分离逻辑的定理证明器的设计与实现

Design and Implementation of Separation Logic Theorem Prover

计算机科学与技术学院

何春晖 PB09210183

冯新宇 教授

二〇一三年六月

中国科学技术大学

University of Science and Technology of China

本科毕业论文

A Dissertation for the Bachelor's Degree

分离逻辑的定理证明器的设计与实现

Design and Implementation of Separation Logic Theorem Prover

姓　　名	何春晖
B.S. Candidate	Chunhui He
导　　师	冯新宇 教授
Supervisor	Prof.Xinyu Feng

二〇一三年六月

June, 2013

中国科学技术大学

学士学位论文

题　　目	分离逻辑的定理证明器的设计与实现
院　　系	计算机科学与技术学院
姓　　名	何春晖
学　　号	PB09210183
导　　师	冯新宇 教授

二〇一三年六月

University of Science and Technology of China

A Dissertation for the Bachelor's Degree

Design and Implementation of Separation Logic Theorem Prover

B.S. Candidate Chunhui He

Supervisor Prof.Xinyu Feng

Hefei, Anhui 230026, China

June, 2013

致　　谢

感谢冯新宇老师、李兆鹏老师和陈意云老师在学习、生活方面对我的帮助、支持和鼓励。冯老师、李老师多次从繁忙工作中抽出时间与我讨论设计方案、传授经验，使我在较短时间内理清了思路；论文初稿完成后，又提出很多意见，帮助我完善论文，最终顺利完成毕业设计。

感谢与我同行去苏州完成毕业设计的朋友们，以及苏州的师兄师姐们。近三个月的学习生活令人难忘。

感谢父母。祝愿身体健康，天天开心！

本课题得到国家自然科学基金（NO.61073040, NO.61003043）资助。

目　录

致　谢 **i**

摘　要 **vii**

Abstract **ix**

第一章　绪论 **1**

第一节 研究背景 1

一、 程序验证 1

二、 携带证明的代码 1

三、 分离逻辑 2

第二节 相关工作 2

一、 定理证明系统 2

二、 分离逻辑证明 2

三、 讨论 2

第三节 本文工作 3

第二章　证明器的总体设计 **5**

第一节 输入语言 5

第二节 证明项 7

一、 相关概念 7

二、 表达证明项的数据结构 8

第三节 结构及流程 8

一、 分离逻辑的证明器所做的工作 8

二、 一阶逻辑的证明器所做的工作 9

第三章　命题逻辑命题的证明 **11**

第一节 输入语言 11

第二节 结构 11

第三节 SAT 求解 13
一、 DPLL 算法 13
二、 证明项的构造 14
第四节 命题的规范化 14
一、 规范到否定范式 15
二、 规范到合取范式 16

第四章 等词与未解释函数理论命题的证明 **19**
第一节 输入语言 19
第二节 决策过程 19
一、 一致闭包 19
二、 实现 20
三、 证明项的构造 20

第五章 线性整数理论命题的证明 **23**
第一节 输入语言 23
第二节 单纯形法 23
一、 标准型 23
二、 决策过程 25
三、 证明项构造 26
第三节 分支限界法 26
一、 决策过程 27
二、 证明项构造 27

第六章 理论整合框架的实现 **29**
第一节 Nelson-Oppen 框架 29
第二节 实现 30
第三节 证明项的生成 30

第七章 分离逻辑命题的证明 **33**

第一节 语言 33

第二节 决策过程 33

第三节 证明项 34

一、 分离逻辑在 Coq 中的表达 34

二、 证明项的构造 35

第八章 实现与性能测试 **37**

第一节 实现 37

第二节 性能测试 37

一、 测试对象 37

二、 测试数据的选取 38

三、 测试结果 39

四、 讨论 39

第三节 结论 40

第九章 总结 **41**

第一节 工作总结 41

第二节 进一步的工作 41

参考文献 **43**

摘　要

本文针对程序验证中定理自动证明的需求，从头设计了一个分离逻辑的定理证明器。

本文设计的定理证明器基于与 Z3 相同的 SMT 结构，但能够输出 Coq 兼容的证明项，并采用类似 Smallfoot 的方法验证分离逻辑命题，以试图改进现有系统在表达能力、自动化和可信性三方面不平衡的问题。

其中，为了输出 Coq 兼容的证明项，本文详细地研究了 SMT 定理证明器每个组件的结构，并针对其特点给出了证明项的构造方法。最终做到了定理证明的每一步都有证明项输出。

根据文中的设计，本文还初步实现了设计中一阶逻辑的证明部分，从而初步验证了设计的可行性。测试中，证明器能够全自动地证明一阶逻辑命题，并且自动构造 Coq 兼容的证明项，验证了其可信、自动的特点。

关键字:　程序验证，自动定理证明，分离逻辑，Coq 兼容证明项生成

Abstract

In this dissertation, we designed a separation logic theorem prover for program verification from scratch.

Our theorem prover is a SMT solver like Z3, but it can produce Coq-compatible proof term, and proves separation logic theorem like Smallfoot. Compared with these existing proof systems, it tries to achieve a better balance between the ability to express, automatization and reliability.

In order to produce Coq-compatible proof term, we emphatically explored a SMT solver, then gave a way to generate proof term for each part.

We also implemented the first-order logic part of our theorem prover for testing feasibility. In testing section, our theorem prover can prove first-order logic theorem automatically, then produce Coq-compatible proof term.

Keywords: Program Verification, Automatic Theorem Prover, Separation Logic, Coq-Compatible Proof Term Generation

第一章　绪论

第一节　研究背景

一、　程序验证

计算机技术已经广泛地应用到我们的日常生活中。种类繁多的计算机软件在提供丰富功能的同时，其缺陷也给人们带来困扰。

在一些关键领域，如航天、电信、银行，软件的缺陷可能引发重大的人身和财产损失。因此，提高这些软件的可靠性十分必要。程序验证（Program Verification）就是一种通过逻辑推理来证明软件具有特定安全性质，从而提高软件可靠性的方法。

1969 年，C. A. R. Hoare 提出了验证程序正确性的公理系统：Hoare 逻辑 [1]。从此 Hoare 逻辑成为程序验证的重要方法。

二、　携带证明的代码

携带证明的代码（Proof Carrying Code）[2][3] 是由 Necula 提出的。携带证明的代码携带了机器可检查的证明项（Machine Checkable Proof Term），证明项能够说明代码能满足所需的安全性质。代码消费方仅需用可信任的证明检查器 (Proof Checker) 去静态地检查证明项，就能够检查程序的安全性。

携带证明的代码的构造给定理证明器提出了新的要求，即定理证明器在宣告命题成立的同时，还要给出相应的机器可检查证明项。

三、 分离逻辑

分离逻辑（Separation Logic）[4] 是 Hoare 逻辑的一种扩充。其核心是引入分离合取（Separation Conjuntion）符号，描述两个堆存储区域不相交，从而能够验证一类带有指针的程序。

第二节 相关工作

一、 定理证明系统

Z3[5][6] 是微软研究院出品的高性能自动定理证明器。Z3 属于 SMT 求解器，由一个 SAT 求解器和多个理论求解器组合而成。Z3 目前能够支持多种一阶理论的自动证明，如线性算术、位向量、未解释函数、数组、量词等。此外，支持多种输入格式和编程接口，被广泛用于程序验证等领域。

Coq[7] 是一种高阶逻辑（Higher-Order Logic）的定理证明辅助系统。Coq 提供了一个表达力很器的语言及一个交互式的证明环境。Coq 的证明过程不是完全自动的，而是需要人交互地给系统提示，最终构造出机器可检查的证明项。

二、 分离逻辑证明

分离逻辑的证明工具以 Smallfoot[8] 为代表。它提供是一个基于分离逻辑的自动的验证工具原型。

三、 讨论

以面向分离逻辑程序验证的角度，现有定理证明器的主要问题是：表达能力、自动化、可信性三者没有作出很好的折中。

Z3 是全自动的，但表达能力不足以包括分离逻辑。虽然能够出具证明，但格式为自定义的，不能被广泛信任的 Coq 证明检查器检查。

Coq 被看作一个逻辑框架而被广泛研究和信任，但由于表达能力过强，很多应该自动化的推理无法自动进行。

Smallfoot 等虽然能表达分离逻辑，但不能出具证明。由于是用于演示分离逻辑验证的原型系统，支持的理论也不够丰富。

第三节　本文工作

本文针对目前定理证明器的不足，从头设计一个分离逻辑的定理证明器。力图达到表达能力、自动化、可信性三者平衡。

具体来说，本文设计的定理证明器基于与 Z3 相同的 SMT 结构，但能够输出 Coq 兼容的证明项，并采用类似 Smallfoot 的方法验证分离逻辑命题。从而分别克服了 SMT 结构可信性不足、表达能力不足的缺点。

本文后续安排如下：

第二章说明证明器的总体设计。接下来的每一章根据设计框图的划分逐一说明具体的设计。最后介绍实现情况和测试，总结全文。

第二章　证明器的总体设计

本章给出证明器的总体设计，并说明证明器的各个部分之间如何协作证明一个命题。

第一节　输入语言

本文设计的证明器需要证明的命题可以分成两部分：一部分是纯的一阶逻辑及其理论，另一部分是分离逻辑。前者主要表达程序中变量、函数之间的相等和不等关系，后者主要表示程序中数据结构的变化。

两者在各自推理的同时，前者也为后者服务，如向后者提供指针运算、指针相等关系等内容。这决定了证明器要支持一阶逻辑中的算术理论和相等理论。

由上述需求，我们设计的命题语言如图2.1所示。

$$
\begin{array}{rrcl}
\text{整数常元} & C & ::= & 0 \mid 1 \mid -1 \mid \ldots \\
\text{整型变元} & X & ::= & x_0 \mid x_1 \mid x_2 \mid \ldots \\
\text{未解释函数} & F & ::= & f_0 \mid f_1 \mid f_2 \mid \ldots \\
\text{命题变元} & A & ::= & A_0 \mid A_1 \mid A_2 \mid \ldots \\
\text{公式项} & T & ::= & C \mid X \mid F(T,\ldots,T) \mid +(T,T) \mid \cdot(C,T) \\
\text{一阶逻辑原子公式} & Y & ::= & T \leq T \mid T = T \mid A \mid \mathrm{true} \\
\text{一阶逻辑公式} & \Pi & ::= & Y \mid \neg\Pi \mid \Pi \to \Pi \mid \Pi \wedge \Pi \mid \Pi \vee \Pi \\
\text{分离逻辑原子公式} & H & ::= & T \mapsto T \mid \mathrm{emp} \\
\text{分离逻辑公式} & \Sigma & ::= & H \mid \Sigma * \Sigma \\
\text{命题} & P & ::= & \Pi \wedge \Sigma \to \Pi' \wedge \Sigma'
\end{array}
$$

图 2.1 命题语言的语法

证明器接受的命题的整体是一个蕴含式，前件和后件都分别由 Π 和 Σ 两部分构成，分别代表一阶逻辑部分和分离逻辑部分。

Π 和 Π' 是一个无量词的一阶逻辑公式。二元函数 $+$ 和 $\cdot$ 分别解释为整数环上的加法和数乘运算，这意味着只能支持线性的整数算术；除此之外，可出现有限个有限元的函数，它们是未解释的。

Σ 和 Σ' 是包含分离逻辑符号的公式。为简化起见，分离逻辑公式只能用 $*$（分离合取）联结词连接，原子公式只能是 emp 和 $T_1 \mapsto T_2$ 的形式，分别代表空堆和仅包含一项的堆。

命题语言语义的定义如图2.2。

$$
\begin{array}{rcl}
s, h \models P & & \\
s \text{ 是栈} & & s : T \to \mathbb{Z} \\
h \text{ 是堆} & & h : \mathbb{Z} \backslash \{0\} \rightharpoonup \mathbb{Z} \\
s, h \models t_1 = t_2 & \iff & s(t_1) = s(t_2) \\
s, h \models t_1 \leq t_2 & \iff & s(t_1) \leq s(t_2) \\
s, h \models \text{true} & \iff & \text{真} \\
s, h \models \neg P & \iff & s, h \models P \text{ 为假} \\
s, h \models P \to Q & \iff & \text{若 } s, h \models P\text{，则 } s, h \models Q \\
s, h \models P \wedge Q & \iff & s, h \models P \text{ 且 } s, h \models Q \\
s, h \models P \vee Q & \iff & s, h \models P \text{ 或 } s, h \models Q \\
s, h \models \text{emp} & \iff & \text{dom}(h) = \emptyset \\
s, h \models t_1 \mapsto t_2 & \iff & s(t_1) \neq 0 \text{ 且 } \text{dom}(h) = \{s(t_1)\} \\
 & & \text{且 } h(s(t_1)) = s(t_2) \\
s, h \models P * Q & \iff & \text{存在 } h1, h2,\ h1 \perp h2 \text{ 且 } h = h1 * h2 \\
 & & \text{且 } s, h1 \models P \text{ 且 } s, h2 \models Q
\end{array}
$$

图 2.2 命题语言的语义

由于分离逻辑隐式包含了堆，这在纯的一阶逻辑中是没有的。因此图中的语义定义主要说明了一阶逻辑符号拓展到有堆出现时的意义。

第二节　证明项

证明项是一种语言，它可以用来表示证明的推理过程。本文实现的证明器通过对每一个命题出具 Coq 兼容的证明项，来满足携带证明代码对携带机器可检查证明的要求。

一、　相关概念

1　Curry-Howard 同构

Curry-Howard 同构 [9] 是指逻辑演算系统与程序类型系统的相对应关系。它指出，自然推理系统与 λ-演算的对应关系如表2.1。

表 2.1 自然推理系统与 λ-演算的对应关系

自然推理	λ-演算
假设	对应类型的自由变量
蕴含引入	函数抽象
蕴含消去	函数应用
定理	对应类型的表达式

于是，命题的证明问题就可以看作一个寻找对应类型的 λ-表达式的过程。例如，我们欲证明希尔伯特系统中的 L1：

$$\vdash P \to (Q \to P)$$

我们可以构造如下 λ-表达式：

$$\lambda(H_0 : P).(\lambda(H_1 : Q).H_0)$$

上式的类型为 $P \to (Q \to P)$，因此它是 L1 的一个证明。

2　归纳构造演算与 Coq

归纳构造演算（Calculus of Inductive Constructions）是一种 λ-演算的扩展，它结合了逻辑中的一些最新进展，因而能力更强。

Coq[7] 是一个基于归纳构造演算的高阶逻辑定理证明辅助系统。其独到之处在于，表达程序和表达证明都使用同一套语言。

上一节 L1 的证明用 Coq 证明项表达如下：

```
(fun H0:P => (fun H1:Q => H0))
```

二、 表达证明项的数据结构

我们将证明项保存为一个类似 λ-表达式的形式。

证明项类型叫做 proof，定义用类 ML 语言描述如下：

```
type param = (string * prop)          (* 参数 *)
type proof =
| Hole                                  (* 空 *)
| Id of string       (* 公理、定理、变量标识 *)
| Lam of param list * proof     (* 函数抽象 *)
| App of proof list             (* 函数应用 *)
```

Lam 和 App 构造分别模拟 λ-表达式的函数抽象和函数应用。唯一不同的是，参数部分都被扩展成了列表，以更加便于实际使用。

Hole 代表空，意味着没有找到命题的证明。

proof 并不能表达纳构造演算的所有构造，但是在本文设计的证明器是足够用的。

第三节 结构及流程

如图2.3所示，证明器逻辑上可以分成两大部分。一部分是传统的基于一阶逻辑的证明器；另一个部分是分离逻辑的证明器。

一、 分离逻辑的证明器所做的工作

证明器开始运行时，接收命题语言 $\Pi \wedge \Sigma \rightarrow \Pi' \wedge \Sigma'$。

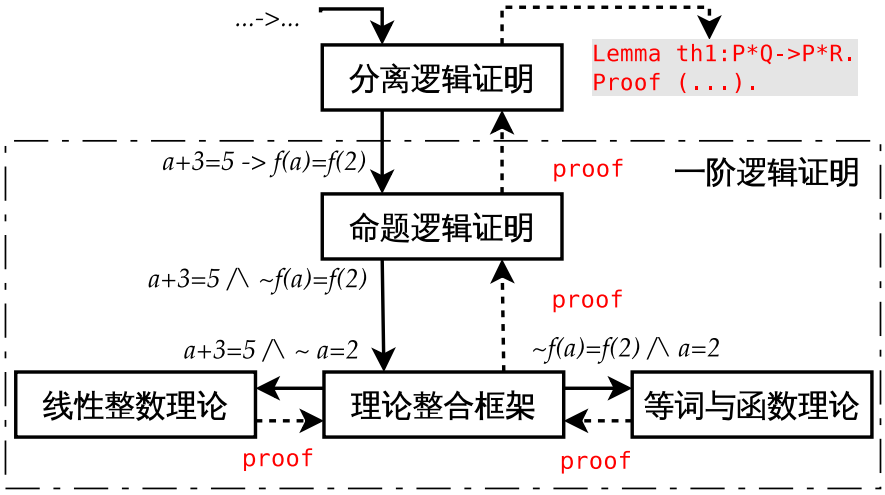


图 2.3 证明器的结构

这个命题的证明可以化为求证：

$$\vdash \quad \Pi \wedge \Pi'' \rightarrow \Pi' \tag{2-1}$$

$$\vdash \quad \Pi \wedge \Sigma \rightarrow \Sigma' \tag{2-2}$$

2-1式是一个纯的一阶逻辑的证明过程，因此直接将该部分命题发送到一阶逻辑的证明器求证。其中 Π'' 是分离逻辑部分中推出的一些算术信息，如指针非空和不同堆的地址不同。

2-2式是一个带分离逻辑的证明过程。该部分将由分离逻辑的证明器使用分离逻辑中的定理将该过程分解成求证一系列纯一阶逻辑命题，并发送到一阶逻辑的证明器求证。该部分的方法在第七章说明。

分离逻辑的证明器综合两式，得到命题的最终证明。

二、 一阶逻辑的证明器所做的工作

一阶逻辑的证明器结构属于目前较先进的 SMT（Satisfiability Modulo Theories）证明器结构。

对于每一个被发送到一阶逻辑的证明器的命题，首先经过命题逻辑证明模块。该模块将所有不同的公式项看作一个独立的命题变元，最终化简成为约束满足（SAT）问题，该步的具体方法在第三章说明。

由于命题逻辑证明求解时不会关心命题公式项的具体内容，因此它在求解时会生成一系列可能否证命题的“反例”赋值。这些“反例”重新构成命题后被发送到理论整合框架。在这个框架中，命题被按照公式项分成不同的部分，送给不同的理论求解器求解。由于考察公式项的具体结构，“反例”将被一一反驳。该步具体方法在第四、五、六章说明。

命题逻辑证明模块综合这些反驳信息，得到一阶逻辑命题的最终证明。

第三章　命题逻辑命题的证明

第一节　输入语言

$$\begin{array}{rcl} & & Y \text{ 是原子公式} \\ \Pi & ::= & Y \mid \neg\Pi \mid \Pi \to \Pi \mid \Pi \wedge \Pi \mid \Pi \vee \Pi \end{array}$$

图 3.1 命题逻辑语言

本部分接收的语言是纯一阶逻辑命题的命题框架。命题框架是指，不理会原子公式的具体结构，只是简单地将不同的原子公式看作不同的命题变元。

第二节　结构

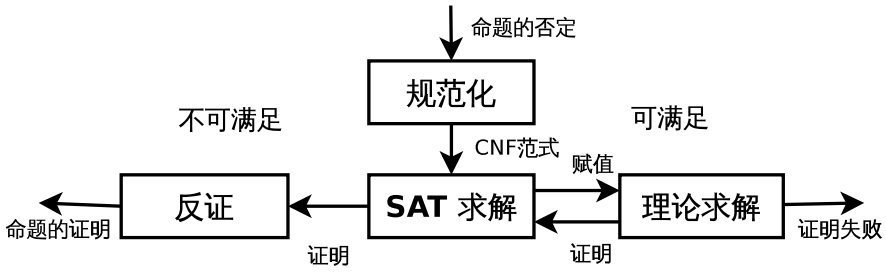


图 3.2 命题逻辑证明的结构

如图3.2，本部分的核心是一个 SAT 求解器。

SAT 求解器能够解决下列一类问题：

定义 (SAT 问题). 设 X 是布尔变量集，$|X| = n$，α 是合取范式，并有 $\alpha = \bigwedge_i (\bigvee_{j_1} x_{ij_1} \vee \bigvee_{j_2} \neg x_{ij_2})$，$x_{ij} \in X$。

求一种赋值方法 $(b_1, b_2, \ldots, b_n)$*，使得* α *为真。若不存在这种赋值方法，则称* α *不可满足。*

我们使用下面的定理来利用 SAT 求解器证明定理：

定理. *若* $\neg\alpha$ *不可满足，则* $\vdash \neg\neg\alpha$*。*

待证的命题首先被否定并化为合取范式（CNF）α'，然后送往 SAT 求解器求解。SAT 求解器有可能返回两种结果：可满足和不可满足。

若结果为不可满足，如图3.2左半部分，则说明 $\vdash \neg\neg\alpha$，直接将 SAT 求解过程的证明证明项取出，并应用在 Coq 定义的双重否定引理：

```
NNPP : forall p:Prop, ~ ~ p -> p.
```

即可得证。

若结果为可满足，如图3.2右半部分，将得到一个原子命题的赋值向量 $\vec{v} = (b_1, b_2, \ldots, b_n)$。这说明，单从命题的命题框架看，无法得出命题在赋值 $\vec{v}$ 下成假。

此时，利用命题编码，可以将赋值转换成一个原子命题的合取形式：

定义 (命题编码)**.** *命题编码是指合取式*

$$\beta = \bigwedge_i y_i$$

其中

$$y_i = \begin{cases} x_i & \text{若} b_i = \text{true} \\ \neg x_i & \text{若} b_i = \text{false} \end{cases}$$

$$x_i \in X$$

此时将此编码 β 送到理论求解器求解。理论求解器若能得出成假结论，则重新触发 SAT 求解器继续求解；否则，证明失败。

第三节　SAT 求解

一、　DPLL 算法

本文实现的 SAT 求解器使用 DPLL 算法求解。

DPLL 算法是由 Davis、Putnam、Loveland、Longemann 联合提出的一种求解 SAT 问题的完备方法。

DPLL 算法的核心是回溯法。如图3.3求解的是合取范式 $P \wedge (\neg P \vee Q) \wedge \neg Q$ 的可满足赋值的搜索树。

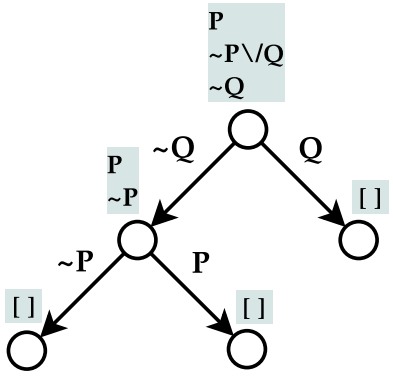


图 3.3 DPLL 算法的的搜索树

算法首先尝试 Q 的赋值。当 Q 赋值为真，则与范式中的 $\neg Q$ 矛盾，因此不可行，回溯；当 Q 赋值为假时，原式变为 $P \wedge (\neg P \vee \text{false}) \wedge \text{true}$，即 $P \wedge \neg P$，然后对 P 做相同的尝试，直到找到一个赋值使得命题成真。

本例经过尝试，发现没有成真的赋值，返回不可满足。

DPLL 算法还有不少优化：

1. 变量删减：将式子中仅出现正文字或仅出现反文字的变量删除
2. 变量排序：按照一定的次序搜索，加快剪枝
3. 约束传播：尽早由部分赋值发现矛盾
4. 常数时间回溯等

在这些优化下，DPLL 算法能够求解变量数上千的 SAT 问题。

二、 证明项的构造

DPLL 算法的证明项构造一般基于归结（resolution）规则构造：

$$\vdash (P \vee Q) \wedge (\neg P \vee R) \to Q \vee R$$

其中 P、Q、R 都是一阶公式。

当使用归结规则构造 DPLL 算法的证明项时，需要先对搜索树做若干分析，才能生成证明项，不是很直接。本文提出一个直接生成证明项的方法。

简单地说，这个方法基于如下定理：

$$\alpha' \wedge E \vdash (P \to \text{false}) \to (\neg P \to \text{false}) \to \text{false}$$

其中 α' 是合取范式，E 是部分赋值的命题编码，P 是范式中的某一文字。

其意义为：在部分赋值 E 下，若文字 P 赋值为真，最终得到矛盾，且文字 P 赋值为假，最终也得到矛盾，则 α' 不可满足。

应用时，只需要在 DPLL 搜索树的每个节点按照后序遍历使用即可生成证明。以图3.3为例，其生成的证明序列是：

$P \wedge (\neg P \vee Q) \wedge \neg Q \wedge Q \wedge P \vdash \text{false}$	(1)	$P \wedge (\neg P \vee Q) \wedge \neg Q$
$P \wedge (\neg P \vee Q) \wedge \neg Q \wedge Q \wedge \neg P \vdash \text{false}$	(2)	$P \wedge \neg P$
$P \wedge (\neg P \vee Q) \wedge Q \wedge Q \vdash \text{false}$	(3)	由定理及 (1) (2)
$P \wedge (\neg P \vee Q) \wedge Q \wedge \neg Q \vdash \text{false}$	(4)	$Q \wedge \neg Q$
$P \wedge (\neg P \vee Q) \wedge Q \vdash \text{false}$	(5)	由定理及 (3) (4)

第四节 命题的规范化

命题规范化要做的事是将任意的命题化成 SAT 求解器能够求解的合取范式的形式。

化简过程分为两步：先将命题化简为否定范式（NNF）, 再将否定范式化为合取范式。

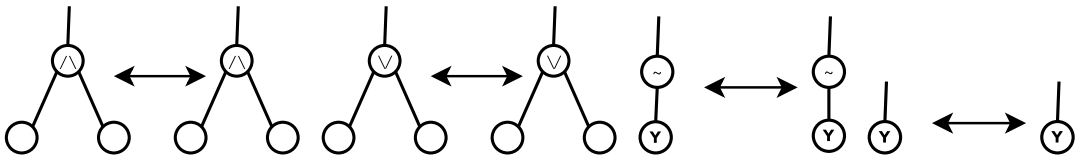
一、 规范到否定范式

规范到否定范式一共要做两件事：消除蕴含词 $\rightarrow$；将否定词 $\neg$ 深入到文字上，并且保证文字前最多只有一个否定词。

从语法结构的观点看，本步骤实际上是由一棵语法树变换到另一棵语法树。由于语法树是由图3.1构造地定义的，因此只需要考察每一个局部结构的变化情况，再逐级递归即可。

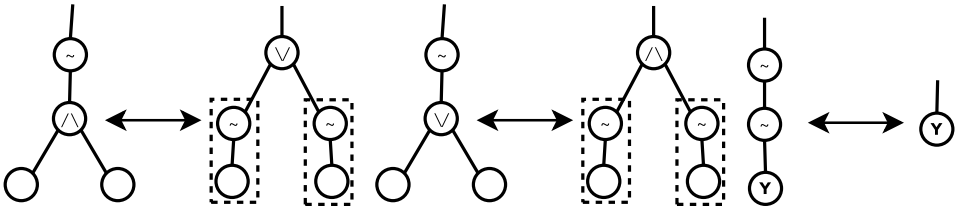
1 变换分类

变换一共分两种情形：A 型、B 型。图中虚线代表递归时应该看作整体。

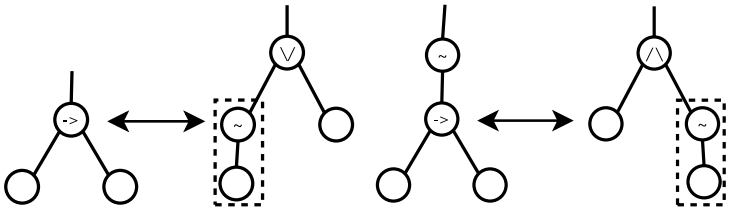


(a) and (b) or (c) not Y (d) Y

图 3.4 A 型：结构不变



(a) not and (b) not or (c) not not Y



(d) imply (e) not imply

图 3.5 B 型：结构改变

2 证明项构造

证明项的构造是直接的，只需在递归时引用 Coq 中预定义的引理即可。

每一种情况对应一个引理，罗列如下。

(1) A 型引理

```
L_and : forall A B C D:Prop, (A<->B)->(C<->D)->((A/\C)<->(B/\D)).
L_or : forall A B C D:Prop, (A<->B)->(C<->D)->((A\/C)<->(B\/D)).
L_not : forall A B:Prop, (A<->B)->((~A)<->(~B)).
iff_p : forall P:Prop, P<->P.
```

(2) B 型引理

```
iff_nand : forall P Q:Prop, ~(P/\Q) <-> (~P\/~Q).
iff_nor : forall P Q:Prop, ~(P\/Q) <-> (~P/\~Q).
iff_nnp : forall P:Prop, ~~P<->P.
iff_imp : forall P Q:Prop, (P->Q)<->(~P\/Q).
iff_nimp : forall P Q:Prop, ~(P->Q)<->(P/\~Q).
```

二、 规范到合取范式

1 变换

从否定范式规范到合取范式是简单的，只需要递归地对公式中的 $\vee$ 做对 $\wedge$ 的分配律即可。

2 证明项构造

理论上，只要在变换时引用 $\vee$ 对 $\wedge$ 的分配引理，即可得到合取范式的证明项。但是实践上这样会存在 $\wedge$ 和 $\vee$ 上交换律和结合律的问题。

例如，$P \vee (Q \vee R)$ 和 $(P \vee Q) \vee R$ 在日常的非严格的推理中，是看作相同的。但是在 Coq 系统中却是两个完全不同的类型。若要是类型相等，必须引用结合律。

因此，直接采用分配律后构造证明项后，结果的语法树几乎一定会出现很多“枝杈”，而非规范的右结合树。不是规范形式将会给 SAT 求解器的证明项构造带来很大负担。

本文提出一个直接由否定范式树生成合取范式每一个的合取支规范形式的方法。

下面以 $P \vee (Q \wedge R)$ 为例，说明其基本步骤。

(1) 从合取范式树抽取合取支 上式的合取范式为 $(P \vee Q) \wedge (P \vee R)$。容易编写一个递归的过程，抽取出合取支 $(P \vee Q)$ 和 $(P \vee R)$ 并暂存。

(2) 决定规范形式 这里可以根据需要（如根据 SAT 求解器的内部表示）决定每一个合取支的规范形式，如取为 $P \vee Q$ 和 $R \vee P$。

(3) 生成合取支的一个析取成分的证明项 选择 $P \vee Q$ 和 $R \vee P$ 里面一个较好证明的析取支，这里都选择 P。即证明：

$$\vdash P \vee (Q \wedge R) \to P$$

由于前件是否定范式，很容易根据 $\wedge$ 和 $\vee$ 的结构，写一个递归过程，引用 Coq 如下定理得到证明：

```
and_ind : forall A B P : Prop, (A -> B -> P) -> A /\ B -> P.
or_ind : forall A B P : Prop, (A -> P) -> (B -> P) -> A \/ B -> P.
```

(4) 补齐其余部分 写一个递归过程，引用下列 Coq 定理，补齐合取支的其他部分。

```
or_introl : forall A B : Prop, A -> A \/ B.
or_intror : forall A B : Prop, B -> A \/ B.
```

本例中，可用 `or_introl` 由 P 得到 $P \vee Q$；用 `or_intror` 由 P 得到 $R \vee P$。

至此，我们得到了否定范式到合取范式每一个合取支的规范形式的证明项。并且，由于 SAT 求解器实际实现时接收的就是合取范式合取支。因此，不必再用 `conj` 引理得到合取范式的证明项。

本文提出的这种方法，相当于把公式的化简、规范化、拆项三步放到了一步之中，减少了证明项的大小。

第四章　等词与未解释函数理论命题的证明

第一节　输入语言

$$
\begin{array}{rcl}
 & & F \text{ 是未解释函数} \\
 & & X \text{ 是整型变元} \\
T & ::= & X \mid F(T,\ldots,T) \\
Y & ::= & T = T \mid T \neq T \\
\Pi & ::= & T \mid \Pi \wedge \Pi
\end{array}
$$

图 4.1 等词与未解释函数语言的语法

本部分接收的语言是关于变元、函数间的相等和不相等关系的合取式。目标是证明合取式的不一致性。

第二节　决策过程

一、　一致闭包

一致闭包（Congruence Closure）[10] 是能解决本章形式的命题的一种效率较高的决策过程。

其基本步骤如下：

初始化 对每个命题项 $t_i \in T$，初始化一个集合，其初始元素为自身。

合并 对命题中的每一个相等关系 $t_p = t_q$，若 t_p 和 t_q 在不同集合 S_p 和 S_q，则合并 S_p 和 S_q 为 S_{pq}。

闭包 反复对命题中出现的所有的相同的函数对 $(f(x), f(y))$，检查 x 和 y 是否在同一集合。若在，合并 $f(x)$ 和 $f(y)$ 所在的集合。

判断 对不相等关系中的每一对 (x, y)，检查 x 和 y 是否在同一集合。若在，则产生矛盾，得到不一致证明。

二、 实现

上述算法的高效实现依赖于一个专门为上述操作设计的数据结构——并查集（Find-Union Set 或 Disjoint Set）。

并查集是多个树形结构组成的森林。每个树结点有一个域，指向其父结点（根结点可规定指向自身）。两个结点所代表的元素在同一个集合中，当且仅当两个结点拥有相同的根。

并查集的基本操作如下：

初始化 每个元素都指向自身。

合并 查找两个元素的根，然后使其中一个根指向另一个。

查找 若两个元素的根相同，则返回真，否则为假。

并查集还有一些优化，如按秩合并、路径压缩等。在这些优化下，其基本操作的时间复杂度约为常数。

用并查集实现一致闭包算法如图4.2所示。

三、 证明项的构造

Coq 中用于证明相等关系的定理有三条：

```
refl_equal : forall (A : Type) (x : A), x = x.
sym_eq : forall (A : Type) (x y : A), x = y -> y = x.
trans_eq : forall (A : Type) (x y z : A), x = y -> y = z -> x = z.
```

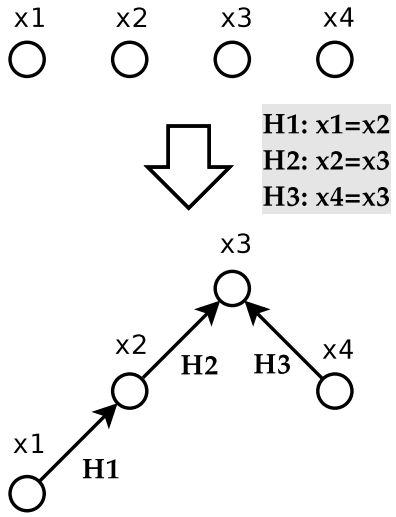


图 4.2 并查集实现的一致闭包算法

分别对应等词的自反、对称、传递性质。

观察图4.2，可见并查集的每一条边对应了一个相等关系。因此可以构造如下算法生成证明项（以证明 $x_1 = x_4$ 为例）：

1. 依传递律生成 $x_1 = x_3$ 的证明项；
2. 依传递律生成 $x_4 = x_3$ 的证明项，再依对称律生成 $x_3 = x_4$ 的证明项；
3. 由前两项依传递律生成 $x_1 = x_4$ 的证明项。

以上解决了生成证明项的总体框架问题，尚需每条边所代表的相等关系的证明项。为此，我们把并查集树结点的存储由单独的父结点指针 `parent` 扩充为二元组 `(parent, proof)`，分别表示父结点指针和对应的边的证明项。

扩展后的并查集的操作算法如下：

初始化 每个元素指向自身，元素 x 的证明项为 `(refl_equal x)`。

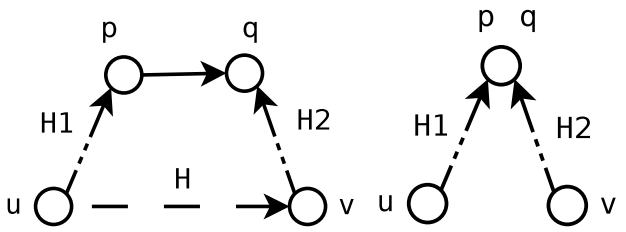
合并 设合并的元素对为 `(u, v)`，相等的证明为 `H`，如图4.3(a)。

1. 查找 `u` 的根为 `p`，使用 `trans_eq` 连接各个边的证明为 `H1`；
2. 查找 `v` 的根为 `q`，使用 `trans_eq` 连接各个边的证明为 `H2`；

3. 令 p 的父结点为 q，附加的证明为 (trans_eq (trans_eq (sym_eq H1) H) H2)。

查找 设查找的元素对为 (u, v)，如图4.3(b)。

1. 查找 u 的根为 p，使用 trans_eq 连接各个边的证明为 H1；
2. 查找 v 的根为 q，使用 trans_eq 连接各个边的证明为 H2；
3. 若 p 和 q 不等，返回假；否则，返回真，附加的证明 (trans_eq H1 (sym_eq H2))。



(a) 合并　　(b) 查找

图 4.3 带证明项的并查集

最后，还剩下求闭包过程中证明项的构造问题。这是通过等词的等价替换性质实现的：

```
eq_ind : forall (A : Type) (x : A) (P : A -> Prop),
       P x -> forall y : A, x = y -> P y
```

第五章　线性整数理论命题的证明

第一节　输入语言

$$
\begin{array}{rcl}
 & & C \text{ 是整数常元} \\
 & & X \text{ 是整型变元} \\
T & ::= & C \mid X \mid +(T,T) \mid \cdot(C,T) \\
Y & ::= & T = T \mid T \neq T \mid T \leq T \mid T > T \\
\Pi & ::= & T \mid \Pi \wedge \Pi
\end{array}
$$

本部分接收的语言是关于整数环上的线性不等式组，目标是证明不等式组的不可满足性。$\neq$ 和 $>$ 关系分别是 $=$ 和 $\leq$ 关系的否定形式。

第二节　单纯形法

一、　标准型

单纯形法是线性规划理论中的一种最优化方法。

原始的单纯形法解决的问题的标准型为：

$$\max Z = c_1x_1 + c_2x_2 + \cdots + c_nx_n$$
$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \cdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \\ x_j \geq 0, \quad j = 1, 2, \cdots, n \\ b_i \geq 0, \quad i = 1, 2, \cdots, m \end{cases}$$

任意的线性规划问题经过适当变换后均可以转换为标准型。

单纯形法在求解时，首先要找到一组初始解，它满足约束条件，但未必最优。接下来，从初始解开始经过一系列迭代，最终达到最优解。如果初始解找不到，则说明问题无解。

单纯形法找初始解的特性为线性不等式组的可满足判定提供了一种天然方法。即若能找到初始解，则不等式组可满足，否则不可满足。

依据上述思路，定义本文中单纯形法解决可满足问题的标准型如下：

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = s_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = s_2 \\ \cdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = s_m \\ x_j\text{无限制}, \quad j = 1, 2, \cdots, n \\ s_i \leq b_i, \quad i = 1, 2, \cdots, m \\ b_i \in \mathbb{Q}, \quad i = 1, 2, \cdots, m \end{cases}$$

上述标准型中去掉了优化函数 Z，因为可满足求解不关心；其次将 x_j 的限制放开，减少变换负担；最后在等式右边加入了 m 个有界的松弛变量，作用也是减少通常形式到标准型的负担。

至此，所有的相等与不等关系都可以化为适当的标准形式，以 $x_1 + x_2$ op 1 为例：

- 若 op 为 $\leq$，标准形式为 $x_1 + x_2 = s_k, \quad s_k \leq b_k = 1$
- 若 op 为 $>$，标准形式为 $-1 \cdot x_1 + -1 \cdot x_2 = s_k, \quad s_k \leq b_k = -(1+\delta)$
- 若 op 为 $=$，标准形式为 $x_1 + x_2 = s_k, \quad s_k \leq b_k = 1$ 和 $-1 \cdot x_1 + -1 \cdot x_2 = s_k, \quad s_k \leq b_k = -1$
- 若 op 为 $\neq$，分为两个目标，标准形式分别为 $x_1 + x_2 = s_k, \quad s_k \leq b_k = 0$ 和 $-1 \cdot x_1 + -1 \cdot x_2 = s_k, \quad s_k \leq b_k = -(1+\delta)$

其中，δ 是一个足够小的数。由于最终求解的是 $\mathbb{Z}$ 上的解，因此只需取 $\delta = 1$。

二、 决策过程

单纯形法的求解使用单纯形表。单纯形表是一个 m 行 n 列的矩阵外加一个 n 列的向量。矩阵的每一行对应一个基变量，每一列对应一个非基变量。向量每一列代表对应非基变量当前的赋值。

1. 初始时，每一行对应松弛变量 s_i，每一列对应原始变量 x_j，矩阵是标准型的系数。向量各分量为零。
2. 接下来，循环求解直到找到可满足实例或不可满足。
   (a) 首先，检查越界基变量。方法是检查所有代表松弛变量的行 i，计算 $s'_i = \sum_{j=1}^{n} a_{ij} \cdot x'_j$，其中 x'_j 是第 j 列代表的非基变量。若 $s'_i > b_i$，b_i 是松弛变量上界，则说明越界，需要进一步调整。否则，已找到可满足实例，返回 “可满足”。
   (b) 然后，重新指派非基变量赋值。由于 $s'_i = \sum_{j=1}^{n} a_{ij} \cdot x'_j$ 越上界，因此需要减少某个系数为正的非基变量，或增加某个系数为负的非基变量，最终使得 s'_i 刚好为 b_i，同时保持所有非基变量也不越界。若找不到，则说明原问题不可满足，返回 “不可满足”。
   (c) 最后，进行旋转操作。旋转操作的目的是交换前两步选中的基变量和非基变量的位置。方法是从 $s'_i = \sum_{j=1}^{n} a_{ij} \cdot x'_j$ 反解出被调整的非基变量 x'_k，然后代入到其他各行并化简。这步对应矩阵的一系列行变换。

三、 证明项构造

证明项的构造基于决策过程返回 “不可满足” 时的单纯形表进行。

为了说明构造方法的可行性，下面先证决策过程结束时单纯形表的几个性质：

定理 (性质 1). *决策过程返回 “不可满足” 时，单纯形表的所有非基变量都是松弛变量。*

证明. 反设决策过程返回 “不可满足” 时，存在某一非基变量 x'_j 不是松弛变量。则 x'_j 是原始变量，x'_j 取值无限制。因此能够通过减小或增大 x'_j 的取值，使得越界基变量刚好不越界。这与返回 “不可满足” 矛盾。 □

定理 (性质 2). *决策过程返回 “不可满足” 时，单纯形表中越界基变量所在行的所有系数为非正。*

证明. 反设决策过程返回 “不可满足” 时，越界基变量行存在某一系数 $a_{ij} > 0$。由性质 1 知，x'_j 为松弛变量，故无下界。因此可以减小 x'_j，使得基变量刚好不越界。这与返回 “不可满足” 矛盾。 □

有了上述几个性质的准备，现在可以说明证明项的构造方法。
设越界变量为 s'_i，$s'_i = \sum_{j=1}^{n} a_{ij} \cdot x'_j$，$s'_i \leq b_i$。
由性质 1 可保证，对 $1 \leq j \leq n$，x'_j 有上界，即 $x'_j \leq b'_j$。
由性质 2 可保证，对 $1 \leq j \leq n$，$a_{ij} \leq 0$，即 $a_{ij} \cdot x'_j \geq a_{ij} \cdot b'_j$。
因此 $s'_i = \sum_{j=1}^{n} a_{ij} \cdot x'_j \geq \sum_{j=1}^{n} a_{ij} \cdot b'_j > b_i$，与 $s'_i \leq b_i$ 得到矛盾。

上述过程除了计算之外，需在 Coq 中定义如下引理：

```
volatile : forall x a : Z, x <= a /\ x > a -> False .
```

应用上述引理即可构造出证明项。

第三节 分支限界法

单纯形法解决的是 $\mathbb{Q}$ 上的线性不等式组的可满足性问题。但是对于一些约束条件，如 $4 \leq 3 \cdot x \leq 5$，在 $\mathbb{Q}$ 上可满足，但在我们讨论的 $\mathbb{Z}$ 上是不可满足的。分支限界法用来解决本问题。

一、 决策过程

分支限界法主要通过逐步通过分割可行域，逐步减小搜索范围的办法，将 $\mathbb{Q}$ 上的判定过程转换为 $\mathbb{Z}$ 上的判定过程。

1. 首先以原始的不等式组作为输入，调用单纯形法决策过程。
2. 根据单纯形法返回值：
   (a) 若为 “不可满足”，则 $\mathbb{Z}$ 上也不可满足，返回 “不可满足”；
   (b) 若为 “可满足”，且可满足实例所有分量为整数，则返回 “可满足”；
   (c) 若为 “可满足”，且可满足实例某一分量 x 为分数 $\frac{a}{b}$，则分割为两部分：一部分补充不等式 $x \leq \lfloor \frac{a}{b} \rfloor$，另一部分补充不等式 $x \geq \lceil \frac{a}{b} \rceil$，然后分别递归求解。

二、 证明项构造

这里证明项的构造是简单的。只要刻画分支的行为即可。

定义如下 Coq 引理：

```
branch : forall x a : Z,
        (x <= a -> False) -> (x > a -> False) -> False.
```

即可。

第六章　理论整合框架的实现

理论整合框架的作用是，将一个含有多种理论的命题分解成一组命题，两者的可满足性是相同的，且命题组的每一个命题只含某一个理论所能处理的符号。

第一节　Nelson-Oppen 框架

Nelson-Oppen 框架 [11] 是 Nelson 和 Oppen 提出的解决理论整合的方案。它的主要步骤如下：

纯化 通过引入新变量，将混合的理论分开。

如命题 $x = 1 \wedge f(2) = 1 \wedge \neg f(+(1, x)) = 1$ 经过纯化后，变成 $x = u \wedge f(v) = u \wedge \neg f(w) = u$ 和 $u = 1 \wedge v = 2 \wedge w = +(1, x)$，前者属于等词与未解释函数理论，后者属于线性整数理论。

证明 将纯化公式送往对应理论求解器求解。若某个理论返回 “不可满足”，则返回 “不可满足”。

等式传播 分如下情况：

若某理论能够推出等式 $x = y$，将 $x = y$ 加入其他理论并重启证明。

若某个理论能够推出 $x_1 = y_1 \vee \cdots \vee x_k = y_k$，则分成 k 种情况，依次将 $x_1 = y_1, \ldots, x_k = y_k$ 加入其他理论并证明。若某一种情况返回 “可满足”，则返回可满足；否则返回 “不可满足”。

若无法推出新的等式，返回 “可满足”。

第二节　实现

Nelson-Oppen 框架实现的难点在于等式传播，而其他部分的实现都是直接的。

等式传播的主要问题是，前述的两种理论求解器都只能证实结论，却不能主动推出结论。

如对于命题 $x \leq 1 \wedge 1 \leq x \rightarrow f(x) = f(1)$，经过否定并规范化后得到 $x \leq 1 \wedge 1 \leq x \wedge \neg f(x) = f(1)$。我们的期望是 $x \leq 1 \wedge 1 \leq x$ 送给线性整数求解器得到 $x = 1$，然后将等式传播给未解释函数求解器，就能得到不可满足的结论。

本文试图采用*推出式*的方法，即改造线性整数求解器，使之能够在某些可满足的情形下附加一组等式，提供闭合可行域中的所有取值，然后一一送给其他求解器求解。这种改造方法在一些特殊的例子下工作得很好，甚至能够解决形如 $1 \leq x \wedge x \leq 3 \wedge f(1) = 1 \wedge f(2) = 2 \wedge f(3) = 3 \rightarrow f(x) = x$ 这样的“归纳型”命题。

然而，处理命题 $x \leq y \wedge y \leq x \rightarrow f(x) = f(y)$ 时却遇到困难。原因是条件 $x \leq y \wedge y$ 确定的区域含有无限个点。若不传播，则命题无法证明；若传播，则会进入死循环。一种办法是让未解释函数求解器也作出提示，如由 $\neg f(x) = f(y)$ 猜测线性整数证明器应该证明 $x = y$。这种方法目前正在继续尝试。

另外还存在*穷举式*的方法，由材料 [12] 给出。它会穷举所有中间结论，并一一尝试证明，因此能够解决推出式的方法。这种方法不需要对理论证明器做修改，模块性较好。这种方法相对于推出式，不能利用具体理论的特性，因此预期性能会下降。

第三节　证明项的生成

Nelson-Oppen 框架涉及证明项求解的部分是命题的纯化部分。

仍以

$$x = 1 \wedge f(2) = 1 \wedge \neg f(+(1, x)) = 1$$

为例说明如何生成证明项。

上述命题在纯化之后，实际证明的是

$$x = u \wedge f(v) = u \wedge \neg f(w) = u \wedge u = 1 \wedge v = 2 \wedge w = +(1, x) \rightarrow false$$

下面构造

$$x = 1 \wedge f(2) = 1 \wedge \neg f(+(1, x)) = 1 \rightarrow$$
$$x = u \wedge f(v) = u \wedge \neg f(w) = u \wedge u = 1 \wedge v = 2 \wedge w = +(1, x)$$

的证明项，然后应用 $\rightarrow$ 的传递特征，就能得到

$$x = 1 \wedge f(2) = 1 \wedge \neg f(+(1, x)) = 1 \rightarrow false$$

的证明项。

而证明项可由下述 Coq 的引理构造：

```
eq_ind : forall (A : Type) (x : A) (P : A -> Prop),
               P x -> forall y : A, x = y -> P y .
```

这就完成了证明项的构造。

第七章　分离逻辑命题的证明

第一节　语言

$$
\begin{array}{rcl}
 & & T \text{ 是公式项} \\
 & & \Pi \text{ 是纯一阶逻辑公式} \\
H & ::= & T \mapsto T \mid \text{emp} \\
\Sigma & ::= & H \mid \Sigma * \Sigma \\
P & ::= & \Pi \wedge \Sigma \to \Sigma'
\end{array}
$$

第二章已经介绍，分离逻辑的证明器将命题语言拆分为纯一阶逻辑公式和带分离逻辑的公式的过程。本章只考虑拆分后带分离逻辑的公式的证明。

第二节　决策过程

分离逻辑命题的证明方法是直接从蕴含式着手正向证明。证明依据的基本原理是分离逻辑中的 Frame 规则：

$$P \to Q \vdash P * R \to Q * R$$

具体的过程如下：

消前后件 emp emp 代表的是空堆，对证明没有帮助，因此可先消去。

如果中途得到了 emp → emp，则直接宣告命题成立。

堆匹配 堆匹配是一个递归搜索的过程。

设命题为 $P_1 * P_2 * \cdots * P_m \to Q_1 * Q_2 * \cdots Q_n$

- 若 $m \neq n$，返回 “不可证”；
- 若 $m = n = 0$，返回 “可证”；
- 否则，将 P_1 依次与 $Q_k, 1 \leq k \leq n$ 匹配，即求证：

$$\Pi \vdash t_1 = t_1'$$
$$\Pi \vdash t_2 = t_2'$$

其中 P_1 为 $t_1 \mapsto t_2$，Q_k 为 $t_1' \mapsto t_2'$。

待证两式均为纯的一阶逻辑公式，可以用一阶逻辑证明器证明。

若有其一不可证，说明 P_1 与 Q_k 不匹配，继续匹配直到 $k > n$，返回 “不可证”。

若均得证，则删除 P_1 和 Q_k，递归。

第三节　证明项

一、　分离逻辑在 Coq 中的表达

原生的 Coq 没有定义分离逻辑及其符号。但由于 Coq 系统极强的表达能力，可以自行定义新逻辑。

1　方案

在 Coq 中定义新逻辑有两类方案。

一类方案是定义语义。即将各符号的含义在 Coq 中表达出来，然后通过这些含义，证明对应的性质。这种方案的好处是，不会凭空引入过多公理，结构比较清晰。

另一类方案是定义语法。即不定义符号的含义，直接将性质用公理的形式表达出来。这种方法引入的公理比较多，不适当地公理会引起全系统的不一致。

本文采用定义语法的方案。

2 符号

分离逻辑引入了三个符号：$*$、$\mapsto$、emp，分别是逻辑联接词、函数符号、命题。定义如下：

```
Parameter star : Prop -> Prop -> Prop.
Parameter mapsto : Z -> Z.
Parameter emp : Prop.
```

3 公理

$*$ 的性质：

```
Axiom star_comm : forall P Q : Prop, star P Q -> star Q P.
Axiom star_assoc : forall P Q R : Prop,
                star P (star Q R) -> star (star P Q) R.
Axiom star_mapsto : forall t1 t2 t3 t4,
                star (mapsto t1 = t2) (mapsto t3 = t4) -> t1 <> t3.
```

$\mapsto$ 的性质是未解释函数的性质加上：

```
Axiom mapsto_0 : forall t1 t2, mapsto t1 = t2 -> t1 <> 0.
```

emp 的性质：

```
Axiom star_emp : forall P : Prop, star P emp -> P.
Axiom star_emp_r : forall P : Prop, P -> star P emp.
```

Frame 规则：

```
Axiom star_frame : forall P Q R:Prop, (P->Q) -> (star P R -> star Q R).
```

二、 证明项的构造

证明项构造分三步：

串联堆块 前面对匹配的堆块 $t_1 \mapsto t_2$ 和 $t_1' \mapsto t_2'$ 已经得到：

$$\Pi \vdash t_1 = t_1'$$
$$\Pi \vdash t_2 = t_2'$$

应用未解释函数的引理，可得到 $t_1 \mapsto t_2 \to t_1' \mapsto t_2'$。

然后依次应用 `star_frame` 公理，可以得到如下形式的定理的证明：

$$P_1' * P_2' * \cdots * P_{m'}' \to Q_1' * Q_2' * \cdots * Q_{n'}'$$

加 emp 应用 `star_emp`、`star_emp_r` 公理，可以在上一步的基础上加入 emp。

重排 应用 `star_comm` 和 `star_assoc` 公理，可以写一个递归的过程，将前后件中的各部分排到需要的位置。

这样，就得到了 $P_1 * P_2 * \cdots * P_m \to Q_1 * Q_2 * \cdots Q_n$ 的证明项。

第八章　实现与性能测试

第一节　实现

根据前面的设计，本文实现了一个定理自动证明器。由于时间所限，并没有实现所有功能。该证明器的特性如下：

- 采用 C 语言编写。
- 实现了 SMT 结构的证明器，支持等词与未解释函数理论、线性整数理论。
- 理论整合部分采用推出式[1]，部分形式的命题会有不停机的问题。
- 支持 Coq 兼容的证明项生成。但理论证明部分，证明项生成没有完全实现，部分证明项生成生成工作由 Coq 的 tactic 代理。
- 未实现分离逻辑命题证明部分。

由于该证明器未实现分离逻辑，因此还不能叫做分离逻辑的定理证明器。但是，由于分离逻辑部分主要是将命题转化为一阶逻辑命题求证，因此对于验证设计中可信性、自动化的目标，以及评价性能，仍然具有意义。

第二节　性能测试

一、　测试对象

我们选择 Z3 和 Coq 与本文的实现作对比。其中，本文的实现与 Z3 都是 SMT 结构，而 Z3 是实现最好的 SMT 证明器之一，故可作速度上限。而本文

[1] 见第六章的实现部分。

的实现在判定命题时，应比 Coq 的判定快，否则便失去了在 Coq 外部做证明的意义。

测试的指标有两个：*判定时间*和*证明项编译时间*。前者是指命题输入到宣告命题成立所需要的时间，后者是指 Coq 兼容的证明项输出被 Coq 的编译器检查所需要的时间。

对于本文的实现，两个测试指标都是有意义的。而对 Z3 来说，因为没有证明项输出，因此只有判定时间。对 Coq 来说，判定与证明项的编译是同时发生的。因此，以下对 Z3 和 Coq 的两种时间不作区分，都只称时间。

二、 测试数据的选取

测试数据的选取有两个原则：使用适度的规模和保证代表性。

使用适度的规模的测试数据，是指数据规模既不能太小，也不能太大。数据规模太小，三个系统运行的时间都很短，时间不易测准；而规模太大，Coq 在判定或编译证明项时容易超时超内存。

保证代表性，是指各种理论都有测试数据覆盖到。注意到 Coq 只能对纯理论的命题做自动判定，而不支持混合理论的命题。故还要有纯理论的例子出现，以测试 Coq 的固有能力。对于混合理论命题，则先由人工给出 Coq 证明脚本，再测试时间。

根据以上原则，本文一共设计了六组测试数据，分别编号为 1 到 6。其简要描述如表8.1：

表 8.1 测试数据的描述

编号	测试内容	Coq 自动化策略
1	命题逻辑，常见重言式	`tauto`
2	命题逻辑，5 变量 CNF 范式	`tauto`
3	命题逻辑，8 变量 3-SAT 问题	`tauto`
4	纯线性整数理论	`omega`
5	等词与未解释函数理论	`congruence`
6	混合理论	人工编写脚本

三、 测试结果

测试采用如下平台：

- CPU: Intel Core i3 330M
- Mem: 4G
- Kernel: Linux 3.8.8-x86_64
- Compiler: GCC 4.7.2, OCAML 4.00.1
- Z3 4.3.1, Coq 8.4p1

测试结果如表8.2，其中时间精度为 0.01 秒，有效位数 2 位。

表 8.2 测试结果，单位：秒

编号	本文判定	Coq	Z3	本文编译
1	0.01	0.06	<0.01	27
2	<0.01	0.32	<0.01	0.14
3	0.05	超内存	0.01	110
4	0.03	0.17	0.01	3.9
5	<0.01	0.01	<0.01	0.75
6	0.02	0.86	0.01	2.3

四、 讨论

若用本文实现的判定时间与 Coq 和 Z3 的时间做对比，则介于两种证明器之间，本文实现的速度较 Coq 具有一定优势。特别是对于 SAT 问题，Coq 的 `tauto` 策略往往在规模稍大时就会超时超内存，而本文的实现则能够证明。另外，Z3 在证明上述数据时，几乎都能在最小时间单位内证出。这说明本文的实现还有较大提升空间。

但是考虑到 Coq 的时间实际上包括判定时间和证明项编译时间，则本文的实现情况还很不理想。

最坏时间出现在数据 1，本文的实现总时间是 Coq 的几百倍。经过对生成证明项的考察，得到如下原因：

1. 数据 1 的蕴含式较多，在证明式规范到 CNF 时出现了 98 个 CNF 子句，这些子句的证明项占用了总证明项的 98%，使 Coq 编译时负担较大。
2. SAT 求解时，迟迟得不到矛盾，造成生成的证明项函数嵌套层数过多、体积过大。Coq 对于这种类型的证明项验证较慢，消耗内存很多，因此减慢了速度。

数据 4 慢的原因是，当前对线性整数理论的证明项生成通过 Coq 的 `omega` 实现。这在证明器完善后将有效缓解。

数据 6 慢的原因是，本文自动生成的证明项比不过人工编写的脚本。

第三节　结论

本章的实现与测试说明：

1. 本文设计的证明器具有可行性，单纯判定命题时的性能指标优于 Coq，并且比 Coq 的自动化程度更高。
2. 本文实现的证明器在生成证明项方面还不够好，其编译时间过长。解决这个问题，需要由以下方面着手：
   (a) 优化决策过程。虽然判定的性能指标基本能够满足要求，但是未优化的决策过程搜索过多，导致证明项庞大。
   (b) 避免生成 Coq 不擅长的证明项类型。目前观察到，对过多嵌套的函数，Coq 在编译时容易超时超内存。而对连续的结构较简单的引理，Coq 编译速度较快。
   (c) 进一步浓缩证明项。可考虑在证明项生成后，加入浓缩处理，如删除冗余、合并公共表达式等。而这可以通过借鉴编译技术的相关内容解决。

第九章　总结

第一节　工作总结

本文介绍了一个分离逻辑的定理证明器的设计，其特点是：

- 基于 SMT 结构，支持等词与未解释函数理论、线性整数理论，初步满足程序验证的需要；
- 相比 Z3 等自动证明系统，能够验证基本的分离逻辑定理，并且能够输出 Coq 兼容的证明项；
- 相比 Coq 等交互式证明系统，具有完全自动化、判定速度快的优势。

其中，为了输出 Coq 兼容的证明项，本文详细地研究了 SMT 定理证明器每个组件的结构，并针对其特点给出了证明项的构造方法。最终做到了定理证明的每一步都有证明项输出，从而提高了证明的可信度。

根据本文中的设计，本文还初步实现了设计中一阶逻辑的证明部分。通过测试，初步验证了设计的可行性。测试中，证明器能够全自动地证明一阶逻辑命题，并且自动构造 Coq 兼容的证明项，验证了其可信、自动的特点。

第二节　进一步的工作

进一步的工作可从三方面展开：

- 完善决策过程。由于时间所限，目前实现的决策过程都比较简陋，效率还不够高。因此，可以进一步完善现有实现，或者实现效率更高的决策过程。

- 改进证明项生成。测试表明，目前的证明项的生成算法用 Coq 编译时速度不理想，部分测试例生成的证明项过大过复杂。第八章的结论部分探讨了改进的方法。
- 完成分离逻辑证明器。实现方面，目前分离逻辑部分的证明器还没有做；设计方面，可以进一步考虑加入树、表等归纳谓词，使得证明器实用。

参考文献

[1] C. A. R. Hoare. An axiomatic basis for computer programming. *COMMUNICATIONS OF THE ACM*, 12(10):576–580, 1969.

[2] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 106–119, New York, NY, USA, 1997. ACM. ISBN 0-89791-853-3. doi: 10.1145/263699.263712. URL `http://doi.acm.org/10.1145/263699.263712`.

[3] George Ciprian Necula. Compiling with proofs. Technical report, 1998.

[4] John Reynolds. Separation logic: A logic for shared mutable data structures. pages 55–74. IEEE Computer Society, 2002.

[5] Z3. Website. URL `http://z3.codeplex.com/`.

[6] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78799-2, 978-3-540-78799-0. URL `http://dl.acm.org/citation.cfm?id=1792734.1792766`.

[7] Coq. Website. URL `http://coq.inria.fr/`.

[8] Smallfoot. Website. URL `http://www.cs.ucl.ac.uk/staff/p.ohearn/smallfoot/`.

[9] William Howard. The formulae-as-types notion of construction. In Curry et al. [15], pages 479–490. ISBN 0123490502.

[10] Greg Nelson, Derek, and C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27:356–364, 1980.

[11] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, October 1979. ISSN 0164-0925. doi: 10.1145/357073.357079. URL `http://doi.acm.org/10.1145/357073.357079`.

[12] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, New York, NY, USA, 1st edition, 2009. ISBN 0521899575, 9780521899574.

[13] Josh Berdine, Cristiano Calcagno, and Peter W. O' hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *In International Symposium on Formal Methods for Components and Objects*, pages 115–137. Springer, 2005.

[14] Josh Berdine, Cristiano Calcagno, and Peter O'Hearn. A decidable fragment of separation logic. In *In FSTTCS*, pages 97–109. Springer, 2004.

[15] Haskell Curry, Jonathan Seldin, and Roger Hindley, editors. *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 1980. Academic Press. ISBN 0123490502.

[16] Yves Bertot and Pierre Castéran. Interactive theorem proving and program development. coq'art: The calculus of inductive constructions, 2004. URL `http://www.labri.fr/publications/l3a/2004/BC04`.

[17] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008. ISBN 3540741046, 9783540741046.

[18] 汪芳庭. *离散数学之三: 数理逻辑*. 中国科学技术大学出版社, 合肥, 1990.

[19] 杨思敏, 李兆鹏, 庄重, and 张臻婷. 出具证明编译器中线性整数命题证明的自动生成. 小型微型计算机系统, (06):1175–1181, 2011.