{–}7)$$

## 2.2 分离逻辑

由于霍尔逻辑无法处理指针的问题，所以在处理指针时，需要使用分离逻辑（Separation Logic）[3]。分离逻辑是霍尔逻辑的扩展，它能够直观地分析程序中复杂的动态内存变化。分离逻辑引入了四种新的断言形式，从而能够更加方便和精确地描述程序

的内存状态：

$$
\begin{aligned}
&\langle \text{ assert } \rangle ::= \cdots \\
&\quad | \ \text{emp} && \text{空堆} \\
&\quad | \ \langle \text{exp} \rangle \mapsto \langle \text{exp} \rangle && \text{单堆} \\
&\quad | \ \langle \text{ assert } \rangle * \langle \text{ assert } \rangle && \text{分离合取} \\
&\quad | \ \langle \text{ assert } \rangle -\!\!* \langle \text{ assert } \rangle && \text{分离蕴含}
\end{aligned}
$$

分离逻辑引入了以下几条推理规则[20]：

$$\textbf{Alloc} \quad \frac{}{\{\text{emp}\}\texttt{x=alloc()}\{x \mapsto -\}} \qquad \text{内存分配规则} \qquad (2\text{–}8)$$

$$\textbf{Deloc} \quad \frac{}{\{x \mapsto -\}\texttt{free(x)}\{\text{emp}\}} \qquad \text{内存释放规则} \qquad (2\text{–}9)$$

$$\textbf{Load} \quad \frac{}{\{x \mapsto v\}\texttt{y=*x}\{y = v \wedge x \mapsto v\}} \qquad \text{指针读取规则} \qquad (2\text{–}10)$$

$$\textbf{Store} \quad \frac{}{\{x \mapsto -\}\texttt{*x=v}\{x \mapsto v\}} \qquad \text{指针写入规则} \qquad (2\text{–}11)$$

$$\textbf{Frame} \quad \frac{\{P\}\texttt{C}\{Q\}}{\{P * F\}\texttt{C}\{Q * F\}} \qquad (2\text{–}12)$$

$$\textbf{Concurrency} \quad \frac{\{P_1\}\texttt{process1}\{Q_1\} \quad \{P_2\}\texttt{process2}\{Q_2\}}{\{P_1 * P_2\}\texttt{process1 || process2}\{Q_1 * Q_2\}} \qquad (2\text{–}13)$$

## 2.3 定理证明与 Coq

### 2.3.1 机械化定理证明

机械化定理证明是指用计算机，以定理证明的方式堆数学定理或计算机软、硬件系统进行形式化验证。在早期，机械化定理证明围绕“计算机如何高度自动化地完成证明”而展开，即“自动定理证明（Automated Theorem Proving）”，也取得了许多成果；但随着计算机科学的发展，人们发现许多数学定理的证明是无法自动化的，因此，人们开始研究“如何将人类的数学知识和计算机的计算能力结合起来”来完成定理证明，即“交互式定理证明（Interactive Theorem Proving）”。

机械化定理证明最初旨在证明数学定理，但从 20 世纪 60 年代晚期开始，人们意识到许多其它问题，比如程序属性、集成电路设计等，都可以表示为定理，并且使用定理证明工具进行解决。因此，机械化定理证明逐渐发展成为一种通用的形式化验证方法。[21]

交互式定理证明工具也被称为证明助手（Proof Assistant），目前主流的证明助手有 Coq①、Isabelle/HOL②、HOL4③、Lean④等。本文使用的证明助手是 Coq。

### 2.3.2 Coq

Coq 是一个交互式证明助手和函数式编程语言[22-23]，旨在帮助程序员和数学家使用形式逻辑开发数学证明。它的基础是归纳构造演算（calculus of inductive constructions）[24]，一个具有依赖类型的类型理论。Coq 被广泛用于计算机科学领域，特别是在形式验证、软件工程和人工智能领域。

Coq 最初由法国国立计算机及自动化研究院（Institut National de Recherche en Informatique et en Automatique, INRIA）研发，目前由法国的几所大学和研究所共同维护。Coq 的主要开发者在 2013 年获得了 ACM 软件系统奖（ACM Software System Award）。

## 2.4 VST

经验证的软件工具链（Verified Software Toolchain, VST）[14]是用于 C 程序的功能正确性证明的工具集，它的结构如图2–1所示，包括[25]：

- 一种基于分离逻辑的程序逻辑，称为 Verifiable C[17]；
- 一个名为 VST-Floyd[16]的证明自动化系统，协助用户将程序逻辑应用于程序；
- Coq 中的完备性证明，保证用户证明的关于程序的任何属性都会在 C 源语言操作语义的任何执行中实际成立。而此证明与 CompCert（经验证的 C 语言优化编译器）[15]的正确性证明组合，因此用户还可以获得有关汇编语言程序行为的保证。

## 2.5 SMT 求解器

可满足性模理论（Satisfiabiliti Module Theories, SMT）[26]是对布尔可满足性问题（Boolean Satisfiability Problem, SAT）的扩展，它将布尔逻辑扩展到了一阶逻辑，引入了算术运算、数组、有限集、比特向量、代数数据类型、字符串、浮点数以及各种理

① https://coq.inria.fr/

② https://isabelle.in.tum.de/

③ https://hol-theorem-prover.org/

④ https://leanprover.github.io/

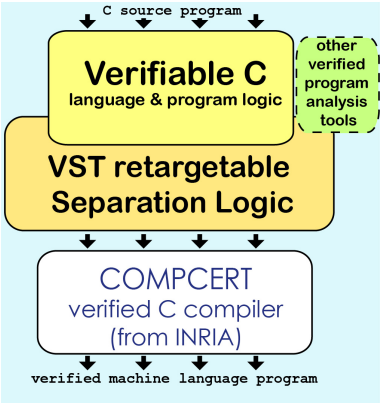


**图 2–1 VST 的结构**

论的结合等。SMT 求解器是一种自动化定理证明工具，它可以用于求解一阶逻辑公式的可满足性问题。SMT 求解器的输入是一阶逻辑公式，输出是“可满足”或“不可满足”。目前主流的 SMT 求解器有 Z3[27]、cvc5[28]、Yices[29]等。

## 2.6 本章小结

本章介绍了本文所涉及的相关背景知识，包括程序验证、分离逻辑、定理证明、Coq、VST、SMT 求解器等。

# 第三章 整体框架

## 3.1 项目概述

VST-IDE 的主要目标是提供一个在线编辑器，用户可以在编辑器中编写 C 语言程序，同时可以在注释中编写断言，从而在开发过程中实现对程序的验证。该项目的原理是，利用修改过的 Compcert 编译器将源代码编译到抽象语法树 (Abstract Syntax Tree, AST)，然后符号执行工具读入 AST，从前置条件开始进行符号执行。在符号执行的过程中，如果在注释中遇到断言，就会进行分离逻辑求解，判断符号执行得到的程序状态是否可以推导出该断言，从而起到验证的作用。

它与现有的验证工具相比，主要优势在于自动化程度更高：用户只需要对于每个函数编写其规约（specification），以及在函数内遇到循环时编写循环不变量（loop invariant），而不需要编写任何其他代码，就可以实现验证。

因此，本项目分为三个部分，分别是编译前端、符号执行和基于分离逻辑的蕴含关系检验。前两个部分分别由另外两名同学完成，这里仅做简略介绍；第三个部分由我完成，也是本文的重点。

### 编译前端

在编译前端，我们对 Compcert 进行了修改，使得它可以编译 “部分程序”，从而将验证的最小单位从一个完整的函数细化到了一行代码。这样就可以让用户在开发的过程中进行 “单步” 的验证，而不是把整个程序写完再进行验证，从而提高开发和验证的效率。

此外，由于 VST 的断言较为复杂，为了降低用户的学习成本，我们还定义了一套 C 语言风格的 “断言语言”，用于在 C 语言注释中编写断言。

### 符号执行

符号执行（Symbolic Execution）[30-31]是一种软件测试技术，即通过创建符号变量来代表程序输入，并系统地探索程序的所有可能路径，以确定其行为并识别潜在的错误或漏洞。符号执行不是用具体的输入来运行程序，而是探索任何可能的输入组合所产生的执行路径，从而对程序的行为进行更全面的测试和分析。这种技术经常被用于软件开发和安全测试，以确定难以发现的错误和漏洞。

在本项目中，我们会从前置条件开始进行符号执行，如果中途遇到控制语句，就会根据控制语句的条件进行分支，然后在后续合并时将各个分支得到的程序状态通过析取的方式进行合并。

#### 基于分离逻辑的蕴含关系检验

在 VST 中，断言有着如下的格式：

EX…PROP(…)LOCAL(…)SEP(…).

其中，

- EX：即 Exists，用于表示该断言中的存在变量；
- PROP：即 Proposition，用于表示与内存无关的真命题；
- LOCAL：即 Local，用于表示 C 语言中的局部变量的值，即程序变量与逻辑变量的映射；
- SEP：即 Separation，用于表示分离逻辑中的空间断言，也就是内存空间中的状态。

本项目通过验证符号执行得到的结果是否能推导出用户在注释中编写的断言（后置条件），从而实现验证。举个例子，对于一行 C 语言代码指令 $C$，如果用户在其前后的注释中分别编写了断言 $P$ 和 $Q$，并且我们根据 $P$ 和 $C$ 进行符号执行，得到了新的程序状态 $Q'$，那么如果我们能够证明 $Q' \vdash Q$（蕴含关系成立），那么就说明程序执行到指令 $C$ 时是符合用户的预期的，从而实现了验证。

为了验证我们对于 $Q' \vdash Q$ 的证明是否正确，该部分还会将其求解过程以 VST 证明的格式输出，从而可以通过 Coq 验证其正确性。

## 3.2 整体架构

### 3.2.1 单个断言蕴含单个断言

我们首先考虑单个断言蕴含单个断言的情况，如图3–1所示，蕴含关系求解器由三个部分组成，即 Prop 求解器、Local 求解器和 Sep 求解器，分别用于求解断言的 PROP、LOCAL 和 SEP 部分。在最外层的断言求解器中，我们会按照既定的顺序调用这几个求解器，从而得到最终的结果。

由于对于不同的数据结构，Sep 的求解规则也不同，因此我们对于 Sep 求解器进行了模块化的设计，在整体框架的基础上设计了十余个子函数，并且在各个数据结构

对应的模块中进行了实现。同时，我们设计了 Sep 合并器，用于将各个数据结构的子模块进行合并，从而可以处理多种数据结构的情况。

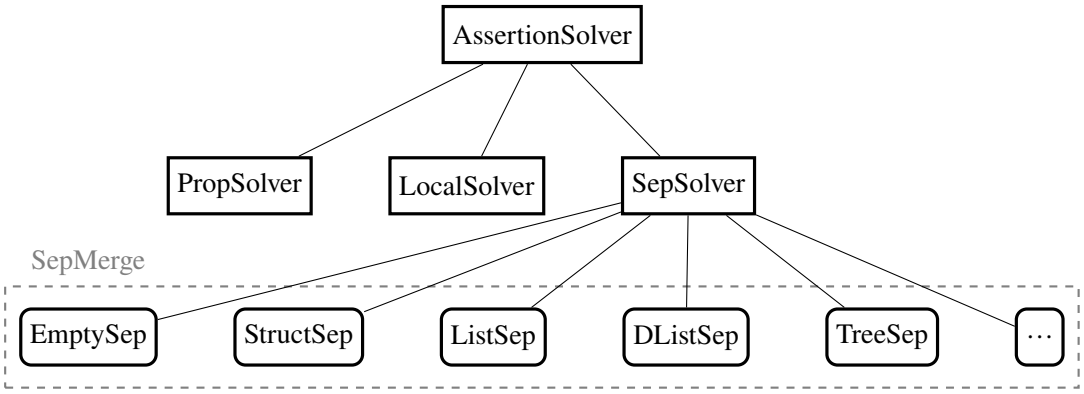


**图 3–1 蕴含关系求解器的结构（单个断言蕴含单个断言）**

### 3.2.2 多个断言蕴含多个断言

由于符号执行过程中如果遇到控制语句就会根据条件进行分支，因此在实际的验证过程中，我们的程序状态可能为多个断言的析取。在这种情况下，蕴含关系求解器需要证明以下的蕴含关系：

$$Q'_1 \vee Q'_2 \vee \cdots \vee Q'_n \vdash Q_1 \vee Q_2 \vee \cdots \vee Q_m, \tag{3–1}$$

这里我们要求对于任意的 $Q'_i$，都存在一个 $Q_j$ 使得 $Q'_i \vdash Q_j$。也就是说，对于符号执行得到的每一个程序状态，都能够找到一个用户编写的断言，使得该程序状态蕴含该断言。因此，多个断言蕴含多个断言的情况可以转化为单个断言蕴含单个断言的情况来解决。

## 3.3 求解流程

我们在求解蕴含关系时，基本的思路是：如果在蕴含关系的两边发现了一模一样的部分，那么就可以把问题转化为求解“两边都去除该部分之后”的蕴含关系。因此，我们会使用“消去”一词来描述这一过程。这一思路也是符合分离逻辑规则的。

根据在 Coq 中编写证明的经验，对于 `Res.Exist`，我们可以把它当作条件中的变量，而不需要关心它们的具体值。并且，我们经常利用它们对 `Tar.Exist` 进行实例化。

为了表达的方便，我们使用了一些简写，如正文前的符号对照表所示。

1. 预处理：
   (a) 去除掉 `Res.Sep` 中实际上为空堆（emp）的部分；
   (b) 从新的 `Res.Sep` 中获得纯事实（pure facts），用于之后的证明；
   (c) 对 `Res.Sep` 进行化简；
   (d) 根据 `Res.Local` 和 `Tar.Local` 的内容，对 `Tar.Exist` 进行实例化（基于 Local 求解器）；
   (e) 去除掉 `Tar.Sep` 中实际上为空堆的部分；
   (f) 根据 `Res.Sep` 和 `Tar.Sep` 的内容，对 `Tar.Exist` 进行实例化（基于 Sep 求解器）；
2. 基于 `Res.Local` 来推导 `Tar.Local`（基于 Local 求解器）；
3. 基于字段（Field）对 `Tar.Sep` 进行拆分，为之后的 Sep 求解器做准备；
4. 基于 `Tar.Sep` 获得可化简的地址（Simplifiable Address, SimplAddr）；
5. 基于 `Res.Sep` 来推导 `Tar.Sep`（基于 Sep 求解器）；
6. 根据之前获得的 SimplAddr，对 `Tar.Sep` 进行化简；
7. 利用 `Res.Prop` 和预处理中获得的纯事实，对 `Tar.Prop` 和求解过程中产生的新的待证明命题进行证明（基于 Prop 求解器）。如果证明失败，则将无法证明的命题返回。

求解器最终会返回“消去”后的 `Res'` 和 `Tar'`，这意味着如果 `Res'` 和 `Tar'` 之间的蕴含关系成立，那么 `Res` 和 `Tar` 之间的蕴含关系也成立，也就是

$$\texttt{Res'} \vdash \texttt{Tar'} \Rightarrow \texttt{Res} \vdash \texttt{Tar}. \tag{3–2}$$

如果 `Tar'.Local`、`Tar'.Sep` 和 `Res'.Sep` 均为空，并且求解的最后一步证明成功（即 `Tar'.Prop` 为空），那么我们就可以认为 `Res` 和 `Tar` 之间的蕴含关系成立。

## 3.4 本章小结

本章首先介绍了项目的整体结构，然后具体介绍了蕴含关系求解器这一部分的结构和求解流程。

# 第四章 具体实现

## 4.1 基本定义

在介绍求解器的具体实现之前，我们首先需要定义一些基本的类型。

### 4.1.1 Ctype 类型类（typeclass）定义

类型类（typeclass）[32]，也就是“类型”的“类”，是支持特设多态（ad hoc polymorphism）的类型系统构造，是通过向参数多态类型的类型变量增加约束完成的。对于一个类型类 `T` 和一个类型变量 `a`，要求对于 `a` 所能实例化的类型，其成员必须支持关联于 `T` 的重载运算。

`Ctype` 代表程序中的变量类型，其定义如下。其中，`ctype` 代表程序中的所有类型，`eqb_ctype` 代表类型的等价判断。

```
Class Ctype := {
  ctype: Type;
  eqb_ctype: ctype -> ctype -> bool;
}.
```

本项目目前没有对程序变量类型进行深入探讨，将所有数据视为 `tt` 类型（`tt` 是 Coq 中的一种单例数据类型 `unit` 的唯一成员），因此一个常见的 `Ctype` 类型对象如下。

```
#[export] Instance C: Ctype := {
  ctype := unit;
  eqb_ctype _ _ := true;
}.
```

### 4.1.2 C 语言注释中的谓词定义

在验证开始之前，我们需要根据结构体的定义，在 C 程序的注释中定义对应的谓词。对于非递归定义的数据结构，我们只需要对其各个数据成员进行描述即可，编译前端会隐式地生成所有需要 load 的内存地址。接下来我们考虑递归定义的数据结构，假设其数据部分均为单个 `int` 类型的变量。

在单链表相关程序中，我们需要定义两种谓词，即 `listrep`（list representation, 单链表）和 `lseg`（list segment representation 单链表段）。其中 `listrep` 有一个参数，代表指向单链表头结点的指针；`lseg` 有两个参数，分别代表指向单链表头结点的指针和指向单链表尾结点的后一个结点的指针。`listrep` 和 `lseg` 的示意图如图4–1和图4–2所示。

```
1  struct list {
2      int head;
3      struct list* tail;
4  };
5  /*@ Let listrep(l) = l == 0 && emp ||
6        exists v, l->head == v && listrep(l->tail)
7   */
8  /*@ Let lseg(x, y) = x == y && emp ||
9        exists v, x->head == v && lseg(x->tail, y)
10  */
```

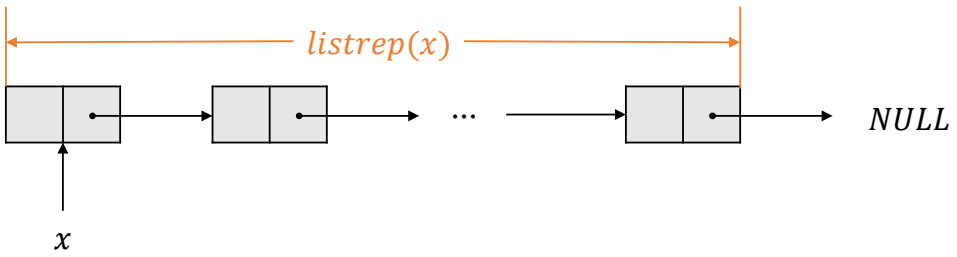


**图 4–1 listrep 谓词示意图**

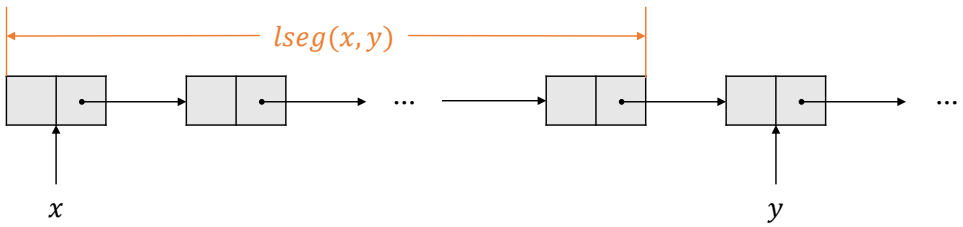


**图 4–2 lseg 谓词示意图**

在双链表相关程序中，我们也需要定义两种谓词，分别是 dlistrep（doubly linked list representation，双链表）和 dlseg（doubly linked list segment representation，双链表段）。其中 dlistrep 有两个参数，分别代表指向双链表头结点的指针和指向双链表头结点的前一个结点的指针；dlseg 有四个参数，分别代表指向双链表头结点的指针、指向双链表头结点的前一个结点的指针、指向双链表尾结点的后一个结点的指针和指向双链表尾结点的指针。dlistrep 和 dlseg 的示意图如图4–3和图4–4所示。

```
1  struct dlist {
2      int head;
3      struct dlist* prev;
4      struct dlist* next;
5  };
6  /*@ Let dlistrep(l, p) = l == 0 && emp ||
7        l->prev == p && dlistrep(l->next, l)
8  */
9  /*@ Let dlseg(x, x_prev, y, y_prev) =
10       x == y && x_prev == y_prev && emp ||
11       x->prev == x_prev && dlseg(x->next, x, y, y_prev)
12 */
```

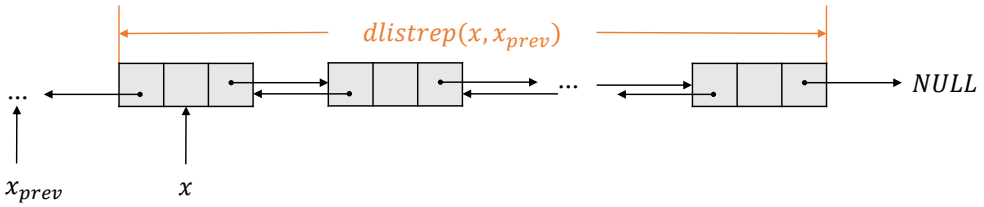


**图 4–3 dlistrep 谓词示意图**

在二叉树相关程序中，我们也需要定义两种谓词，分别是 treerep（tree representation，树）和 partial_treerep(partial tree representation，部分树)。其中 treerep 有两个参数，分别代表指向根结点的指针和指向根结点的父结点的指针；

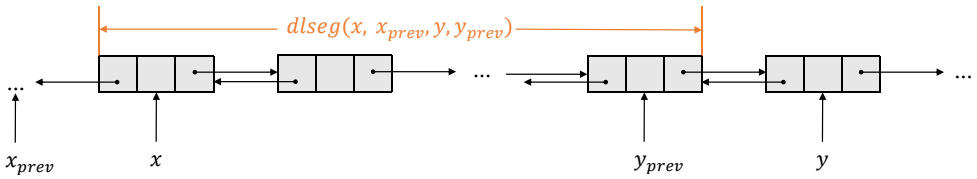


**图 4–4 dlseg 谓词示意图**

partial_treerep 有四个参数，分别代表指向根结点的指针、指向根结点的父结点的指针、指向空缺部分的指针和指向空缺部分的父结点的指针。

```
struct tree {
    int data;
    struct tree* left;
    struct tree* right;
    struct tree* parent;
};
/*@ Let treerep(p, p_par) = p == 0 && emp ||
    exists v p_lch p_rch,
      p->data == v && p->left == p_lch &&
      p->right == p_rch && p->parent == p_par &&
      treerep(p_lch, p) * treerep(p_rch, p)
 */
/*@ Let partial_treerep(p, p_par, p_root, p_top) =
    p == p_root && p_par == p_top && emp ||
    (exists v p_rsib p_gpar,
      p_par->data == v && p_par->left == p &&
      p_par->right == p_rsib && p_par->parent == p_gpar &&
      partial_treerep(p_par, p_gpar, p_root, p_top) *
      treerep(p_rsib, p_par)) ||
    (exists v p_lsib p_gpar,
      p_par->data == v && p_par->left == p_lsib &&
      p_par->right == p && p_par->parent == p_gpar &&
      partial_treerep(p_par, p_gpar, p_root, p_top) *
```

```
24        treerep(p_lsib, p_par))
25  */
```

treerep 和 partial_treerep 的示意图如图4–5和图4–6所示。可以看到，partial_treerep 是 treerep 挖去一个子树后的结果。这里我们用蓝色和绿色对 partial_treerep 的结点进行了标记，相同颜色的结点代表它们是同一棵“半树”（half tree），也就是根结点加上它的左子树或右子树。因此 partial_treerep 也可以看作是半树的双链表段，这样就实现了链表结构和树结构的统一。

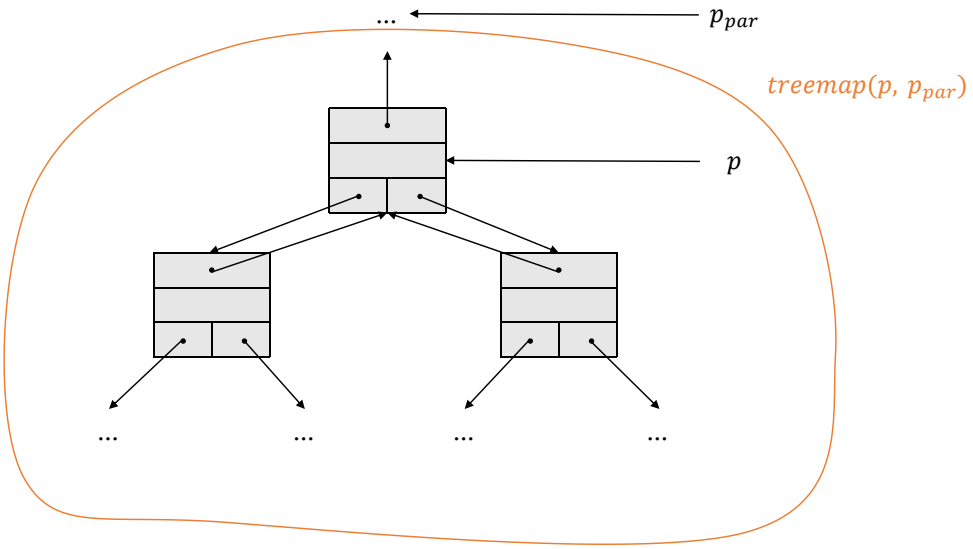


**图 4–5 treerep 谓词示意图**

### 4.1.3 分离逻辑谓词定义

为了实现项目的模块化和可拓展性，我们定义了 SepPred 类型，用于对前一小节中定义的分离逻辑谓词进行统一的表示。SepPred 类型的定义中，pred 构造子代表一个谓词，其第一个参数为谓词的编号，第二个参数为谓词的参数列表。

```
Inductive SepPred: Type :=
  | pred: ident -> list ExprVal -> SepPred.
```

我们另外维护了一个 ident 到数据结构类型的映射，一个可能的定义如表4–1所示。以下三点值得注意：

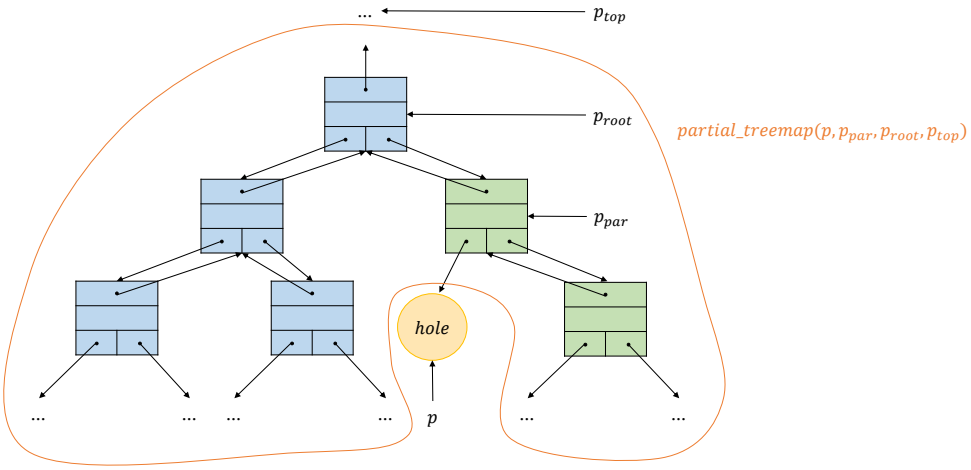


**图 4–6 partial_treerep 谓词示意图**

1. 该表是基于用户在 C 语言程序注释中的定义生成的，我们会识别出用于所定义的是哪一种数据结构，并为其分配编号。因此这里列出的编号分配仅仅是一个例子，实际情况下可能有所不同。
2. 对于同一类型的数据结构，由于其数据字段的情况不同，我们可能会定义多个谓词。比如该表中的“单链表 1”和“单链表 2”，它们都是单链表，但“单链表 1”的数据字段可能为单个 `int` 类型变量，而“单链表 2”的数据字段可能为两个 `int` 类型变量。
3. 对于每个谓词的参数，这里我们使用字母来表示。在实际情况中，这些参数将会是具体的逻辑变量。

**表 4–1 常见数据结构及其谓词定义**

| 数据结构类型 | 编号 | 一个可能的对象 | 对应的 SepPred |
|---|---|---|---|
| 单链表 1 | 1 | `listrep1(x)` | `pred 1 [x]` |
| 单链表 2 | 2 | `listrep2(x)` | `pred 2 [x]` |
| 单链表段 1 | 3 | `lseg(x, y)` | `pred 3 [x; y]` |
| 双链表 1 | 4 | `dlistrep(x, xp)` | `pred 4 [x; xp]` |
| 双链表段 1 | 5 | `dlseg1(x, xp, y, yp)` | `pred 5 [x; xp; y; yp]` |
| 双链表段 2 | 6 | `dlseg2(x, xp, y, yp)` | `pred 6 [x; xp; y; yp]` |
| 树 1 | 7 | `treerep(p, pp)` | `pred 7 [p; pp]` |
| 部分树 1 | 8 | `treerep(p, pp, pr, pt)` | `pred 8 [p; pp; pr; pt]` |

### 4.1.4 空间断言定义

基于程序类型 Ctype 的定义和分离逻辑谓词 SepPred 的定义，我们可以定义出空间断言（spatial assertion），也就是内存空间上的 Separation 类型。空间断言的定义如下所示，它有两个隐式参数，分别为程序类型 ctype 和分离逻辑谓词 SepPred。

```
Inductive Separation {ctype: Type} {SepPred: Type}: Type :=
  | data_at: ExprVal -> ctype -> ExprVal -> Separation
  | undef_data_at: ExprVal -> ctype -> Separation
  | other_sep: SepPred -> Separation.
```

- data_at address type val 表示在 address 处有一个类型为 type 的数据，其值为 val；
- undef_data_at address type 表示在 address 处有一个类型为 type 的未定义数据；
- other_sep sepPred 表示一个其他类型的分离逻辑谓词，其参数 sepPred 的类型为 SepPred。

对于 data_at address type val 和 undef_data_at address type 中的第二个参数 type，其类型是之前小节中定义的 ctype，由于我们并没有对程序变量类型进行细分，即所有情况下的 type 均为 Coq 中的 tt 类型，因此在后文中我们将该参数省略。对于 other_sep sepPred，在后文中我们将其简写为谓词 sepPred 的用户定义形式。这两个简写均不会造成歧义。

### 4.1.5 表达式类型与扩展表达式类型的定义

表达式（ExprVal, expression value）的定义有如下几种情况：

- 常量表达式：Ez_val
- 逻辑变量表达式：V_vari
- 基于结构体字段的地址表达式：Vfield_address
- 算术运算表达式：加（+）、减（-）、乘（*）、除（/）、取余（%）、取负数（-）
- 逻辑运算表达式：与（&&）、或（||）、非（!）
- 位运算表达式：按位与（&）、按位或（|）、按位异或（^）、按位取反（~）、左移（<<）、右移（>>）

扩展表达式（ExtExpr, extended expression）是与用户定义的分离逻辑谓词相对应的表达式变体，其作用在于对于两个谓词 Pred id args1 和 Pred id args2，我们从其参数列表中提取出扩展表达式类型的信息，并通过这个信息来判断这两个谓词是否在描述同一块内存。我们在求解过程中经常需要用到这一判断，比如下文中 SimplAddr 的类型就是扩展表达式。

这一设计是为了后续工作的扩展性考虑。对于本项目目前所涉及的数据结构，其扩展表达式类型的定义为 ExprVal，也就是单个表达式类型。但在后续工作中，如果涉及到更为复杂的数据结构，可能需要定义出更为复杂的扩展表达式类型，比如多个表达式组成的元组类型等。

### 4.1.6 与 SepPred 和 ExtExpr 相关的两个类型类

由于不同的数据结构有着不同的分离逻辑谓词和扩展表达式类型，对于同一方法有着不同的实现，因此我们定义了两个类型类，不同的数据结构实现这两个类型类中的方法并定义它们对应的实例。这样一来，我们就可以在后续的证明中使用这两个类型类中的方法，而不用关心具体的数据结构，实现了对数据结构的抽象。其中，与分离逻辑谓词相关的方法在 SepPredClass 中定义，与分离逻辑谓词无关但与扩展表达式类型相关的方法在 ExtExprClass 中定义。这两个类型类的定义如下所示。

```
Class ExtExprClass (C: Ctype) := {
  ExtExpr: Type; (* 扩展表达式类型 *)
  eqb_E: ExtExpr -> ExtExpr -> bool;
  (* 以及一些方法的声明 *)
}.
Class SepPredClass (C: Ctype)
                   (extendExprClass: ExtExprClass C) := {
  (* 一些方法的声明 *)
}.
```

## 4.2 预处理阶段

### 4.2.1 空堆的筛除

我们需要筛除目前能够判定为空堆的空间断言，以免其影响到后续的求解。由于 data_at 和 undef_data_at 的定义决定了它们不可能为空堆，因此我们主要考虑 other_sep 的情况，也就是考虑用户自定义的分离逻辑谓词。

对于不同类型的谓词来说，其判定为空堆的标准也不同。比如对于双链表段 dlseg(x, xp, yp, y)，根据其定义，x 和 y 都是指向链表某个结点的指针，如果这两个指针中至少有一个为 NULL，那么我们就可以认为这个双链表段实际上为空堆。再根据双链表段的定义，我们还可以知道 x == y 和 xp == yp 一定成立。也就是说，

- 如果我们能证明 x == NULL，那么我们就可以认为它为空堆，并且得到 [y == NULL; xp == yp] 这两个 Prop；
- 如果我们能证明 yp == NULL，那么我们就可以认为它为空堆，并且得到 [xp == NULL; x == y] 这两个 Prop。

如果我们的操作对象是 Res，那么得到的 Prop 就可以被当作条件（即下一小节所讨论的纯事实），用于之后的证明；如果我们的操作对象是 Tar，那么得到的 Prop 将会是我们需要证明的目标。

### 4.2.2 纯事实的获取

#### 4.2.2.1 问题与解决思路

纯事实（pure fact）是指我们在求解过程中获得的、不涉及内存状态的命题，可以类比为 VST 证明中的 assert_PROP 操作。它主要用于求解的最后阶段对 Tar.Prop 的证明，以及判断求解过程中是否能够对 Tar.Sep 进行变换。纯事实是通过对 Res.Sep 进行分析得到的，主要从三个角度考虑：

1. 从“一定为空堆”的判定中获取，这一点在上一小节中已经讨论过；
2. 从“一定不为空堆”的判定中获取：这一点主要针对 data_at 和 undef_data_at 这两个空间断言，也就是说对于 data_at addr val 和 undef_data_at addr，我们都可以得到 addr != NULL。
3. 根据分离逻辑规则获取：这一点的基本思路是，如果两个空间断言同时存在，根据分离逻辑规则，它们一定是不相交的，那么我们就可以得到它们的“位置”一定是不同的。

比如,如果 `Res.Sep` 中同时存在 `Data_at x1 v1` 和 `lseg y x2 * Data_at x2 v2`,那么我们就可以得到 `x1 != x2` 和 `x1 != y`。`x1 != x2` 是直接与分离逻辑规则相矛盾的,而 `x1 != y` 则可以通过反证法证明。如果 `x1 == y`,我们对 `lseg y x2` 进行分类讨论:如果 `lseg y x2` 为空堆,那么 `y == x2`,即 `x1 == x2`,矛盾;如果 `lseg y x2` 不为空堆,那么地址 `y` 所对应的内存被描述了两次,与分离逻辑规则相矛盾。

#### 4.2.2.2 实现细节

该算法分为两个步骤。第一步,我们将 `Res.Sep` 分解成多个部分,每个部分的空间断言之间互不相交,一个部分内部的空间断言在逻辑上的相连的。分解空间断言的算法如算法4–1所示。形象地说,该算法的输入是一个“线团”,我们首先找到“线团”中的“线头”,将其“抽出”,这样就得到了一根“线”,也就是一组空间断言,然后对于剩余的线团重复该操作即可,最后输出的结果是多根“线”,也就是几组空间断言。对于每组空间断言,我们要求它的结尾一定是非空的,相当于必须要有一个“端点”,这是为了避免这一组中的每个空间断言都为空堆的情况。

第二步,我们会对得到的各个部分之间两两组合,得到纯事实,如算法4–2所示。该算法在对两组空间断言进行对比前进行了判断,保证这两个空间断言组的末尾不是同一个地址。比如,如果两组空间断言分别为 `lseg x1 y * data_at (&(y->field1)) z1` 和 `lseg x2 y * data_at (&(y->field2)) z2`,我们无法得到 `z1 != z2`。

#### 4.2.2.3 举例说明

对于如下的情况:

```
data_at x1 v1 * Data_at x2 v2 * lseg(y1, y2) * lseg(y2, y3)
* lseg(y3, x1) * listrep(x4) * lseg(y4, x4)
```

在第一步,我们将整个空间断言分为三个部分:

```
1. lseg(y1, y2) * lseg(y2, y3) * lseg(y3, x1) * Data_at x1 v1
2. data_at x2 v2
3. lseg(y4, x4) * listrep(x4)
```

在第二步,我们基于第一步的结果,可以得到如下的纯事实:

**算法 4–1** 分解空间断言的算法

```
Input: Tar.Sep，类型为 list Separation
Output: 分解后的 Tar.Sep，类型为 list (list Separation)
1  Function Divide(SepList1, SepList2):
2      if SepList2 = [] then
3          return []
4      else
5          SepList1′ ← SepList2[0] :: SepList1
6          if SepList2[0] 是“一连串”空间断言的第一个 then
7              if 对于 SepList2[0] 是否为空堆的判别是不依赖于其它空间断言的 then
                   // 比如 data_at、listrep 等
8                  return [SepList2[0]] :: Divide(SepList1, SepList2[1:])
9              else
                   // 数据结构段相关的谓词，比如 lseg、dlseg 等
10                 CheckExprList ← SepList2[0] 的“下一个”空间断言的地址
11                 SepListToCheck ← SepList2[1 :] ++SepList1
12                 if 根据 CheckExprList，在 SepListToCheck 中找到了“完整的一串空间断
                     言” then
13                     SepList3 ← 这一串空间断言
14                     return [SepList2[0] :: SepList3] ::
                         Divide([], SepListToCheck - SepList3)
15                 else
16                     return Divide(SepList1', SepList2[1:])
17         else
18             return Divide(SepList1', SepList2[1:])
19 Function DivideSep:
20     return Divide(nil, Tar.Sep)
```

1. 由 1 和 2 可以得到：`x1 != x2 && y3 != x2 && y2 != x2 && y1 != x2;`
2. 由 1 和 3 可以得到：`x1 != x4 && x1 != y4 && y3 != x4 && y3 != y4 && y2 != x4 && y2 != y4 && y1 != x4 && y1 != y4;`
3. 由 2 和 3 可以得到：`x2 != x4 && x2 != y4`。

**算法 4–2** 由分组的空间断言获取纯事实的算法

**Input:** 分解后的 $Tar.Sep$，类型为 `list (list Separation)`
**Output:** 纯事实 $PureFact$，类型为 `list Proposition`

```
Function GenPureFacts(GroupedSepList):
    if GroupedSepList = [] then
        return []
    else
        Group1 ← GroupedSepList[0]
        for Group2 in GroupedSepList[1:] do
            if Group1 和 Group2 的末尾不是同一个地址 then
                for Sep1 in Group1 do
                    for Sep2 in Group2 do
                        将 "Sep1的地址 ≠ Sep2的地址" 加入到 PureFact 中
        return PureFact + +GenPureFacts(GroupedSepList[1:])
```

## 4.3 求解器的设计

### 4.3.1 Prop 求解器

本项目所涉及的命题（Proposition）主要在一阶逻辑的基础上，增加了数值大小比较的运算符。所涉及的所有运算符如表4–2所示。

**表 4–2 命题定义所涉及的运算符**

| 运算符类型 | 运算符列表 |
|---|---|
| 一元逻辑运算符 | ¬ |
| 二元逻辑运算符 | ∨ ∧ → ↔ |
| 一元比较运算符 | *isptr* *is_pointer_or_null* |
| 二元比较运算符 | ≤ ≥ < > = |

Prop 求解器主要应用于求解后期使用 `Res.Prop` 证明 `Tar.Prop`，以及求解过程中对于一些临时产生的 Prop 的证明。比如我们对于单链表段 `lseg x y` 进行操作之前，需要先利用 Prop 求解器和之前获得的纯事实来证明它不是空堆，也就是证明 `x != y`。

Prop 求解器的输入是两个 Prop 列表，每个 Prop 列表表示多个 Prop 的合取，这里我们以 `ResP` 和 `TarP` 表示一般情况下的输入，与 `Res.Prop` 和 `Tar.Prop` 相

区别。其返回值是一个 `option` 类型：如果求解成功，则返回无法被证明的 Prop 列表；如果求解失败则返回 `None`。Prop 求解器的求解过程分为三个步骤：

1. 对 `ResP` 和 `TarP` 进行预处理。
   (a) 如果 Prop 列表中存在永假式，那么整个 Prop 列表的值一定是 False；
   (b) 使用德摩根定律，将逻辑运算符 Not 进行下推；
   (c) 将逻辑运算符 And 连接的 Prop 拆分为两个 Prop；
   (d) 将针对字段地址相等的描述化简为对结构体地址相等的描述。
2. 判断是否能够从 `ResP` 证明出 False，如果可以的话，那么 `False |- TarP` 一定为真，无论 `TarP` 是什么内容。
3. 如果无法推出，则尝试使用 `ResP` 证明 `TarP`。

### 4.3.2 Local 求解器

Local 求解器主要应用与 Local 相关存在变量的实例化，以及 Local 的“消去”。由于我们在证明过程中不会对 Local 进行修改，所以这两个步骤实际上可以同时完成，如算法4–3所示。

**算法 4–3** Local 求解器

**Input:** *Res.Local* 和 *Tar.Local*，类型为 `list Local`；Tar.EX，类型为 `list ident`
**Output:** 消去剩下的 *Res.Local* 和 *Tar.Local*，类型为 `list Local`；*InstMap*，类型为 `list (ident * ExprVal)`；*TarProp*，类型为 `list Proposition`

```
1  Function LocalSolver(Res.Local, Tar.Local, Tar.EX):
2      for reslocal in Res.Local do
3          for tarlocal in Tar.Local do
4              if reslocal的程序变量 与 tarlocal的程序变量 相同 then
5                  从 Res.Local 中删除 reslocal
6                  从 Tar.Local 中删除 tarlocal
7                  if tarlocal的逻辑变量 在 Tar.EX 中 then
8                      将 reslocal的逻辑变量 与 tarlocal的逻辑表达式 的对应关系加入
                        InstMap
9                      将 tarlocal的逻辑变量 从 Tar.EX 中删除
10                 else
11                     将 “reslocal的逻辑变量 = tarlocal的逻辑变量” 加入到 TarProp
12     return Res.Local, Tar.Local, InstMap, TarProp
```

### 4.3.3 Sep 求解器

Sep 求解器主要应用与 Sep 相关存在变量的实例化，以及 Sep 的“消去”。由于我们会在证明过程中可能对 Sep 进行修改，可能产生新的存在变量，所以这两个步骤需要分开进行，即先在预处理阶段进行 Sep 的实例化，然后在求解阶段进行 Sep 的消去，同时进行新产生的存在变量实例化。整体如算法4–4所示。

由于我们会对 `Tar.Sep` 中的用户自定义谓词进行拆分（在4.4.1小节中介绍），使其转化为适合消去的形式，所以在这里我们不需要对用户自定义谓词进行复杂的处理，只需要对于两边完全相同的谓词进行消去即可。

## 4.4 求解过程的特殊操作

### 4.4.1 基于字段的 `Tar.Sep` 拆分

#### 4.4.1.1 问题与相关定义

一般来说，`Res.Sep` 与 `Tar.Sep` 并不会完全相同，需要我们对 `Tar.Sep` 进行变换之后再将它们输入到 Sep 求解器，才能够得到最终的结果。这里的变换主要是对用户自定义谓词的拆分，比如将一个单链表拆分为其头结点加上剩余部分。由于拆分得到的结果是与用户所定义的结构体中的“字段”（field，或称为“数据成员”、“成员变量”）相关的，所以我们称这种拆分为“基于字段的拆分”。

对于大多数数据结构来说，我们可以将其字段分为两类，即数据字段（data field）和链接字段（link field）。当然，数据结构不同时，其链接字段的个数和具体链接方式也不同；逻辑上的数据结构相同时，其数据字段的具体情况也可能不同。因此，对于所有的数据结构，我们定义了类型类 `DataField`:

```
Class DataField (C: Ctype) := {
  data_field: ExprVal -> ident -> list ident * list
Separation * list Proposition;
  is_data_field: Separation -> option ExprVal;
}.
```

对于每一个特定的数据结构，我们定义了类型类 `LinkField`:

```
(* 单链表 *)
Class Link_Field (C: Ctype) := {
```

```
link_field: ExprVal -> ExprVal -> Separation;
is_link_field: Separation -> option (ExprVal * ExprVal);
}.
(* 双链表 *)
Class LinkField (C: Ctype) := {
  next_field: ExprVal -> ExprVal -> Separation;
  is_next_field: Separation -> option (ExprVal * ExprVal);
  prev_field: ExprVal -> ExprVal -> Separation;
  is_prev_field: Separation -> option (ExprVal * ExprVal);
}.
(* 二叉树 *)
Class Link_Field (C: Ctype) := {
  lch_link_field: ExprVal -> ExprVal -> Separation;
  is_lch_link_field: Separation -> option (ExprVal *
ExprVal);
  rch_link_field: ExprVal -> ExprVal -> Separation;
  is_rch_link_field: Separation -> option (ExprVal *
ExprVal);
  par_link_field: ExprVal -> ExprVal -> Separation;
  is_par_link_field: Separation -> option (ExprVal *
ExprVal);
}.
```

可以看到，无论是 DataField 还是各种情况下的 LinkField，都定义了两个方法：some_field 和 is_some_field。其中，some_field 相当于字段的构造，其输入是一个或多个 ExprVal 类型，输出是基于这些信息构造出的空间断言；is_some_field 相当于字段的检验，其输入是一个空间断言，如果该空间断言符合该字段的定义，则返回该字段所在结构体对象的地址和该字段的值，否则返回 None。

对于 DataField 的字段构造方法 data_field，由于我们可能产生新的存在变量，因此会传入一个 ident 类型参数，表示目前最大的变量标识符，从而防止变量标识符的冲突。data_field 除了输出构造的空间断言之外，还会输出新产生的存在变量列表，以及该数据字段所遵循的 Prop。

这两个定义将成为我们对 `ExtExprClass` 和 `SepPredClass` 进行实例化得到新的类型时引入的额外参数。

#### 4.4.1.2 实现细节

拆分的思路是：对于 `Tar.Sep` 中的每一个自定义谓词，去 `Res.Sep` 中寻找同一地址的 `data_at` 空间断言，并判断该断言是否符合字段的定义。如果符合某种字段的定义，则按照定义拆分对应的自定义谓词。这里我们以双链表为例来介绍自定义谓词的拆分子函数，如算法4–5所示。

### 4.4.2 SimplAddr 的获取与使用

#### 4.4.2.1 问题与解决思路

可化简的地址（Simplifiable Address, SimplAddr）主要应用于判断数据结构段的相关谓词是否为空堆，比如单链表段、双链表段、部分树（即半树的双链表段）等。以单链表段 `lseg x y` 为例，当 `x == y` 时，我们并不能认为 `lseg x x == emp`，因为有可能 `lseg x x == exists z, lseg x z * data_at z x`，也就是环状链表。

我们根据分离逻辑的规则来区分这两种情况：因为分离合取是将内存分为“互不相交”的几个部分，所以一个地址所表示的内存空间不能被多次描述。依然是这个例子，如果空间断言为 `lseg x x * listrep x`，由于 `listrep x` 的存在（要求 `x != 0`），对于地址 `x` 已经有一个空间断言对其进行描述，那么 `lseg x x` 如果不为空堆的话，就相当于对地址 `x` 再次进行了描述，那么整个空间断言就违反了分离逻辑的规则，也就等价于 `False`。因此我们可以认为 `lseg x x == emp`。

对于空间断言不能违反分离逻辑规则的合理性论证：如果 `Res.Sep` 为 False，因为 False 可以推导出任意断言，所以我们忽略这种情况；如果 `Tar.Sep` 为 False，那么相当于我们需要证明的断言是 False，这种情况是不合理的，所以我们忽略这种情况。

#### 4.4.2.2 实现细节

对于 `Tar.Sep` 的 SimplAddr 需要在 `Tar.Sep` 拆分后、Sep 消去前获取，这是因为此时 Sep 的数量达到最大，也就说内存被划分为最多的部分，此时获得的 SimplAddr 能够尽可能地覆盖到所有的可能性。同理，对于 `Res.Sep` 的 SimplAddr 也需要 Sep 消去前获取。在调用 Sep 求解器进行 Sep 消去之后，如果 `Res.Sep` 或 `Tar.Sep` 中

还存在 Sep 没有被消去，则对于每一个剩余的 Sep，判断其扩展表达式类型的信息是否在之前对于二者分别获取的 SimplAddr 之中。如果在的话，则可以认为该 Sep 为 emp，从而起到化简的效果。

SimplAddr 获取算法与4.2.2小节中所述的算法类似，如算法4–6所示。

### 4.4.3 模块的组合

在4.4.1小节中我们提到，可以通过定义不同的链接字段来构造出不同的数据结构，也可以通过定义不同的数据字段来构造出同一种数据结构的不同情况。当我们的程序中定义了多种谓词时，就需要对这些谓词进行组合，从而实现对于多种数据结构的支持。

我们定义了空谓词 `EmptySep`,它对应的 `ExtExprClass` 和 `SepPredClass` 中的方法都会返回默认值（一般为空值）。然后我们基于 Coq 中的 `sum` 类型，将谓词类型进行“相加”。`sum` 的定义如下所示。

```
Inductive sum (A B: Type) : Type :=
  | inl: A -> sum A B
  | inr: B -> sum A B.
```

比如，如果我们在 C 语言中定义了三种单链表，将其对应的空间断言定义记为 `ListSep1`、`ListSep2` 和 `ListSep3`,则我们可以使用 `empty + listrep + listrep + listrep` 来表示用户自定义谓词的类型，然后使用 `inl (inr L1)` 表示 `ListSep1` 类型的谓词 `L1`，`inl (inr (inr L2))` 表示 `ListSep2` 类型的谓词 `L2`，`inr (inr (inr L3))` 表示 `ListSep3` 类型的谓词 `L3`，从而实现了多种谓词的共存和组合。

## 4.5 证明过程的输出

为了验证求解器的求解是否正确，我们在求解过程中保存了所使用到的证明规则，并且在求解结果中输出。

### 4.5.1 证明规则的定义

我们对于 Prop、Local 和 Sep 定义了不同的证明规则。

对于 Prop，我们将证明规则分为了三种类型：assume、derive 和 contra。assume 表示我们所假设的 Prop，也就是条件中的 Prop；derive 表示该 Prop 是由其他 Prop 通过某种规则推导出来的；contra 表示在目前的条件中存在两个互相矛盾的 Prop，即 `Res.Prop` 为 False 的情况。

对于 derive，我们还定义了一个 RuleSymbols 类型，表示推导所使用的规则。这里我们参考了 cvc5[28]中定义的证明规则和 Alethe[33]证明形式。由于我们目前只需要证明相等和不相等两种命题，因此仅需要定义一些基础的规则加上相等和不相等的推导规则即可。

```
Inductive RuleSymbols :=
  | refl
  | resolution
  | and
  | or
  | equiv1
  | equiv2
  | not_equiv1
  | not_equiv2.
Inductive PropRule :=
  | assume: ident -> Proposition -> PropRule
  | derive: ident -> list Proposition -> RuleSymbols ->
list ident -> PropRule
  | contra: Proposition -> Proposition -> PropRule.
```

对于 Local，我们定义了以下两个证明规则，分别对应于逻辑变量是存在变量和逻辑变量是自由变量的情况。对于前者，我们会将存在变量实例化，对于后者，我们会生成待证明的 Prop。

```
Inductive LocalRule: Type :=
  | elim_with_inst_ex: ident -> ExprVal -> LocalRule
  | elim_with_prop: ident -> Proposition -> LocalRule.
```

对于 Sep，由于我们会将变量实例化和空间断言消去分开进行，因此定义 inst_ex 规则用于实例化变量，而不对空间断言进行修改。除此之外，我们定义了空堆筛除、SimplAddr 获取和使用、纯事实的获取等操作的相关规则。我们还定义了 other_sep，用于为子模块中的证明规则提供接口。

```
Inductive SepRule {rule: Type} {ctype: Type} {ExtExpr: Type}
: Type :=
  | empty_sep: Separation -> list Proposition -> SepRule
  | inst_ex: ident -> ExprVal -> SepRule
  | elim_with_prop: Separation -> Proposition -> SepRule
  | simpl_to_emp_addr: list Separation -> ExtExpr ->SepRule
  | simpl_to_emp: Separation -> ExtExpr -> list Proposition
-> SepRule
  | get_pure_fact_emp: Separation -> list Proposition ->
SepRule
  | get_pure_fact_no_emp: Separation -> list Proposition ->
SepRule
  | get_pure_fact_division: list Separation -> list
Separation -> list Proposition -> SepRule
  | other_rule: rule -> SepRule.
```

### 4.5.2 证明规则的筛选

在求解过程中我们会获取很多信息，比如纯事实和 SimplAddr 等，但这些信息并非全都是有用的。因此为了保持输出证明规则的简洁性，我们只需要保留在证明过程中使用到了的信息即可。我们筛选证明规则的方法如下：

- 对于纯事实获取的相关证明规则：根据 Prop 求解器输出的 Prop 证明过程，其中的 assume 规则中所使用到的 Prop 即为我们所需要的纯事实。我们只需要保留获取这一部分纯事实的证明规则即可。
- 对于 SimplAddr 获取的相关证明规则：根据 SimplAddr 使用的相关规则，我们只需要保留获取“使用了的 SimplAddr”的规则即可。

### 4.5.3 证明规则转换到 VST 证明代码

VST-IDE 将所有变量都当作可寻址的变量（addressable variable），Local 所定义的逻辑变量表示程序变量的地址，对于程序变量的赋值都看作该地址所表示的内存的赋值，逻辑变量本身的值并没有改变。但在 VST 中并非这样处理。为了使得证明规则与 VST 证明代码一致，我们并非根据 VST-IDE 的符号执行结果来打印证明规则，而是根据 VST 中的证明目标来打印证明规则。我们会在 VST-IDE 中手动定义出蕴含关系两边的断言，然后打印出对应的 Coq 代码，包括以 `assert` 来引入该证明关系（记为 $\gamma$），以及 $\gamma$ 的证明。然后，我们会手动地调用 $\gamma$，从而完成证明。

在求解过程中，我们会将证明规则保存在一个列表中。在求解结束后，我们会将这个列表转换为 VST 证明代码。我们将证明规则转换为 VST 证明代码的方法如下：

- 对于 Prop 的证明规则：我们首先找到其最终证明的 Prop，并根据它打印出 `assert_PROP`，然后根据其推导过程中使用到的证明规则，打印出相应的证明代码；
- 对于 Local 的证明规则：首先我们打印出存在变量实例化的相关证明代码，然后打印 `go_lower` 来完成 Local 的消去。
- 对于 Sep 的证明规则：首先我们打印出存在变量实例化的相关证明代码，然后对于拆分和化简操作，我们直接调用在 VST 环境下定义好的引理，最后我们打印 `entailer` 来完成对于 Sep 的消去。

## 4.6 本章小结

本章详细介绍了蕴含关系求解器的实现。首先介绍了相关类型的定义，然后介绍了预处理算法、求解器算法、拆分算法、化简算法和模块组合等。最后介绍了证明规则的定义、输出以及转换到 VST 证明代码的方法。

**算法 4–4** Sep 求解器

**Input:** *Res.Sep* 和 *Tar.Sep*，类型为 `list Separation`；*Tar.EX*，类型为 `list ident`

**Output:** 消去剩下的 *Res.Sep* 和 *Tar.Sep*，类型为 `list Sep`；*InstMap*，类型为 `list (ident * ExprVal)`；*TarProp*，类型为 `list Proposition`

```
1  Function SepInstantiation(Res.Sep, Tar.Sep, Tar.EX):
2      for resSep in Res.Sep do
3          for tarSep in Tar.Sep do
4              if resSep = data_at addr1 val1 且 tarSep = data_at addr2 val2 then
5                  if addr1 = addr2 且 val1所对应的逻辑变量 在 Tar.EX 中 then
6                      将 (val1所对应的逻辑变量, val2) 加入 InstMap
7                      将 val1所对应的逻辑变量 从 Tar.EX 中删除
8              else if resSep 和 tarSep 都是用户自定义谓词 then
9                  调用用户自定义谓词相关子函数进行处理
10     return InstMap
11 Function SepSolver(Res.Sep, Tar.Sep, Tar.EX):
12     for resSep in Res.Sep do
13         for tarSep in Tar.Sep do
14             if resSep = data_at addr1 val1 且 tarSep = data_at addr2 val2 then
15                 if addr1 = addr2 then
16                     从 Res.Sep 中删除 resSep
17                     从 Tar.Sep 中删除 tarSep
18                     if val1所对应的逻辑变量 在 Tar.EX 中 then
19                         将 (val1所对应的逻辑变量, val2) 加入 InstMap
20                         将 val1所对应的逻辑变量 从 Tar.EX 中删除
21                     else
22                         将 val1 = val2 加入到 TarProp 中
23             else if resSep 和 tarSep 都是用户自定义谓词 then
24                 调用用户自定义谓词相关子函数进行处理
25     return Res.Sep, Tar.Sep, InstMap, TarProp
```

**算法 4–5** 自定义谓词的拆分子函数（以双链表为例）

**Input:** $resSep$，类型为 `Separation`；$tarSep$，类型为 `SepPred`；$Tar.EX$，类型为 `list ident`；target 中变量的最大编号 $max\_id$，类型为 `ident`

**Output:** 新产生的存在变量 $tarEx'$，类型为 `list ident`；拆分后的空间断言 $tarSep'$，类型为 `Separation`；新产生的待证命题 $TarProp$，类型为 `list Proposition`

```
1  Function FieldSplit(resSep, tarSep, Tar.EX):
2      if tarSep = dlistrep(x, x_prev) 且 x所对应的逻辑变量 不在 Tar.EX 中 then
3          if resSep 是以 x 为地址的 DataField 或 NextField 或 PrevField then
4              将 tarSep 拆成头结点 + 剩余部分
5          else if resSep 是用户自定义谓词 dlseg(x', x'_prev, y, y_prev) 且 x = x' then
6              将 tarSep 拆成开头的双链表段 + 剩余部分
7      else if tarSep = dlseg(x, x_prev, y, y_prev) 且 x和y_prev所对应的逻辑变量 都不在
         Tar.EX 中 then
8          if x ≠ y then
               // 保证 tarSep ≠ emp
9              if resSep 是以 x 为地址的 DataField 或 NextField 或 PrevField then
10                 if x = addr then
11                     将 tarSep 拆成头结点 + 剩余部分
12                 else if y_prev = addr then
13                     将 tarSep 拆成尾结点 + 剩余部分
14             else if resSep 是用户自定义谓词 dlseg(x', x'_prev, y', y'_prev) then
15                 if x = x' then
16                     将 tarSep 拆成开头的双链表段 + 剩余部分
17                 else if y_prev = y'_prev then
18                     将 tarSep 拆成结尾的双链表段 + 剩余部分
```

**算法 4–6** 获取 SimplAddr 的算法

```
Input: Tar.Sep，类型为 list Separation
Output: SimplAddr 列表，类型为 list ExtExpr
1  Function Get(SepList1, SepList2):
2      if SepList2 = [] then
3          return []
4      else
5          SepList1′ ← SepList2[0] :: SepList1
6          if SepList2[0] 是 data_at then
7              调用 data_at 相关子函数进行处理
8          else if SepList2[0] 是 other_sep then
9              if 对于 SepList2[0] 是否为空堆的判别是不依赖于其它空间断言的 then
                   // 比如 listrep、dlistrep 等
10                 return [SepList2[0]的地址] :: Get(SepList1, SepList2[1:])
11             else
                   // 数据结构段相关的谓词，比如 lseg、dlseg 等
12                 CheckExprList ← SepList2[0]的“下一个”空间断言的地址
13                 SepListToCheck ← SepList2[1 :] ++SepList1
14                 if 根据 CheckExprList，在 SepListToCheck 中找到了“完整的一串空
                     间断言” then
15                     SepList3 ← 这一串空间断言
16                     AddrList ← 这一串空间断言所对应的地址
17                     return SepList2[0]的地址 ::
                         AddrList ++Get(nil, SepListToCheck - SepList3)
18                 else
19                     return Get(SepList1', SepList2[1:])
20         else
21             return Get(SepList1', SepList2[1:])
22 Function GetSimplAddr:
23     return Get(nil, Tar.Sep)
```

# 第五章 测试结果

## 5.1 测试样例

目前我们能够验证的程序如表5–1所示。

**表 5–1 测试样例**

| 数据结构类型 | 操作 | 验证状态 |
|---|---|---|
| 单链表 | 反转 | 成功 |
| 单链表 | 遍历 | 成功 |
| 单链表 | 连接两个链表 | 成功 |
| 双链表 | 反转 | 成功 |
| 双链表 | 连接两个链表 | 成功 |
| 双链表 | 直接删除结点 | 成功 |
| 双链表 | 遍历删除结点 | 成功 |
| 二叉树 | 插入结点 | 调试中 |
| 二叉树 | 删除结点 | 调试中 |
| 二叉树 | 查找结点 | 调试中 |
| 其它 | 交换两个变量的值 | 成功 |

## 5.2 一个典型的例子

接下来我们通过对连接两个单链表的程序进行验证来介绍本项目的流程，重点关注求解器的求解流程。待验证的 C 程序如下所示，包括了用户自定义谓词的定义，以及函数前后的规约条件和循环不变量。

```
1 struct list {
2     int head;
3     struct list *tail;
4 };
5 /*@ Let listrep(l) = l == 0 && emp || listrep(l->tail) */
6 /*@ Let lseg(x, y) = x == y && emp || lseg(x->tail, y) */
7
8 struct list * append(struct list * x, struct list * y)
```

```
9  /*@ Require listrep(x) * listrep(y)
10     Ensure  listrep(__return)
11  */
12 {
13     struct list *t, *u;
14     if (x == 0) {
15         return y;
16     } else {
17         t = x;
18         u = t->tail;
19         /*@ u == t->tail &&
20             listrep(y) *
21             listrep(u) *
22             lseg(x, t)
23          */
24         while (u) {
25             t = u;
26             u = t->tail;
27         }
28         t->tail = y;
29         return x;
30     }
31 }
```

我们的编译前端会对源代码进行编译，并对注释中定义的谓词进行处理，判断出这里定义了单链表相关的两个谓词，并且 DataField 为空堆（也就是不考虑数据字段），LinkField 为指向下一个结点的指针。同时，编译前端也会对注释中定义的规约和断言进行处理，将其转化为后端中的断言形式。

然后符号执行部分会一行一行地对程序进行符号执行，不断地从函数的前置条件开始更新程序状态。在整个程序中，有如下几个位置会调用蕴含关系求解器：

- 第 15 行 Return，此时的程序状态应该要能够蕴含函数的后置条件；

- 第 19 行循环不变量，此时的程序状态（进入循环前）应该要能够蕴含循环不变量；
- 第 26 行循环体结束，此时的程序状态（经过循环体后）应该也要能够蕴含循环不变量；
- 第 29 行 `Return`，此时的程序状态应该要能够蕴含函数的后置条件。

我们以第 19 行，即“循环前的程序状态蕴含循环不变量”为例，来介绍求解器的求解流程。存在变量中的数字代表变量的编号，同时为了表述的直观，我们采用了简写的方式，将 `data_at addr val` 简写为 `addr |-> val`，将 `Pred` 改写为更加可读的谓词形式等。我们需要证明以下蕴含式成立：

```
1  EX 23 22 19 18 24,
2  PROP[V_vari 14 != Ez_val 0]
3  LOCAL[Temp u (V_vari 23); Temp t (V_vari 22);
4        Temp x (V_vari 19); Temp y (V_vari 18) ]
5  SEP[V_vari 23 |-> V_vari 24; V_vari 22 |-> V_vari 14;
6      V_vari 19 |-> V_vari 14; V_vari 18 |-> V_vari 15;
7      &((V_vari 14)->tail) |-> V_vari 24;
8      listrep(V_vari 15); listrep(V_vari 24)]
9  |=
10 EX 28 29 30 31 32 33 34 35 36,
11 PROP[V_vari 29 == V_vari 32]
12 LOCAL[Temp u (V_vari 28); Temp t (V_vari 30);
13       Temp y (V_vari 33); Temp x (V_vari 35)]
14 SEP[V_vari 28 |-> V_vari 29; V_vari 30 |-> V_vari 31;
15     V_vari 33 |-> V_vari 34; V_vari 35 |-> V_vari 36;
16     &((V_vari 31)->tail) |-> V_vari 32;
17     listrep(V_vari 34); listrep(V_vari 29);
18     lseg(V_vari 36, V_vari 31) ]
```

求解过程如下：

1. 预处理：
   (a) 筛除 `Res.Sep` 中的空堆：根据已有的 Prop 条件，并不能判断出 `Res.Sep` 中任何空间断言为空堆，因此这一步没有任何效果；

(b) 从新的 Res.Sep 中获得纯事实：

- 根据 data_at 非空，可以获得：

```
[ V_vari 23 != Ez_val 0; V_vari 22 != Ez_val 0;
V_vari 19 != Ez_val 0; V_vari 18 != Ez_val 0;
V_vari 14 != Ez_val 0 ]
```

- 根据空间断言不相交，可以获得：

```
[ V_vari 23 != V_vari 22; V_vari 23 != V_vari 19;
V_vari 23 != V_vari 18; V_vari 23 != V_vari 14;
V_vari 23 != V_vari 15; V_vari 23 != V_vari 24;
V_vari 22 != V_vari 19; V_vari 22 != V_vari 18;
V_vari 22 != V_vari 14; V_vari 22 != V_vari 15;
V_vari 22 != V_vari 24; V_vari 19 != V_vari 18;
V_vari 19 != V_vari 14; V_vari 19 != V_vari 15;
V_vari 19 != V_vari 24; V_vari 18 != V_vari 14;
V_vari 18 != V_vari 15; V_vari 18 != V_vari 24;
V_vari 14 != V_vari 15; V_vari 14 != V_vari 24;
V_vari 15 != V_vari 24 ]
```

(c) Local 相关存在变量的实例化与 Local 的消去：Local 全部消去，并且获得实例化映射：

```
[ (28, V_vari 23); (30, V_vari 22);
(33, V_vari 18); (35, V_vari 19) ]
```

然后根据该映射，对 Tar.Sep 进行实例化。目前需要证明的蕴含式变为：

```
1 EX 23 22 19 18 24,
2 PROP[V_vari 14 != Ez_val 0 :: pure_facts]
3 LOCAL[]
4 SEP[V_vari 23 |-> V_vari 24; V_vari 22 |-> V_vari 14;
5 V_vari 19 |-> V_vari 14; V_vari 18 |-> V_vari 15;
6 &((V_vari 14)->tail) |-> V_vari 24;
7 listrep(V_vari 15); listrep(V_vari 24)]
8 |=
9 EX 29 31 32 34 36,
```

```
PROP[V_vari 29 == V_vari 32]
LOCAL[]
SEP[V_vari 23 |-> V_vari 29; V_vari 22 |-> V_vari 31;
V_vari 18 |-> V_vari 34; V_vari 19 |-> V_vari 36;
&((V_vari 31)->tail) |-> V_vari 32;
listrep(V_vari 34); listrep(V_vari 29);
lseg(V_vari 36, V_vari 31) ]
```

(d) 筛除 Tar.Sep 中的空堆：根据已有的 Prop 条件，同样并不能判断出 Tar.Sep 中任何空间断言为空堆，因此这一步也没有任何效果；

(e) 根据 Res.Sep 和 Tar.Sep 的内容，对 Tar.Exist 进行实例化（基于 Sep 求解器）：获得实例化映射：

```
[ (29, V_vari 24); (31, V_vari 14);
(34, V_vari 15); (36, V_vari 14); ]
```

然后根据该映射，对 Tar.Sep 进行实例化。目前需要证明的蕴含式变为：

```
EX 23 22 19 18 24,
PROP[V_vari 14 != Ez_val 0 :: pure_facts]
LOCAL[]
SEP[V_vari 23 |-> V_vari 24; V_vari 22 |-> V_vari 14;
V_vari 19 |-> V_vari 14; V_vari 18 |-> V_vari 15;
&((V_vari 14)->tail) |-> V_vari 24;
listrep(V_vari 15); listrep(V_vari 24)]
|=
EX 32,
PROP[V_vari 24 == V_vari 32]
LOCAL[]
SEP[V_vari 23 |-> V_vari 24; V_vari 22 |-> V_vari 14;
V_vari 18 |-> V_vari 15; V_vari 19 |-> V_vari 14;
&((V_vari 14)->tail) |-> V_vari 32;
listrep(V_vari 15); listrep(V_vari 24);
lseg(V_vari 14, V_vari 14) ]
```

2. 基于字段(Field)对 `Tar.Sep` 进行拆分：没有在 `Res.Sep` 中找到以 `V_vari 15` 或 `V_vari 24` 为地址的 `data_at`，因此不会对 `Tar.Sep` 进行拆分，这一步没有任何效果；
3. 基于 `Tar.Sep` 获得可化简的地址（Simplifiable Address, SimplAddr）：获得 SimplAddr 如下

   `[ V_vari 23; V_vari 22; V_vari 18; V_vari 19; V_vari 14; V_vari 15; V_vari 24 ]`
4. Sep 相关存在变量的实例化与 Sep 的消去：获得实例化映射 `[(32, V_vari 24)]`，并根据该映射对 `Tar.Sep` 进行实例化。目前需要证明的蕴含式变为：

```
EX 23 22 19 18 24,
PROP[V_vari 14 != Ez_val 0 :: pure_facts]
LOCAL[]
SEP[]
|=
PROP[V_vari 24 == V_vari 24]
LOCAL[]
SEP[lseg(V_vari 14, V_vari 14) ]
```

5. 根据之前获得的SimplAddr,对 `Tar.Sep` 进行化简:对于 `lseg(V_vari 14, V_vari 14)`，由于 `V_vari 14` 在 SimplAddr 中，我们可以将其化简为空堆。至此，`Tar.Sep` 已经全部消去。
6. 对 `Tar.Prop` 和求解过程中产生的新的待证明命题进行证明：待证命题为 `V_vari 24 == V_vari 24`，显然成立。至此，整个证明过程结束。

最后输出证明规则。由于我们在后续的证明中并没有使用到任何纯事实，所以获取纯事实相关的证明规则都会被省略。这里我们用 `PRule`、`LRule` 和 `SRule` 分别表示 Prop 规则、Local 规则和 Sep 规则。

```
[LRule (elim_with_inst_ex 28 (V_vari 23));
LRule (elim_with_inst_ex 30 (V_vari 22));
LRule (elim_with_inst_ex 33 (V_vari 18));
LRule (elim_with_inst_ex 35 (V_vari 19));
SRule (inst_ex 29 (V_vari 24));
```

```
SRule (inst_ex 31 (V_vari 14));
SRule (inst_ex 34 (V_vari 15));
SRule (inst_ex 36 (V_vari 14));
SRule (simpl_to_emp_addr
      [&((V_vari 14)->tail) |-> V_vari 32] (V_vari 14));
SRule (elim_with_prop (V_vari 23 |-> V_vari 24)
      [V_vari 24 == V_vari 24]);
SRule (elim_with_prop (V_vari 22 |-> V_vari 14)
      [V_vari 14 == V_vari 14]);
SRule (inst_ex 32 (V_vari 24));
SRule (elim_with_prop (&((V_vari 14)->tail) |-> V_vari 24)
      [V_vari 24 == V_vari 24]);
SRule (elim_with_prop (V_vari 18 |-> V_vari 15)
      [V_vari 15 == V_vari 15]);
SRule (elim_with_prop (V_vari 19 |-> V_vari 14)
      [V_vari 14 == V_vari 14]);
SRule (elim_with_prop (listrep(V_vari 24)) []);
SRule (elim_with_prop (listrep(V_vari 15)) []);
SRule (simpl_to_emp lseg(V_vari 14, V_vari 14) (V_vari 14)
      []);
PRule ([derive 1 [V_vari 24 == V_vari 24] refl []]);
PRule ([derive 1 [V_vari 14 == V_vari 14] refl []]);
PRule ([derive 1 [V_vari 24 == V_vari 24] refl []]);
PRule ([derive 1 [V_vari 15 == V_vari 15] refl []]);
PRule ([derive 1 [V_vari 14 == V_vari 14] refl []]);
]
```

## 5.3 本章小结

本章介绍了 VST-IDE 目前能够验证的程序，然后使用了一个例子对求解过程进行了详细介绍。

# 第六章 全文总结与展望

## 6.1 总结

本文提出了一种交互式程序验证工具 VST-IDE，它允许用户以注释的形式进行谓词的自定义和断言的编写，从而降低程序验证的门槛。本文首先介绍了 VST-IDE 的整体架构，然后对蕴含关系求解的部分进行了详细介绍，介绍了相关定义、预处理算法、求解器算法、拆分算法、化简算法和模块组合等。最后本文以验证连接两个单链表的程序作为例子，详细地描述了求解的过程。

## 6.2 展望

在未来，本项目还有以下改进方向：

1. 进一步修改和完善算法，完善对二叉树相关的程序的验证，实现对更加复杂的数据结构和算法的验证；
2. 在 Prop 求解器部分接入功能更强大的 SMT 求解器，使其能够求解更加复杂的 Prop；
3. 采用 C 语言进行求解器的实现，从而提高求解器的运行效率。

# 参考文献

[1] CCF 形式化方法专业委员会. 形式化方法的研究进展与趋势[J]. 2017~2018 中国计算机科学技术发展报告, 2018: 1-68.

[2] 王戟, 詹乃军, 冯新宇, 等. 形式化方法概貌[J]. Journal of Software, 2019, 30(1).

[3] REYNOLDS J C. Separation logic: A logic for shared mutable data structures[C]//Proceedings 17th Annual IEEE Symposium on Logic in Computer Science. 2002: 55-74.

[4] 秦胜潮, 许智武, 明仲. 基于分离逻辑的程序验证研究综述[J]. 软件学报, 2017, 28(8): 2010-2025.

[5] 王捍贫, 张博闻. 分离逻辑的技术基础与研究现状[J]. 广州大学学报: 自然科学版, 2019, 18(2): 1-9.

[6] BERDINE J, CALCAGNO C, O'HEARN P W. Symbolic execution with separation logic[C]//Programming Languages and Systems: Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005. Proceedings 3. 2005: 52-68.

[7] BERDINE J, CALCAGNO C, O'HEARN P W. Smallfoot: Modular automatic assertion checking with separation logic[C]//Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures 4. 2006: 115-137.

[8] DISTEFANO D, O'HEARN P W, YANG H. A local shape analysis based on separation logic[C]//Tools and Algorithms for the Construction and Analysis of Systems: 12th International Conference, TACAS 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25-April 2, 2006. Proceedings 12. 2006: 287-302.

[9] YANG H, LEE O, BERDINE J, et al. Scalable shape analysis for systems code[C]//Computer Aided Verification: 20th International Conference, CAV 2008 Princeton, NJ, USA, July 7-14, 2008 Proceedings. 2008: 385-398.

[10] CALCAGNO C, DISTEFANO D, O'HEARN P, et al. Compositional shape analysis by means of bi-abduction[C]//Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 2009: 289-300.

[11] CALCAGNO C, DISTEFANO D, O'HEARN P W, et al. Compositional shape anal-

ysis by means of bi-abduction[J]. Journal of the ACM (JACM), 2011, 58(6): 1-66.

[12] JACOBS B, SMANS J, PIESSENS F. A quick tour of the VeriFast program verifier [C]//Programming Languages and Systems: 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28-December 1, 2010. Proceedings 8. 2010: 304-311.

[13] JACOBS B, SMANS J, PHILIPPAERTS P, et al. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java.[J]. NASA Formal Methods, 2011, 6617: 41-55.

[14] APPEL A W. Verified Software Toolchain: (Invited Talk)[C]//Programming Languages and Systems: 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings 20. 2011: 1-17.

[15] LEROY X, BLAZY S, KÄSTNER D, et al. CompCert-a formally verified optimizing compiler[C]//ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress. 2016.

[16] CAO Q, BERINGER L, GRUETTER S, et al. VST-Floyd: A separation logic tool to verify correctness of C programs[J]. Journal of Automated Reasoning, 2018, 61: 367-422.

[17] APPEL A W, BLAZY S, LEROY X, et al. Program logics for certified compilers[M]. Cambridge University Press, 2014.

[18] FLOYD R. Assigning meanings to program[C]//Proc. Symposia in Applied Mathematics, 1967: vol. 19. 1967: 19-32.

[19] HOARE C A R. An axiomatic basis for computer programming[J]. Communications of the ACM, 1969, 12(10): 576-580.

[20] O'HEARN P. Separation logic[J]. Communications of the ACM, 2019, 62(2): 86-95.

[21] 江南, 李清安, 汪吕蒙, 等. 机械化定理证明研究综述[J]. 软件学报, 2019, 31(1): 82-112.

[22] PIERCE B C, de AMORIM A A, CASINGHINO C, et al. Logical Foundations: vol. 1 [M]. Ed. by PIERCE B C. Electronic textbook, 2023.

[23] The Coq Proof Assistant Reference Manual - Version 8.17[A]. The Coq Development Team, 2023.

[24] PAULIN-MOHRING C. Introduction to the calculus of inductive constructions[Z]. 2015.

[25] ANDREW W. APPEL L B, CAO Q. Verifiable C: vol. 5[M]. Ed. by PIERCE B C. Electronic textbook, 2023.

[26] BARRETT C, TINELLI C. Satisfiability modulo theories[M]. Springer, 2018.

[27] DE MOURA L, BJØRNER N. Z3: An efficient SMT solver[C]//Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14. 2008: 337-340.

[28] BARBOSA H, BARRETT C, BRAIN M, et al. cvc5: A versatile and industrial-strength SMT solver[C]//Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings, Part I. 2022: 415-442.

[29] DUTERTRE B. Yices 2.2[C]//Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26. 2014: 737-744.

[30] KING J C. Symbolic execution and program testing[J]. Communications of the ACM, 1976, 19(7): 385-394.

[31] CADAR C, SEN K. Symbolic execution for software testing: three decades later[J]. Communications of the ACM, 2013, 56(2): 82-90.

[32] WADLER P, BLOTT S. How to make ad-hoc polymorphism less ad hoc[C]// Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 1989: 60-76.

[33] SCHURR H J, FLEURY M, BARBOSA H, et al. Alethe: Towards a Generic SMT Proof Format[C]//PxTP 2021-7th Workshop on Proof eXchange for Theorem Proving: vol. 336. 2021: 49-54.

# 致 谢

感谢曹钦翔老师这两年多来对我的指导，带领我接触形式化验证的科研实践。在本项目的推进过程中，曹老师不断给我们指明正确的方向，帮助我们解决了许多难题。

感谢秦健行同学和吴熙炜学长的帮助，我在与他们的讨论中解决了许多问题。

感谢上海交通大学和电院计算机系在这四年里对我的培养，感谢这四年里所有帮助过我的老师和同学。

感谢我的女朋友一直以来的陪伴和支持。

# VST-IDE: INTERACTIVE PROGRAM VERIFICATION TOOL (SEPARATION LOGIC SOLVING)

Formal verification is a means of ensuring software correctness, whereby a program is abstracted into mathematical propositions and verified through logical rules, ensuring complete program accuracy. Formal verification finds extensive applications in fields such as aviation and space industry where safety is crucial. However, it requires a high level of expertise and verification cost is also high. The main objective of this project is to reduce the threshold and cost of C program verification, and merge verification with the development process, thus helping software developers to improve verification efficiency and achieve "safety in development".

VST-IDE project verifies C language programs. C language is a widely used general-purpose programming language, applied in fields such as system software, application software, embedded software, etc. C language is concise in syntax and easy to understand, but its semantics are complex and prone to errors. Therefore, formal verification of C language programs is a challenging task. The focus of this project is on memory safety, such as detecting issues such as memory leaks, null pointer references, illegal memory access, etc. The programs verified in this project include function calls, thus it is an interprocedural analysis.

VST-IDE is an online editor where users can write C language programs and write assertions in comments to achieve program verification during development. The principle of this project is to use a modified Compcert compiler to compile source code into Abstract Syntax Tree (AST), and then a symbolic execution tool reads the AST and performs symbolic execution starting from the preconditions. During symbolic execution, if an assertion is encountered in the comment, separation logic is used for solving to determine whether the program state obtained from symbolic execution can deduce the assertion, thereby achieving verification.

Compared to existing verification tools, the main advantage of VST-IDE is its higher degree of automation: users only need to write specifications for each function, and write loop invariants when encountering loops in the function, without writing any other code, to achieve verification.

The VST-IDE project is divided into three parts: the compiler frontend, symbolic execution, and entailment checking based on separation logic. The first two parts are completed by other students and are briefly introduced here. The third part is completed by me, and is the focus of this article.

In the compiler frontend, we modified Compcert to compile "partial programs", i.e. unfinished programs, such as an incomplete function, so that users can verify during development rather than verifying the entire program after completion, thereby improving development and verification efficiency. Furthermore, since VST assertions are relatively complex, we define a C-style "assertion language" in C language comments, to reduce the learning cost for users.

Symbolic execution is a program analysis technique in the computer science field that replaces precise values with abstract symbols as program input variables to derive abstract output results for each path. This technique has certain applications in hardware and low-level program testing, and can effectively detect vulnerabilities in programs. In this project, we start symbolic execution from preconditions, and branch according to the conditions of control statements encountered during execution, then merge the program states obtained from various branches by disjunction during subsequent mergers.

This article focuses on entailment relation solving, which aims to determine whether the assertion obtained from symbolic execution can infer the assertion defined by the user in the comments. This project has modularized the spatial assertion section of the assertion, and by implementing different sub-modules, it can support the solving of different data structures. During the solving process, this project also performs some special operations to handle specific cases. Finally, this project outputs the solving process in the form of proof rules and VST proof code for users to verify the correctness of the solving process.

This article mainly discusses the verification method of single assertions and the implication relationship between them, since the verification of the implication relationship of multiple assertions is based on the single case. The verification of implication relationship is mainly divided into several parts: pre-processing (including filtering of empty heaps and acquisition of pure facts), solvers for Prop/Local/Sep, field-based space assertion splitting algorithm, and simplification algorithm for space assertions. After introducing the overall solution process, this article first introduces some important definitions, and then provides

detailed introductions for each part.

We first introduce some definitions, including variable types of the source program, predicate definitions in the C language comments, separation logic predicate definitions, space assertion definitions, expression types and extended expression types (ExtExpr) definitions, and definitions of two type classes used for modular implementation.

We need to filter out space assertions that can currently be determined as empty heaps to avoid affecting subsequent solutions, mainly considering user-defined separation logic predicates. For different types of predicates, their standards for determining empty heaps are also different. If our operation object is on the left side of the implication relationship, then the proposition obtained can be used as a condition (i.e., pure fact) for subsequent proofs. If our operation object is on the right side of the implication relationship, then the proposition obtained will be our target for proof.

Pure fact refers to a proposition that we obtain during the solution process and does not involve memory state. It is mainly used to prove Target's Prop in the final stage of the solution and to determine whether Target's Sep can be transformed in the solution process. Pure facts are obtained by analyzing Resource's Sep from three perspectives: "definitely an empty heap", "definitely not an empty heap", and determination based on separation logic rules.

The Prop solver is mainly used to prove Target's Prop using Resource's Prop in the later stage of the solution and to prove some temporary Props generated during the solution process. The Local solver is mainly used for instantiation of variables related to Local and for eliminating Local. Since we will not modify Local during the proof process, these two steps can actually be completed simultaneously. The Sep solver is mainly used for instantiation of variables related to Sep and for eliminating Sep. Since we may modify Sep during the proof process and may generate new existence variables, these two steps need to be performed separately, i.e., instantiating Sep in the pre-processing stage and then eliminating Sep, while also instantiating newly generated existence variables during the solution stage.

Generally, Resource's Sep and Target's Sep are not completely identical, and we need to transform Target's Sep before inputting them into the Sep solver in order to obtain the final result. The transformation here mainly involves the splitting of user-defined predicates, such as splitting a linked list into its header node and the remaining portion. Since the result of splitting is related to the field of the structure defined by the user, we call this kind of splitting

"field-based splitting". For most data structures, we can divide their fields into two categories, i.e., data fields and link fields. For each custom predicate in Target's Sep, we search for a single heap space assertion with the same address in Resource's Sep, and determine whether the assertion meets the definition of the field. If it meets a certain definition of the field, the corresponding custom predicate is split according to the definition.

Simplifiable Address (SimplAddr) is mainly used to determine whether the related predicates of the data structure segment are empty heaps, such as linked list segments, double linked list segments, and partial trees (i.e., double linked list segments of half-trees). We obtain SimplAddr before eliminating the split segment, and compare the remaining space assertion with SimplAddr after elimination to determine whether the space assertion is an empty heap.

In order to verify the correctness of the solver's results, we save the proof rules used during the solution process and output them in the solution result. We have defined different proof rules for Prop, Local, and Sep. At the same time, in order to maintain the simplicity of the output proof rules, we only retain the information used in the proof process when outputting them.

Finally, we introduced the program samples that the VST-IDE project is currently able to verify, and explained the process of implication relationship verification through an example in detail.