中国科学技术大学

硕士学位论文

SPARCv8 汇编程序

形式化验证

作者姓名： 查君鹏

学科专业： 计算机软件与理论

导师姓名： 冯新宇 教授

完成时间： 二〇一九年五月三十日

University of Science and Technology of China

A dissertation for master's degree

Formal Verification of SPARCv8 Assembly Code

Author: Junpeng Zha

Speciality: Computer software and theory

Supervisor: Prof. Xinyu Feng

Finished time: May 30, 2019

中国科学技术大学学位论文原创性声明

本人声明所呈交的学位论文，是本人在导师指导下进行研究工作所取得的成果。除已特别加以标注和致谢的地方外，论文中不包含任何他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的贡献均已在论文中作了明确的说明。

作者签名：_____ 签字日期：_____

中国科学技术大学学位论文授权使用声明

作为申请学位的条件之一，学位论文著作权拥有者授权中国科学技术大学拥有学位论文的部分使用权，即：学校有权按有关规定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，可以将学位论文编入《中国学位论文全文数据库》等有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。本人提交的电子文档的内容和纸质论文的内容相一致。

保密的学位论文在解密后也遵守此规定。

☑ 公开 ☐ 保密（____ 年）

作者签名：_____ 导师签名：_____

签字日期：_____ 签字日期：_____

摘　要

系统软件是计算机系统的核心部分，其安全性和可靠性是构建高可信计算机系统的关键。而形式化验证技术基于严格的数学理论和方法，能够为系统软件的正确性提供强有力的保证。在系统软件中，内嵌汇编代码常被用于实现与底层硬件平台之间的交互。汇编代码的安全性和正确性对于保证整个系统的正确性是非常重要的。在某航天嵌入式操作系统的形式化验证工作项目中，该操作系统的内嵌汇编程序是由 SPARCv8 指令集编写的，而项目前期工作所采用的验证框架，无法验证这部分内嵌汇编程序的正确性。为了完善这一部分工作，本文基于已有的形式化验证技术提出了一套用于 SPARCv8 汇编程序验证的方法，并做出了如下贡献：

首先，本文为 SPARCv8 汇编程序设计了一种霍尔风格的验证逻辑。并为验证逻辑中的推理规则提供了直接风格的（direct-style）、基于语义的解释。该验证逻辑支持 SPARCv8 汇编程序中函数调用的模块化验证，以及 SPARCv8 的三个特性的验证：包括延时跳转、特殊寄存器的延时写入和寄存器窗口机制。

其次，应用所设计的验证逻辑，本文验证了某航天嵌入式操作系统中任务上下文切换程序的主要功能模块，累计共验证了约 250 行 SPARCv8 汇编程序代码。验证逻辑的定义，可靠性证明以及任务上下文切换程序的验证工作都在 Coq 辅助定理证明工具中实现。

最后，本文提出了一种 SPARCv8 汇编程序与其所对应的抽象汇编原语之间的模拟关系，并证明了如果 SPARCv8 汇编程序与其所对应的汇编原语之间满足该模拟关系，那么他们之间满足上下文精化关系，即调用具体 SPARCv8 代码的程序，不会比调用抽象汇编原语的程序产生更多的行为。我们使用该方法证明了一个 SPARCv8 实现的任务上下文切换程序和任务调度原语 `switch` 之间满足该模拟关系。

关键词：SPARCv8；汇编程序验证；霍尔逻辑；任务上下文切换；Coq；精化验证

ABSTRACT

System software is the core part of the computer system. Its safety and reliability are important to build high confidence system. Formal verification is based on rigorous mathematical theories and methods, and can provide a strong guarantee for the correctness of system software. In system software, inline assembly code is usually used to interact with low-level hardware platform. So, safety and correctness of the assembly code is crucial to guarantee the safety of the whole system. In our current formal verification project of an aerospace embedded operating system, the operating system's inline assembly codes are implemented by SPARCv8 . However, the verification framework used previously cannot verify the correctness of this part of assembly code. In order to improve this part of work, this paper proposes a method for formal verificaiton of SPARCv8 assembly code. It is based on the existing technology about formal verification, and makes the following contributions :

First, we design a Hoare-style program logic for SPARCv8 , and give direct-style semantic interperation for the logic judgement. Our logic supports modular verification of function call, and three main features of SPARCv8 : delayed control transfer, delayed write and register windows.

Second, we use our logic to verify the main body of context switch routine in a realistics embedded OS kernal for aerospace crafts, which consists of around 250 lines of SPARCv8 code. The definition of our logic, soundness proof and verification work for context switch routine are all mechanized in Coq.

Finally, we propose a simulation relation between the SPARCv8 program and its corresponding abstract assembly primitive. We prove that if the SPARCv8 program and an assembly primitive satisfy this simulation relation, then they satisfy the contextual refinement, which means the programs call this SPARCv8 program will not produce more behaviors than the program call this assembly primitive. We use this method to verify that a context switch routine implemented by SPARCv8 satisfies this simulation relation with the `switch` primitive.

Key Words: SPARCv8 ; assembly program verification; Hoare logic; context switch routine; Coq; refinement verification

目　录

第 1 章　绪　　论

1.1　研究动机

如今，计算机系统已在国防、通讯、金融、交通、医疗等关键领域中得到广泛应用，构建高可信软件已成为世界范围的重要课题。软件系统中的任何一个微小的错误都有可能导致整个计算机系统的崩溃。目前，国内外由于软件缺陷而导致的灾难和事故屡见不鲜。例如不久前，埃塞俄比亚航空公司的一架波音 737 MAX 8/9 由于传感器将错误数据传给飞控系统，导致飞控系统错误降低机头，致使飞机最终坠毁。而操作系统，作为软件系统的核心和基础，其安全性和可靠性更是构建高可信计算机系统的关键，通过形式化验证技术能够为操作系统的安全性和可靠性提供强有力的保证。

操作系统作为应用软件和底层硬件系统的中间层，存在一些和具体硬件平台交互的程序模块，例如任务上下文切换模块以及中断管理模块。这部分代码由于涉及到大量和寄存器相关的操作，所以通常采用汇编代码编写。由此可见，验证汇编代码的正确性是操作系统内核形式化验证工作中必不可少的一部分。不过当前的一些操作系统验证工作中，内嵌汇编代码要么没有被验证[1-2]，要么缺少一套通用的验证逻辑而采用操作语义进行验证[3]。所以对于操作系统中内嵌汇编代码的验证工作仍然存在一定的挑战。

笔者所在实验室目前正承担着某航天嵌入式操作系统的验证项目，该项目的目标是使用笔者所在实验室研发的抢占式操作系统验证框架[4]，验证该航天嵌入式操作系统的正确性。该航天嵌入式操作系统中与硬件平台交互的模块采用 SPARCv8 指令集[5] 编写。但操作系统验证框架中并没有提供验证具体汇编代码的方法，而是定义了一些抽象的原子操作来替代具体的汇编程序。而 SPARCv8 指令集由于其本身的一些特性，也给我们的程序编写以及形式化验证工作带来一定的挑战。这些特性可以总结为：

- 延时跳转特性

  不同于 x86，ARM 等指令集， SPARCv8 有两个程序指针，程序指针 pc 和下一个程序指针 npc。程序指针 pc 指向当前正在执行的指令，而 npc 指向下一条将要被执行的指令。 SPARCv8 中的的控制转移指令将 npc 指向目标程序点，而 pc 则被赋予了原来 npc 的值。这导致了 pc 在一个指令周期后才会指向跳转目标，从而使得控制转移被延时了一个周期。

- 特殊寄存器的延时写入特性

  SPARCv8 中存在一类特殊寄存器，我们可以使用 wr 指令来修改这些寄存器的值。不过，写入操作并不会立即生效，而是会被延时 X 个指令周期之

后再生效。这里的 X 是一个预定义的 0 到 3 之间的系统参数。

- 寄存器窗口机制

  SPARCv8 使用寄存器窗口和窗口旋转机制来实现程序上下文管理，这样可以避免将当前方法的运行时上下文直接保存到内存的栈空间，从而提高程序的执行效率。

综上所述，目前的操作系统验证框架缺乏对内嵌 SPARCv8 汇编程序验证的支持，而 SPARCv8 的三个主要特性，以及与 C 语言等高级语言相比，汇编代码结构性较差的问题，都会为我们的验证工作带来一定的挑战。这些，都是本文要重点讨论的内容。

1.2　国内外相关工作

目前，国内外对于 SPARC 指令集的形式化验证工作相对较少，南洋理工大学的 Zhe Hou 等人[6] 在 Isabelle/HOL 中对 SPARC 指令集进行了形式化建模。而 Wang 等人[7] 则在 Coq 辅助定义证明工具[8] 上对 SPARCv8 指令集操作语义进行了形式化建模，并利用操作语义进行一些验证工作。但利用操作语义模拟程序执行来进行验证, 受限于程序的执行步数，进一步的验证工作还需要定义相关的程序逻辑来完善。

虽然关于 SPARCv8 的研究工作不多，但国内外目前已经有不少关于汇编代码和机器码的验证工作。但大部分工作[9-11] 都不支持函数调用的验证或者只是简单地将函数调用指令视为普通的跳转指令，将函数返回指令视为高阶代码指针。这些工作主要关注于汇编程序的安全性，却不能满足我们验证内嵌汇编程序功能性的需求。SCAP[12] 中提供了一种验证汇编代码中函数调用和返回的方法，本文验证 SPARCv8 中函数调用和返回指令的方法正是借鉴了 SCAP 中的思想。不过 SCAP 的工作是基于一个小的汇编代码子集上进行的，如何扩充到具有具体特性的 SPARCv8 汇编程序上，也是我们的研究工作之一。另外，作为 Foundational Proof-Carrying Code（FPCC）[13] 项目中的一部分，Tan 和 Appel 提出了一种用于验证汇编代码控制转移的验证逻辑 $\mathcal{L}_c$[10]，虽然他们的验证逻辑是实现在 SPARC 机器语言上的，但如何解决 SPARC 的延时跳转、特殊寄存器延时写入以及窗口机制并不是 $\mathcal{L}_c$ 逻辑所关注的重点，并且 $\mathcal{L}_c$ 也无法验证程序的功能正确性。

很多汇编代码的验证技术目前也已运用到实际的汇编代码验证工作中，例如：Proof-Carrying Code（PCC）[11] 和 Typed Assembly Language（TAL）[14-15] 等工作主要支持 x86 程序的验证。Myreen 则提出了一个 ARM 程序的验证框架[16] 基于真实的 ARM 指令集。Yang 和 Hawblitzel 等人[17] 验证了一个 x86 指令实现的

实验性操作系统 Verve。Verve 分为两层：高层是通过 TAL[15] 实现的，而底层称为“Nucleus”用于提供对硬件和内存的原子访问。Nucleus 中的代码是通过自动定理证明工具 Z3 SMT solver 验证的，而我们的工作是希望扩展和完善已有的操作系统验证框架，通过验证逻辑验证操作系统内嵌汇编代码的正确性，并最终生成机器可检查的证明。另外，相较于 x86 和 ARM， SPARCv8 指令集由于具有延时跳转、特殊寄存器的延时写入、以及寄存器窗口机制等特性的存在，相关的验证工作的难度也会更大。

Ni 等人[18] 验证了一个 x86 指令实现的 19 行的任务上下文切换程序，他们工作的目的是希望将此作为一个例子来展示 XCAP[19] 验证框架可以支持验证存在内嵌代码指针（embedded code pointers, ECP）的程序。我们的工作验证了一个约 250 行的来源于真实操作系统内核的任务上下文切换程序，该任务上下文切换程序的实现涉及到内部函数的调用，以及 SPARCv8 的三个特性：延时跳转、特殊寄存器延时写入以及寄存器窗口机制，验证难度更大。不过我们的验证逻辑并不支持返回地址的切换，我们将此工作放在精化验证中解决。另外，Guo 等人[20] 提出了一种模块化验证任务上下文切换函数的方法，并证明了如果调用抽象 `switch` 指令的程序是安全的，那么调用具体任务切换函数的程序也是安全的，不过，该方法要求寄存器状态只能被保存在固定的内存空间（即任务的任务控制块 TCB 中），而 SPARCv8 的寄存器状态中因为涉及到寄存器窗口，在进行任务上下文切换时，需要将临时保存在寄存器窗口中的运行时上下文保存到内存栈中，而任务的栈空间在程序运行过程中会发生变化的，所以本文第6章工作中所考虑的情况则更为复杂。Gu 等人[21] 在 CCAL 框架下验证了一个 x86 实现的任务上下文切换程序和任务调度原语 `cswitch` 之间满足上下文精化关系，他们的方法主要针对的是自己所设计的 x86 指令实现的任务上下文切换程序，因为文章中没有给出具体的实现细节，暂不清楚该方法能否运用到 SPARCv8 程序实现的任务上下文切换程序的验证中。

1.3　本文贡献

针对第1.1节所提出的研究动机与需求，以及1.2 节中所指出了国内外研究工作的不足，本文主要做了如下贡献，我们结合图 1.1对每一项贡献做详细说明。

首先，为了验证操作系统内核中内嵌 SPARCv8 汇编代码的正确性，我们为 SPARCv8 汇编程序设计了一套验证逻辑（对应于图1.1中的 ①），该验证逻辑能够支持 SPARCv8 汇编程序的三个主要特性的验证以及函数调用的模块化验证。为了支持验证 SPARCv8 的三个主要特性，我们分别针做了如下工作：

- 延时跳转特性：我们重新定义了基本代码块，将 `jmp` 和 `retl` 等延时跳转指

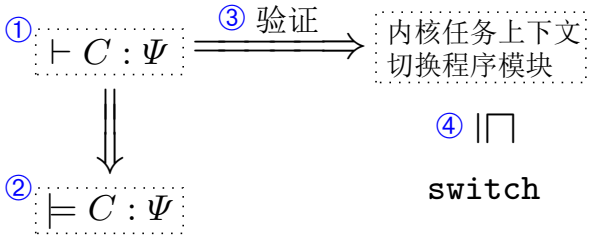

图 1.1　本文工作展示

令及其直接后继指令一起视为一个基本代码块的结尾来解决 SPARCv8 中的延时跳转问题。

- 特殊寄存器的延时写入特性：为了解决 SPARCv8 中特殊寄存器延时写入的问题，我们定义了一种新的断言 $\triangleright_t \mathtt{sr} \mapsto w$ 来描述特殊寄存器的状态，这个断言表示特殊寄存器的值 `sr` 将会在至多 t 个周期之后为 w。
- 寄存器窗口机制：SPARCv8 主要通过 `save` 和 `restore` 指令来操作寄存器窗口来实现高效的函数执行上下文管理。我们为 `save` 和 `restore` 指令设计了相应的推理规则来支持 SPARCv8 中的寄存器窗口机制。

此外，为了支持模块化地验证 SPARCv8 中的函数调用，我们借鉴了 SCAP[12] 中提出的验证汇编程序中函数调用的方法。不过， SPARCv8 中的函数调用指令 `call` 是延时跳转指令，所以还需要结合我们之前对 SPARCv8 中延时跳转特性的处理。另外，SCAP 的工作中使用一个保证 g 来描述程序的功能，而本文的工作则选择了传统霍尔逻辑中前断言和后断言的方式来刻画程序规范，目的是为了复用已有的一些技术（例如 Coq 证明策略[22-24]）来简化我们的验证工作。

其次，我们赋予了验证规则直接风格的（direct-style）基于语义的解释，将函数返回指令视为当前函数结束的标志，并在此基础上建立了验证逻辑的可靠性（对应于图1.1中的 ②）。而之前相关工作中在进行验证逻辑的可靠性证明时，要么采用基于语法的可靠性证明（例如：SCAP[12]），要么将函数返回指针视为高阶代码指针，并赋予函数返回指令后继传递风格（Continuation-passing-style，CPS-style）的语义（例如：$\mathcal{L}_c$[10]），这些建立验证逻辑可靠性的方法很难适用于内嵌汇编代码和 C 代码交互的内核代码验证工作中，因为我们使用的操作系统验证框架[1] 验证操作系统内核时，C 语言的验证工作是以函数为基本单位，并采用使用直接风格（direct-style）语义的验证逻辑。

第三，我们用我们的验证逻辑，验证了某航天嵌入式操作系统中任务上下文切换程序的主要功能模块（对应于图1.1中的 ③），任务上下文切换程序在多任务操作系统内核中扮演着至关重要的角色，因为需要操作寄存器和栈，所以它通常是由与具体硬件平台相关的汇编代码编写的。 SPARCv8 中因为存在寄存器窗口

机制，在进行任务上下文切换过程中不仅需要保存当前寄存器的值，还需要保存已使用的寄存器窗口中的内容，所以，使用 SPARCv8 实现的任务上下文切换程序比使用其它汇编指令的实现要复杂，验证的工作量也更大，本文工作累计验证了约 250 行 SPARCv8 汇编程序，累计编写了 6690 行 Coq 证明脚本。

最后，因为操作系统验证框架[4] 中使用抽象的 `switch` 原语来替代具体的任务上下文切换程序，所以作为相关工作的补充和完善，本文提出了一种能够保证 SPARCv8 汇编程序与其所对应的汇编原语之间上下文精化关系的一种模拟关系，保证了调用具体 SPARCv8 代码的程序，不会比调用抽象汇编原语的程序产生更多的行为。我们证明了一个 SPARCv8 实现的任务上下文切换程序与抽象汇编原语 `switch` 之间满足该模拟关系[①]（对应于图1.1中的 ④）。

1.4　本文组织结构

- 第2章介绍了 SPARCv8 指令集以及基本的程序形式化验证基础等相关的背景技术；
- 第3章给出了 SPARCv8 指令集的形式化建模，包括 SPARCv8 汇编程序的语法和语义；
- 第4章介绍我们为 SPARCv8 设计的验证逻辑以及在 Coq 辅助定理证明工具中的实现。验证逻辑包括：断言语言，推理规则，以及验证逻辑可靠性的定义和证明。其中，对于 SPARCv8 函数调用的模块化验证，以及 SPARCv8 三个主要特性的验证：延时跳转、特殊寄存器延时写入和寄存器窗口机制的支持都体现在推理规则的设计中；
- 第5章介绍应用我们的验证逻辑验证某航天嵌入式操作系统的任务上下文切换程序主要功能模块的工作，包括：所验证程序的程序结构和功能的介绍，程序规范的形式化定义，以及相关的 Coq 实现。
- 第6章介绍如何证明汇编原语的汇编代码实现与汇编原语之间存在上下文精化关系。我们在这一章定义了高层调用抽象汇编原语的 C 语言和底层 SPARCv8 程序，和能够保证 SPARCv8 汇编程序与其所对应的汇编原语之间上下文精化关系的一种模拟关系，并证明了一个 SPARCv8 实现的任务上下文切换程序与抽象汇编原语 `switch` 之间满足该模拟关系。

[①]我们这里证明的是一个简化版的任务上下文切换程序和汇编原语之间满足我们定义的模拟关系，不过该简化版的任务上下文切换程序保留了原内核中任务上下文切换程序的主要功能，证明原操作系统内核中的任务上下文切换程序和 `switch` 抽象原语之间满足定义的模拟关系的工作会在未来进行完善。

第 2 章　相关背景知识介绍

本章我们将简要介绍一下本文工作中所涉及到的相关技术和一些背景知识。我们将在第2.1节介绍一下本文所关注的 SPARCv8 指令集，并以一个例子来详细介绍 SPARCv8 的三个特性：延时跳转、特殊寄存器的延时写入以及寄存器窗口机制。然后，在第2.2节和第2.3 节介绍一下文本工作的理论基础霍尔逻辑，以及一个支持汇编程序模块化验证的验证框架 SCAP。另外，本章在第2.4节介绍了一些与精化验证相关的知识，包括什么是程序之间的精化关系，我们为什么需要关心程序间的精化关系，以及一些基本的精化验证技术及其实际运用。最后，在第2.5节简要介绍一下定理证明工具 Coq。

2.1　SPARCv8 指令集

SPARC[25]，即可扩充处理器架构（Scalable Processor ARChitecture），是国际上流行的 RISC 处理器体系结构之一。SPARC v7/v8 是目前嵌入式控制系统常用的处理器标准版本，并在各种处理器、工作站以及嵌入式系统中得到广泛的应用。 SPARCv8 拥有一些独特的机制来获得更好的性能、更低的功耗以及编程的灵活性[5]。如：为了支持更灵活的函数跳转，而引入的多种控制转移指令；为了提高函数调用时上下文的保存和恢复的效率，而引入的窗口寄存器和窗口旋转机制；写特殊寄存器时，写入操作的延时写特性等等。本文重点关注 SPARCv8 的三个主要特性：延时跳转、特殊寄存器的延时写入和寄存器窗口机制。我们用图2.1中的例子来说明这三个特性。

```
CALLER：                          ChangeY：
     …                             5    rd  Y, %l0
1    mov  1, %o0                   6    wr  %i0, 0, Y
2    call  ChangeY                 7    nop
3    save  %sp, –64, %sp           8    nop
4    mov  %o0, %l0                 9    nop
     …                             10   ret
                                   11   restore  %l0, 0, %o0
```

图 2.1　SPARCv8 代码的一个例子

图2.1中包括两个函数：CALLER 和 ChangeY 。CALLER 会调用函数 ChangeY ，而 ChangeY 函数负责修改特殊寄存器 Y 的值，并返回 Y 原来的值。ChangeY

函数需要输入一个参数来作为特殊寄存器 Y 的新值，这里我们在第 1 行将值 1 赋予寄存器 $\%o_0$，$\%o_0$ 负责函数调用时参数的传递。

在程序的第 2 行，CALLER 会调用函数 ChangeY，此时，程序指针 pc 指向第 2 行，而 npc 指向第 3 行。执行指令 call 会将 npc 的值赋予 pc，而让 npc 指向目标函数 ChangeY 的起始地址第 5 行。这意味着控制流将不会在执行完 call 指令之后立即发生转移，而是发生在下一个指令周期，即执行完第 3 行的 save 指令之后。同理，当 ChangeY 函数返回时（第 10 行），在执行完 ret 指令之后，控制流也不会立即返回函数调用者，而是同样会被延时一个指令周期，即执行完后继指令 restore（第 11 行）之后再发生转移，我们将此特性称为“延时跳转”。

SPARCv8 使用 save（图2.1中的第三行）保存当前过程的上下文，使用指令 restore（第 11 行）来恢复上一个过程的上下文。不过， SPARCv8 并不是直接将寄存器的值保存到内存栈中，而是通过窗口旋转机制来实现高效的运行时上下文管理。 SPARCv8 中的 32 个特殊寄存器在逻辑上被划分成四组：global（$r_0 \sim r_7$），out（$r_8 \sim r_{15}$）, local（$r_{16} \sim r_{23}$）和 in（$r_{24} \sim r_{31}$）寄存器。为了方便描述，我们给这些寄存器组起了相应的别名：“$\%g_0 \sim \%g_7$”，“$\%o_0 \sim \%o_7$”，“$\%l_0 \sim \%l_7$”和“$\%i_0 \sim \%i_7$”。这四组寄存器中 out，local 和 in 寄存器组成了当前寄存器窗口，其中 local 寄存器是当前上下文所私有的，而 in 和 out 寄存器则是和相邻的窗口共享的，为了方便参数的传递。save 指令的执行会旋转寄存器窗口，将当前窗口旋转到下一个窗口，此时，原来当前窗口的 local 和 in 寄存器将无法被访问，而原本的 out 寄存器变成新的当前窗口的 in 寄存器。同理，restore 指令执行相反的操作，恢复之前窗口的 local 和 in 寄存器。我们称此特性为“寄存器窗口机制”。

在第 6 行，指令 wr 尝试将值 $\%i_0 \oplus 0$（$\oplus$ 表示异或运算）赋予特殊寄存器 Y。但是，这个写操作并不会立即生效，而是会被延时 X 个指令周期，这里 X 是一个 0 到 3 之间的系统常量。考虑到代码的可移植性，程序员在编程的过程中通常并不依赖于延时 X 的具体值，而是假设延时周期为最大值 3。因此，在 7~9 行我们插入了 3 个 nop 指令。如果我们在第九行之前读取 Y 寄存器的值，那么结果是不确定的（与具体的延时 X 有关），而第 9 行之后，被延时的写入操作生效。我们称此特性为“特殊寄存器的延时写入特性”。

2.2　霍尔逻辑

霍尔逻辑[26] 是程序形式化验证技术的基础理论，虽然传统的软件测试效率更高，自动化程度更好，但测试的结果其实并不可靠。测试的结果依赖与测试中

所使用的测试用例，测试用例越完善，覆盖度越高，就越可能找出程序中存在的错误。但通常情况，希望测试用例的覆盖度达到 100% 几乎不可能。相比于软件测试，使用霍尔逻辑验证程序的正确性基于严格的数学推理，严格证明所验证的程序满足其所对应的程序规范，能够为程序的正确性提供更严格的保证。

霍尔逻辑使用霍尔三元组（Hoare Tripe）来描述程序的正确性，霍尔三元组的形式如下：

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

其中 P 和 Q 都是断言（assertion）, 用来刻画程序的状态，通常采用一阶谓词逻辑表示，这里的 C 则表示我们的程序，即一段指令序列。我们通常称 P 为前断言，用来描述程序执行前的状态，而将 Q 称为后断言，用来表示程序执行结束后的状态应满足什么样的条件。更直白一点来说，霍尔三元组表示的是：当程序从某一状态 s 开始执行，并且 s 满足前断言 P，那么程序执行结束之后，假设结束时的程序状态为 s'，那么末状态 s' 将满足后断言 Q。

霍尔逻辑为程序验证提供了一套推理规则，例如下面的赋值语句规则（ASSN rule）和顺序规则（CONSEQ rule）：

$$\frac{}{\{Q\,[x \mapsto a]\}\ x = a\ \{Q\}}\ \text{(ASSN)} \qquad \frac{\{P\}\ C_1\ \{R\} \quad \{R\}\ C_2\ \{Q\}}{\{P\}\ C_1;\ C_2\ \{Q\}}\ \text{(CONSEQ)}$$

我们可以使用推理规则来形式化的验证程序满足它所对应的程序规范，例如我们可以如下程序中如果起始状态 $x = 2$ 并且 $y = 3$，那么执行完如下程序之后，x 和 y 的值被交换，即：$x = 3$ 并且 $y = 2$。

$$\{x = 2 \wedge y = 3\}\ r = x;\ x = y;\ y = r\ \{x = 3 \wedge y = 2\}$$

我们可以通过应用赋值语句规则（ASSN rule）和顺序规则（CONSEQ rule）来证明，证明过程如下：

1. $\{x = 2 \wedge y = 3\}\ r = x\ \{r = 2 \wedge y = 3\}$ ASSN
2. $\{r = 2 \wedge y = 3\}\ x = y\ \{r = 2 \wedge x = 3\}$ ASSN
3. $\{r = 2 \wedge x = 3\}\ y = r\ \{x = 3 \wedge y = 2\}$ ASSN
4. $\{x = 2 \wedge y = 3\}\ r = x;\ x = y\ \{r = 2 \wedge x = 3\}$ CONSEQ 1, 2
5. $\{x = 2 \wedge y = 3\}\ r = x;\ x = y;\ y = r\ \{x = 3 \wedge y = 2\}$ CONSEQ 4, 3

在后续的形式化验证研究中，人们在霍尔逻辑的基础上继续对其进行扩展和完善，使其具有更强的表达能力，例如分离逻辑[27] 扩展了霍尔逻辑对指针程序的验证以及局部推理的支持。

2.3　SCAP

本文为 SPARCv8 所设计验证逻辑的思路来源于之前的 SCAP[12] 工作，SCAP 是对原有的汇编代码验证框架 CAP[9] 的扩展。在 CAP 验证框架中，代码堆中的汇编程序被视为一个个基本代码块，我们可以对这些代码块分别进行验证。但是，CAP 主要关注的是被验证程序的安全性，它将函数调用指令视为普通的跳转指令，将函数返回指针视为高阶代码指针，从而无法实现汇编函数的模块化验证，也无法验证汇编程序的功能正确性。

由于汇编代码的结构较差，使得我们很难去跟踪一个函数或异常处理程序的范围，而 SCAP 验证框架提供了一种能够模块化验证汇编代码中函数调用的方法，它将函数调用的 `jal` 指令视为函数调用的起始点，而将 `jr` 指令视为函数调用的返回点，这样就赋予了结构性较差的汇编代码程序一个逻辑上的函数概念。SCAP 使用一个二元关系 g 来描述函数的功能。下面是 SCAP 中提供的函数调用规则（CALL rule）和函数返回规则（RET rule），我们以此为例来说明 SCAP 的核心思想：

$$\frac{\begin{array}{l}\mathtt{f}, \mathtt{f}_{\mathrm{ret}} \in \mathrm{dom}(\Psi) \quad (p', g') = \Psi(\mathtt{f}) \quad (p'', g'') = \Psi(\mathtt{f}_{\mathrm{ret}}) \\ \forall H, R.\ p\ (H, R) \to p'\ (H, R\{\$\mathtt{ra} \rightsquigarrow \mathtt{f}_{\mathrm{ret}}\}) \\ \forall H, R, S'.\ p\ (H, R) \to g'\ (H, R\{\$\mathtt{ra} \rightsquigarrow \mathtt{f}_{\mathrm{ret}}\})\ S' \to \\ \qquad (p''\ S' \wedge (\forall S''.\ g'\ S'\ S'' \to g\ (H, R)\ S'')) \\ \forall S, S'.\ g'\ S\ S' \to S.R(\$\mathtt{ra}) = S'.R(\$\mathtt{ra})\end{array}}{\Psi \vdash \{(p, g)\}\ \mathtt{jal}\ \mathtt{f}, \mathtt{f}_{\mathrm{ret}}}\ (\mathrm{CALL})$$

$$\frac{\forall S.\ p\ S \to g\ S\ S}{\Psi \vdash \{(p, g)\}\ \mathtt{jr}\ \$\mathtt{ra}}\ (\mathrm{RET})$$

在函数调用规则（CALL rule）中，`jal` 指令调用函数 `f`, 并将返回地址 $\mathtt{f}_{ret}$ 保存在寄存器 `$ra` 中，函数调用规则保证了当前函数在调用函数 `f` 结束之后仍然可以继续执行。这里的问题在于如何保证函数 `f` 调用结束之后程序会返回到正确的地址 $\mathtt{f}_{\mathrm{ret}}$ 继续执行？这里函数调用规则对其所调用的函数 `f` 的程序规范做了约束，性质“$\forall S, S'.\ g'\ S\ S' \to S.R(\$\mathtt{ra}) = S'.R(\$\mathtt{ra})$”保证了函数 `f` 执行结束时，寄存器 `$ra` 中保存的返回地址和发生函数调用时一样，继而保证了函数调用结束后会返回到正确的程序点继续执行。

而这里的函数返回规则（RET rule）则非常简单，它并不需要检查返回地址是否是一个有效地址，因为函数调用规则（CALL rule）已经保证了所调用的函数可以返回正确的程序点，并继续执行。本文为 SPARCv8 设计的验证逻辑中对函数调用的模块化验证的支持也借鉴了 SCAP 中的思想。

2.4　精化验证

精化验证是程序验证的一个重要分支。程序精化关系是建立在两个程序之间的关系。简单来说，如果程序 C 是对程序 $\mathbb{C}$ 的精化，那么程序 C 所产生的行为是程序 $\mathbb{C}$ 的子集，由于程序 C 不会产生 $\mathbb{C}$ 不具有的行为，因此我们在任何使用程序 $\mathbb{C}$ 的地方，都可以使用程序 C 来替代。例如，我们不关心局部变量 t 的值时，下述代码段可以看做对 "$x++$" 的精化。

$$\texttt{local t; t} := x; x := \texttt{t} + 1$$

通过上述例子我们可以发现，"$x++$" 抽象层次更高，更为简单，屏蔽了一些我们不关心的实现细节。所以在具体的验证过程中，使用 "$x++$" 可以简化我们的验证工作，这也解释了为什么我们的操作系统验证工作中[1] 中使用汇编原语来替代具体的汇编实现，目的是简化验证工作。当然，我们需要证明具体的汇编代码实现是其所对应原子操作的精化，来保证程序中任何使用原子操作的地方，都可以用其所对应的汇编代码实现来替代。精化验证已经在程序验证领域得到了很多实际运行，例如：编译器的验证工作 CompCert[28]。常用的证明程序之间精化关系的手段是定义程序之间的模拟关系。不过，早期 CompCert 中所定义的模拟关系是建立在完整串行程序上的，所以该模拟关系无法处理多程序模块并发执行的程序的情况。

为了支持并发程序和多模块程序的精化验证，Liang 等人[29] 在 2012 年提出了一种基于依赖/保证的模拟关系 RGSim（Rely-Guarantee-based Simulation）, 作为并发程序精化的通用验证技术。RGSim 以依赖/保证条件为参数，描述了任务和并发环境之间的影响。RGSim 将多任务程序的精化证明分解为单个任务上的精化证明，具有并发环境下的可组合性。该方法已经在多线程抢占式操作系统的验证工作[4] 中得到实际运用，并且同样适用于处理多个程序模块之间交互的情况（例如：Compositional Compcert[30]）。

2.5　Coq 辅助定理证明工具

Coq[8] 是一个定理证明辅助工具，提供交互式的方式进行定理的证明。Coq 中的规范说明语言 *Gallina* 可以描述程序设计语言中常用的类型和程序，我们可以使用它来表达规范说明（specification），开发满足规范的程序。Coq 系统还提供了一种表达能力很强的逻辑，通常被称为高阶逻辑（high-order logic），可以用来开发证明。本文所提出的验证逻辑以及任务上下文切换程序主要功能模块的验证，都在 Coq 辅助定义证明工具中实现。

第 3 章　SPARCv8 指令集的形式化建模

我们将在本章重点介绍 SPARCv8 中一些关键的指令，以及机器状态和操作语义的形式化定义。

3.1　语法和程序状态

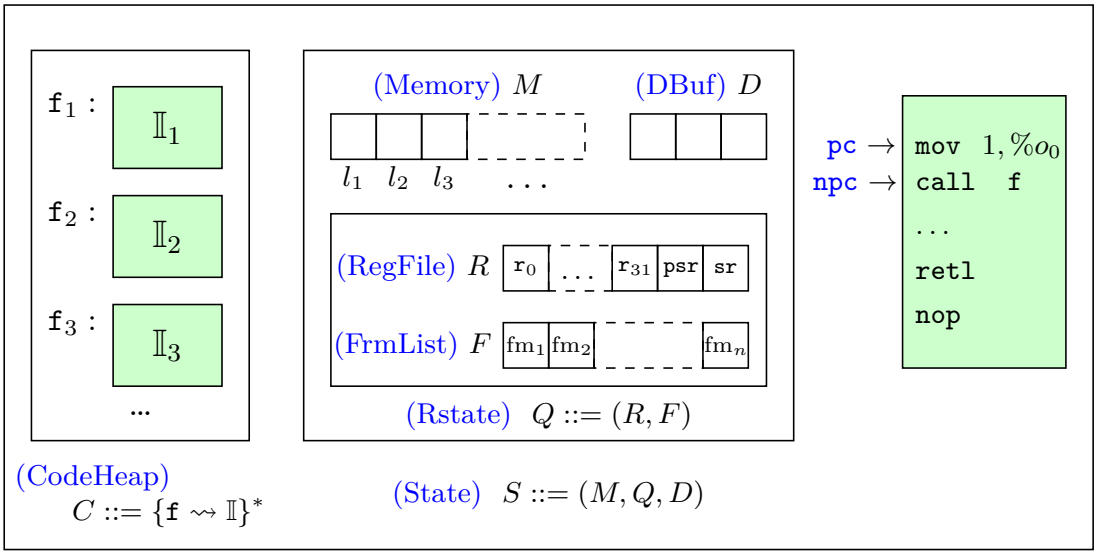

图 3.1　SPARCv8 程序机器模型

图3.1展示了我们对 SPARCv8 程序形式化建模的主体结构[①]，并在图3.2中给出了 SPARCv8 汇编语言的的语法和机器状态的形式化定义。程序配置 ***P*** 包括：代码堆 ***C***，机器状态 ***S***，程序指针 `pc` 和 `npc`。代码堆 ***C*** 是一个从标签 `f` 到 SPARCv8 指令 Comm 的部分映射[②]，这里我们将数值 w，标签 `f` 和地址 l 都称为“字”，类型是 32 位整型。标签 `f` 可以理解为代码堆中指令存储的位置，而地址 l 则理解为内存中数值存储的位置。我们用 c 来表示 SPARCv8 指令，这里，我们将指令分为两类：简单指令 `i` 以及控制转移指令（例如：`call` 和 `jmp`）。

机器状态 ***S*** 被定义为一个三元组：内存 ***M***，寄存器状态 ***Q***，以及延时缓存 ***D***。内存 ***M*** 的定义是一个从地址 l 到数值 w 的局部映射，这里地址 l 和 w 的类型都 32 位整型。寄存器状态 ***Q*** 是一个二元组，包括：寄存器文件 ***R*** 和帧链表

[①]图3.1中因为空间关系，我们将 FrameList 简写为 FrmList，将 Delay Buffer 简写为 DBuf。

[②]图3.2 将代码堆表示为标签到指令序列的映射，我们在后续介绍指令序列时，会介绍如何基于当前的代码堆定义由一个给定的起始标签从代码堆中获取一个指令序列。

$$
\begin{array}{rrcl}
\text{(Word)} & w, \mathtt{f}, l & \in & \text{Int32} \\
\text{(Prog)} & \mathrm{P} & ::= & (\boldsymbol{C}, \boldsymbol{S}, \mathtt{pc}, \mathtt{npc}) \\
\text{(CodeHeap)} & \boldsymbol{C} & \in & \text{Word} \rightharpoonup \text{Comm} \\
\text{(State)} & \boldsymbol{S} & ::= & (\boldsymbol{M}, \boldsymbol{Q}, \boldsymbol{D}) \qquad \text{(RState)}\ \boldsymbol{Q} ::= (\boldsymbol{R}, \boldsymbol{F}) \\
\text{(Memory)} & \boldsymbol{M} & \in & \text{Word} \rightharpoonup \text{Word} \qquad \text{(ProgCount)}\ \mathtt{pc}, \mathtt{npc} \in \text{Word} \\
\text{(OpExp)} & \mathtt{o} & ::= & \mathtt{r} \mid w \qquad \text{(AddrExp)}\ \mathtt{a} ::= \mathtt{o} \mid \mathtt{r} + \mathtt{o} \\
\text{(Comm)} & c & ::= & \mathtt{i} \mid \mathtt{call\ f} \mid \mathtt{jmp\ a} \mid \mathtt{retl} \mid \mathtt{be\ f} \\
\text{(SimpIns)} & \mathtt{i} & ::= & \mathtt{ld\ a\ r}_d \mid \mathtt{st\ r}_s\ \mathtt{a} \mid \mathtt{save\ r}_s\ \mathtt{o\ r}_d \mid \mathtt{restore\ r}_s\ \mathtt{o\ r}_d \\
 & & \mid & \mathtt{add\ r}_s\ \mathtt{o\ r}_d \mid \mathtt{rd\ sr\ r}_d \mid \mathtt{wr\ r}_s\ \mathtt{o\ sr} \mid \mathtt{nop} \mid \cdots \\
\text{(InstrSeq)} & \mathbb{I} & ::= & \mathtt{i};\mathbb{I} \mid \mathtt{jmp\ a; i} \mid \mathtt{call\ f; i};\mathbb{I} \mid \mathtt{retl; i} \mid \mathtt{be\ f; i};\mathbb{I}
\end{array}
$$

图 3.2　SPARCv8 语言的语法和机器状态

$$
\begin{array}{rrcl}
\text{(RegFile)} & \boldsymbol{R} & \in & \text{RegName} \rightharpoonup \text{Word} \\
\text{(RegName)} & \mathtt{rn} & ::= & \mathtt{r}_0 \mid \cdots \mid \mathtt{r}_{31} \mid \mathtt{psr} \mid \mathtt{sr} \\
\text{(PsrReg)} & \mathtt{psr} & ::= & \mathtt{n} \mid \mathtt{z} \mid \mathtt{v} \mid \mathtt{c} \mid \mathtt{cwp} \\
\text{(SpeReg)} & \mathtt{sr} & ::= & \mathtt{wim} \mid \mathtt{Y} \mid \mathtt{asr}_0 \mid \cdots \mid \mathtt{asr}_{31} \\
\text{(FrameList)} & \boldsymbol{F} & ::= & \text{nil} \mid \text{fm} :: \boldsymbol{F} \qquad \text{(Frame)}\ \text{fm} ::= [w_0, \cdots, w_7] \\
\text{(DelayBuff)} & \boldsymbol{D} & ::= & \text{nil} \mid (t, \mathtt{sr}, w) :: \boldsymbol{D} \\
\text{(DelayCycle)} & t & \in & \{0, 1, \cdots, X\}
\end{array}
$$

图 3.3　寄存器文件，帧链表和延时缓存

$\boldsymbol{F}$。我们在图3.3中形式化地定义了寄存器文件和帧链表。寄存器文件 $\boldsymbol{R}$ 是一个寄存器名到数值的局部映射，而寄存器名包括：32 个通用寄存器（$\mathtt{r}_0 \sim \mathtt{r}_{31}$），处理器状态寄存器 psr，以及特殊寄存器 sr。特殊寄存器 psr 包括：条件标志位 n、z、v 和 c，以及记录当前寄存器窗口编号的 cwp。条件标志位在程序执行的过程中可以被算术和逻辑运算指令修改，主要用于程序需要条件跳转的情况。我们接下来详细介绍帧链表 $\boldsymbol{F}$ 以及延时缓存 $\boldsymbol{D}$。

寄存器窗口和帧链表　SPARCv8 拥有 32 个通用寄存器，而这 32 个通用寄存器在逻辑上被分为四组：global ($\mathtt{r}_0 \sim \mathtt{r}_7$)，out ($\mathtt{r}_8 \sim \mathtt{r}_{15}$)，local ($\mathtt{r}_{16} \sim \mathtt{r}_{23}$) 以及 in ($\mathtt{r}_{24} \sim \mathtt{r}_{31}$)。其中 out，local 和 in 寄存器组成了*当前寄存器窗口*。

当我们进入或者退出某个函数或者异常处理程序时，通常需要保存或者恢复一些通用寄存器的值作为运行时的上下文。不过， SPARCv8 并没有选择将他们直接保存到内存的栈空间，而是利用多个寄存器窗口来实现高效的上下文管理。多个寄存器窗口组成了一个循环栈，通过窗口的旋转机制来保存和恢复上下

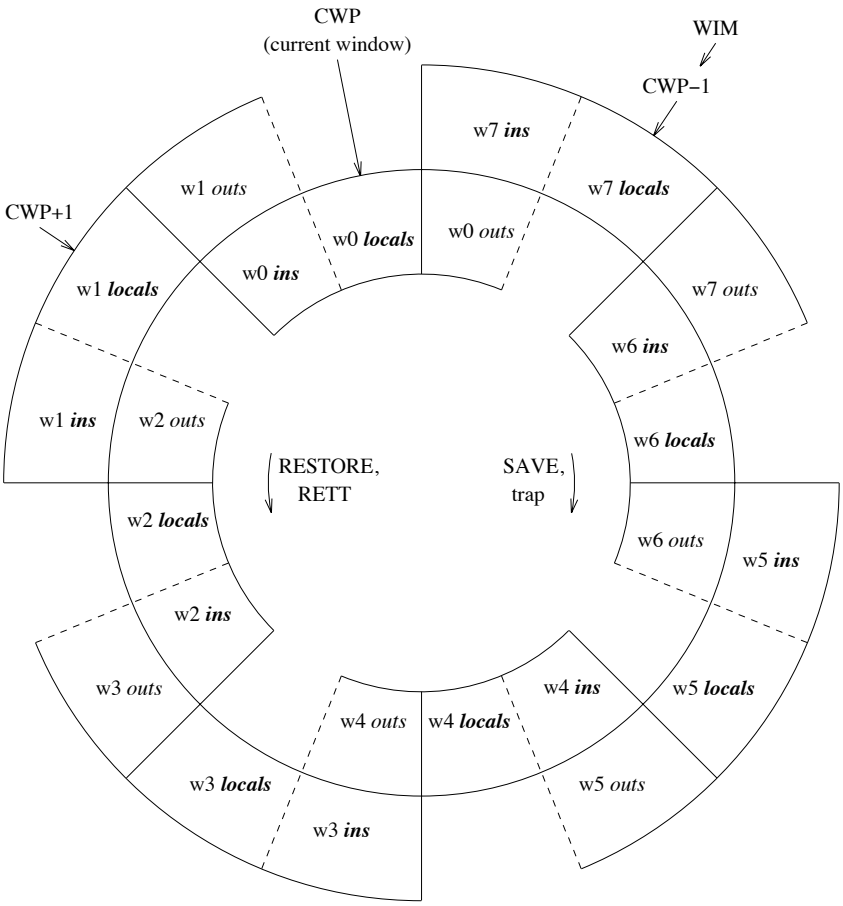

图 3.4　寄存器窗口 (图片来源于[5])

文。如图3.4所示，这里有 N 个寄存器窗口（此处 $N = 8$），包含 $2 \times N$ 个寄存器组，每个寄存器组有 8 个寄存器。处理器状态寄存器 `psr` 中的 `cwp` 字段中记录了当前窗口的编号。

每个窗口的 in 和 out 寄存器都与相邻的窗口共享，这样可以方便窗口之间的参数传递。例如，图3.4中 w_0 号窗口的 in 寄存器，也是 w_1 号窗口的 out 寄存器；而 w_0 的 out 寄存器，同时也是 w_7 号窗口的 in 寄存器。这同时也解释了为什么每个窗口有三组寄存器，可寄存器组的总数确是 $2 \times N$ 个，因为 N 个窗口中每个窗口只有 local 寄存器组是自己私有的，而 out 和 in 寄存器组都与另一个窗口共享。

当我们需要保存当前上下文时，`save` 指令的执行会旋转窗口，将 `cwp` 的值减一（模 N）。这样 `cwp` 就指向窗口 w_7，这样 w_7 就成了当前窗口。此时，w_0 的 out 寄存器变成了 w_7 窗口的 in 寄存器，而 w_0 的 in 和 local 寄存器则被移出了当前窗口，注意只有当前窗口的寄存器才能被处理器访问，所以，此时 w_0 的 in 和 local 寄存器是无法被访问的，类似被压入到循环栈的栈顶。而 `restore` 指令则执行和 `save` 指令相反的操作，类似与出栈操作。

寄存器窗口旋转机制中还涉及到 `wim` 寄存器，这个寄存器可以理解为一个

$$\mathsf{out} \overset{\text{def}}{=} [\mathtt{r}_8, \cdots, \mathtt{r}_{15}] \qquad \mathsf{local} \overset{\text{def}}{=} [\mathtt{r}_{16}, \cdots, \mathtt{r}_{23}] \qquad \mathsf{in} \overset{\text{def}}{=} [\mathtt{r}_{24}, \cdots, \mathtt{r}_{31}]$$

$$R([\mathtt{r}_i, \cdots, \mathtt{r}_{i+k}]) \overset{\text{def}}{=} [R(\mathtt{r}_i), \cdots, R(\mathtt{r}_{i+k})]$$

$$R\{[\mathtt{r}_i, \cdots, \mathtt{r}_{i+7}] \rightsquigarrow \mathrm{fm}\} \overset{\text{def}}{=} R\{\mathtt{r}_i \rightsquigarrow w_0\} \cdots \{\mathtt{r}_{i+7} \rightsquigarrow w_7\} \quad \text{where} \quad \mathrm{fm} = [w_0, \cdots, w_7]$$

$$\mathbf{win\_valid}(w_{id}, R) \overset{\text{def}}{=} 2^{w_{id}} \,\&\, R(\mathtt{wim}) = 0 \quad \text{where } \& \text{ is the bitwise AND operation.}$$

$$\mathbf{next\_cwp}(w_{id}) \overset{\text{def}}{=} (w_{id} + N - 1)\%N \qquad \mathbf{prev\_cwp}(w_{id}) \overset{\text{def}}{=} (w_{id} + 1)\%N$$

$$\mathbf{save}(R, F) \overset{\text{def}}{=} \begin{cases} (R', F') & \textit{if } w'_{id} = \mathbf{next\_cwp}(R(\mathtt{cwp})), \mathbf{win\_valid}(w'_{id}, R), \\ & F = F'' \cdot \mathrm{fm}_1 \cdot \mathrm{fm}_2,\ F' = R(\mathsf{local}) :: R(\mathsf{in}) :: F'', \\ & R'' = R\{\mathsf{in} \rightsquigarrow R(\mathsf{out}), \mathsf{local} \rightsquigarrow \mathrm{fm}_2, \mathsf{out} \rightsquigarrow \mathrm{fm}_1\}, \\ & R' = R''\{\mathtt{cwp} \rightsquigarrow w'_{id}\}, \\ \bot & \textit{if } \neg\mathbf{win\_valid}(\mathbf{next\_cwp}(R(\mathtt{cwp})), R) \end{cases}$$

$$\mathbf{restore}(R, F) \overset{\text{def}}{=} \begin{cases} (R', F') & \textit{if } w'_{id} = \mathbf{prev\_cwp}(R(\mathtt{cwp})), \mathbf{win\_valid}(w'_{id}, R), \\ & F = \mathrm{fm}_1 :: \mathrm{fm}_2 :: F'',\ F' = F'' \cdot R(\mathsf{out}) \cdot R(\mathsf{local}), \\ & R'' = R\{\mathsf{in} \rightsquigarrow \mathrm{fm}_2, \mathsf{local} \rightsquigarrow \mathrm{fm}_1, \mathsf{out} \rightsquigarrow R(\mathsf{in})\}, \\ & R' = R''\{\mathtt{cwp} \rightsquigarrow w'_{id}\}, \\ \bot & \textit{if } \neg\mathbf{win\_valid}(\mathbf{prev\_cwp}(R(\mathtt{cwp})), R) \end{cases}$$

图 3.5　`save` 和 `restore` 指令相关的辅助定义

位数组，每一位有对用于寄存器窗口中的一个窗口，用来标识该窗口是有效还是无效的，如果这一位为 1，则表示所对应窗口是无效的，如果是 0，则表示该窗口是有效的。SPARCv8 在初始状态时通常会将 `wim` 的某一位置为 1，而其它位则都是 0。被置为无效的窗口，在寄存器窗口这样一个循环栈中用于标识栈的结束位置，以防止出现栈溢出的情况。我们结合图 3.5中的相关定义做进一步说明，当执行 `save` 指令进行窗口旋转时，我们需要知道下一个窗口是否是有效的，我们通过函数 **win_valid**(w_{id}, R)（定义在图3.5中）判断当前目标窗口 w_{id} 是否是有效的（`wim` 的值从 R 中获取）。同理，执行 `restore` 指令时也需要做同样的窗口有效性的检查。

在形式化建模的工作中，我们定义了一个帧链表 F 来刻画 SPARCv8 中的寄存器窗口机制。由图3.3 帧链表（FrameList）的定义可知，帧链表是一个由帧

（Frame）组成的序列，每个帧是一个 8 个字的数组，代表了寄存器窗口中的一个寄存器组（一个寄存器组包含 8 个寄存器）。帧链表 F 中包含了寄存器窗口中除当前窗口的三个寄存器组以外的所有寄存器组。图3.5中的定义的 **save** 和 **restore** 操作展示了如何使用帧链表来模拟 SPARCv8 中的寄存器窗口旋转，这里我们以 **save** 操作为例进行说明。**save** 操作会将当前寄存器文件 R 中的 local 和 in 寄存器组放入帧链表 F 的头部，并从 F 的尾部取出最后两个寄存器组（帧）作为新窗口的 local 和 out 寄存器，而原窗口的 out 寄存器则变为了新窗口的 in 寄存器。**restore** 做相反的操作，这里不再赘述。

延时缓存　在图3.3中，我们将延时缓存 D 定义为一个延时写入操作的序列。因为 `wr` 指令并不会立即修改目标寄存器的值，所以我们会先将该写入操作放入延时缓存 D 中。我们用一个三元组 "$(t, \mathtt{sr}, w)$" 来表示被加入缓存的延时操作，其中 t 表是被延时的指令周期，`sr` 表示希望修改的目标寄存器，以及待写入的值 w。

指令序列　我们定义指令序列 $\mathbb{I}$ 来表示一个基本代码块，它是一个以控制转移指令为结束标志的指令序列。考虑到 SPARCv8 的延时跳转特性，所以在图 3.2中指令序列 $\mathbb{I}$ 的定义中，我们要求每一个延时跳转指令后面都必须包含一个后继的简单指令 `i`，因为实际的控制转移是发生在执行完延时跳转指令的后继指令 `i` 之后的。我们将 `jmp` 和 `retl` 及其直接后继指令视为指令序列结束的标志，注意，我们并没有将 `call` 指令视为一个基本代码块的结束标志，因为我们为函数调用赋予了直接风格的语言，所以我们希望被调用者（callee）执行结束之后会返回调用者（caller），并继续执行调用者（calller）的后继指令。我们在下面定义 $C[\mathtt{f}]$ 来表示如何从代码堆 C 中获取一个起始地址为 `f` 的指令序列。

$$
C[\mathtt{f}] = \begin{cases}
\mathtt{i};\mathbb{I} & C(\mathtt{f}) = \mathtt{i} \text{ and } C[\mathtt{f}+4] = \mathbb{I} \\
c;\mathtt{i} & c = C(\mathtt{f}) \text{ and } c = \mathtt{jmp\ a} \text{ or } \mathtt{retl} \\
 & \text{and } C(\mathtt{f}+4) = \mathtt{i} \\
c;\mathtt{i};\mathbb{I} & c = C(\mathtt{f}) \text{ and } c = \mathtt{call\ f} \text{ or } \mathtt{be\ f} \\
 & \text{and } C(\mathtt{f}+4) = \mathtt{i} \text{ and } C[\mathtt{f}+8] = \mathbb{I} \\
\textbf{undefined} & \text{otherwise}
\end{cases}
$$

3.2　操作语义

我们使用的 SPARCv8 的操作语义来源于之前 Wang 等人[7] 的工作，但因为我们这里没有考虑中断和异常，所以去掉了他们工作中对中断和异常的处理，另外为了方便做自动推理，我们这里将寄存器文件 R 定义为一个部分映射（parital

mapping），而没有使用全部映射（total mapping）的方式来定义寄存器文件。图3.6自顶向下展示了我们所定义的操作语义的结构。操作语义的定义共分为四层：（1）最顶层是程序的状态转移规则 $C \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto (S', \mathtt{pc}', \mathtt{npc}')$，刻画整个 SPARCv8 程序的状态转移；（2）其次，是控制流指令转移规则 $C \vdash (S, \mathtt{pc}, \mathtt{npc}) \circ\!\!\longrightarrow (S', \mathtt{pc}', \mathtt{npc}')$，例如 `call`，`jmpl` 等跳转指令的语义都是在这一层的定义中给出；（3）第三层定义了 `save` 和 `restore` 以及 `wr` 等会操作寄存器窗口和延时缓存的指令的转移规则 $S \bullet\!\xrightarrow{\mathtt{i}} S'$；（4）最后一层是简单指令的转移规则 $(M, R) \xrightarrow{\mathtt{i}} (M', R')$。

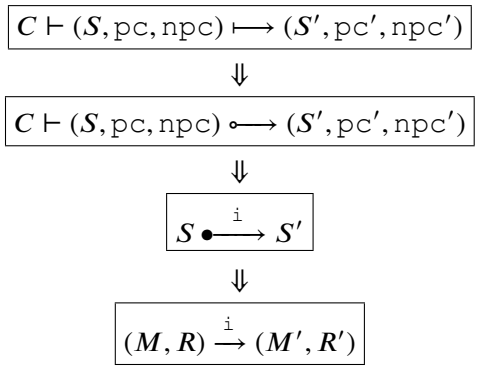

图 3.6　操作语义建模结构

$$\frac{(R, D) \rightrightarrows (R', D') \qquad C \vdash ((M, (R', F), D'), \mathtt{pc}, \mathtt{npc}) \circ\!\!\longrightarrow ((M', (R'', F'), D''), \mathtt{pc}', \mathtt{npc}')}{C \vdash ((M, (R, F), D), \mathtt{pc}, \mathtt{npc}) \longmapsto ((M', (R'', F'), D''), \mathtt{pc}', \mathtt{npc}')}$$

图 3.7　程序转移规则

我们首先在图3.7中定义最顶层的程序转移规则。从这条规则中我们可以看出，每个指令周期，在执行 `pc` 指针所指向的当前指令之前，我们会先执行和延时写入操作相关的状态转移 $(R, D) \rightrightarrows (R', D')$，具体的转移规则定义如下：

$$\frac{}{(R, \mathrm{nil}) \rightrightarrows (R, \mathrm{nil})} \qquad \frac{(R, D) \rightrightarrows (R', D')}{(R, (t{+}1, \mathtt{sr}, w) :: D) \rightrightarrows (R', (t, \mathtt{sr}, w) :: D')}$$

$$\frac{(R, D) \rightrightarrows (R', D') \qquad \mathtt{sr} \in \mathrm{dom}(R)}{(R, (0, \mathtt{sr}, w) :: D) \rightrightarrows (R'\{\mathtt{sr} \rightsquigarrow w\}, D')} \qquad \frac{(R, D) \rightrightarrows (R', D') \qquad \mathtt{sr} \notin \mathrm{dom}(R)}{(R, (0, \mathtt{sr}, w) :: D) \rightrightarrows (R', D')}$$

这些规则告诉我们，每个指令周期，延时缓存中延时为 0 的延时写入操作将会生效，延时不为 0 的写入操作，延时周期减一，并继续保留在延时缓存中。注意，如果延时写入操作的目标寄存器 `sr` 不在寄存器文件 R 的定义域中时（因为我们将 R 定义为局部映射），这个写入操作不会对寄存器文件 R 做任何修改。我们可以证明如下引理：

引理 3.1　$(R, D) \rightrightarrows (R', D')$ 并且 $R = R_1 \uplus R_2$，当且仅当存在 R_1' 和 R_2'，使得 $(R_1, D) \rightrightarrows (R_1', D')$, $(R_2, D) \rightrightarrows (R_2', D')$, 并且 $R' = R_1' \uplus R_2'$。

这里的符号“$\uplus$”表示不相交的集合并运算，所以，$R_1 \uplus R_2$ 表示寄存器文件

R_1 和寄存器文件 R_2 的并集，当 R_1 和 R_2 不相交的话，否则没有定义。引理3.1在很多与描述延时缓存相关断言的可靠性证明中扮演着重要的角色，这些我们将在第4 章介绍验证逻辑时重点讨论。

对于当前指令所产生的状态转移，我们将它分为三类进行归纳定义：控制转移（$C \vdash \_ \circ\!\!\longrightarrow \_$），save，restore 和 wr 指令转移[①]（$\_ \bullet\!\!\xrightarrow{-} \_$）；以及简单指令转移（$\_ \xrightarrow{-} \_$）。

$$\frac{C(\mathtt{pc}) = \mathtt{i} \qquad (M,(R,F),D) \bullet\!\!\xrightarrow{\mathtt{i}} (M',(R',F'),D')}{C \vdash ((M,(R,F),D),\mathtt{pc},\mathtt{npc}) \circ\!\!\longrightarrow ((M',(R',F'),D'),\mathtt{npc},\mathtt{npc}+4)}$$

$$\frac{C(\mathtt{pc}) = \mathtt{jmp\ a} \qquad [\![\mathtt{a}]\!]_R = \mathtt{f}}{C \vdash ((M,(R,F),D),\mathtt{pc},\mathtt{npc}) \circ\!\!\longrightarrow ((M,(R,F),D),\mathtt{npc},\mathtt{f})}$$

$$\frac{C(\mathtt{pc}) = \mathtt{call\ f} \qquad \mathtt{r}_{15} \in \mathrm{dom}(R)}{C \vdash ((M,(R,F),D),\mathtt{pc},\mathtt{npc}) \circ\!\!\longrightarrow ((M,(R\{\mathtt{r}_{15} \rightsquigarrow \mathtt{pc}\},F),D),\mathtt{npc},\mathtt{f})}$$

$$\frac{C(\mathtt{pc}) = \mathtt{retl} \qquad R(\mathtt{r}_{15}) = \mathtt{f}}{C \vdash ((M,(R,F),D),\mathtt{pc},\mathtt{npc}) \circ\!\!\longrightarrow ((M,(R,F),D),\mathtt{npc},\mathtt{f}+8)}$$

图 3.8　控制流转移指令转移规则

图3.8给出了控制流转移指令转移规则，首先，如果当前指令是简单指令 i（即 $C(\mathtt{pc}) = \mathtt{i}$），我们通过调用转移规则 ($\_ \bullet\!\!\xrightarrow{-} \_$) 来得到执行指令 i 所对应的程序状态转移；如果当前指令是 jmp，call 和 retl 等跳转指令，则分别定义它们所对应的程序状态转移，我们可以通过它们的转移规则来对 SPARCv8 中的延时跳转特性有了更直观的认识。我们可以发现，当执行 jmp，call 等延时跳转指令时，程序指针 pc 被置为 npc，而将目标地址赋给 npc，从而解释了为什么控制流的转移会被延时一个指令周期。执行 call 指令时，pc 的值会被保存在 $\mathtt{r}_{15}$ 寄存器中，而 retl 指令会将 $\mathtt{r}_{15}+8$ 作为函数的返回地址（调用该函数 call 指令的第二条后继指令）。

图3.9给出了 save，restore 和 wr 等指令的转移规则。首先，如果指令 i 不是 save，restore 和 wr 等指令，我们调用简单指令转移规则 ($\_ \bullet\!\!\xrightarrow{\mathtt{i}} \_$) 来得到执行指令 i 所对应的程序状态转移。对于写特殊寄存器的执行 wr 的状态转移规则，wr 指令希望将前两个操作数的计算结果按位异或之后写入特殊寄存器 sr，但是，由于 SPARCv8 中特殊寄存器的延时写入特性，这个写操作并不会立即修改寄存器文件 R，而是先将其放入延时缓存 D 中。我们定义操作

[①]注意我们在对 SPARCv8 形式化建模时，save，restore 和 wr 等指令也属于简单指令，但因为这些指令会操作寄存器窗口和延时缓存，而其它简单指令则只操作寄存器文件和内存，所以在定义操作语义时单独分出一层来定义 save，restore 和 wr 指令的操作语义。

$$\frac{(M,R) \xrightarrow{\mathtt{i}} (M',R')}{(M,(R,F),D) \bullet\!\!\xrightarrow{\mathtt{i}} (M',(R',F),D)} \qquad \frac{\begin{array}{c}R(\mathtt{r}_s)=w_1 \quad [\![\mathtt{o}]\!]_R=w_2 \quad w=w_1 \oplus w_2 \\ \mathtt{sr}\in \mathrm{dom}(R) \quad D'=\textbf{set\_delay}(\mathtt{sr},w,D)\end{array}}{(M,(R,F),D) \bullet\!\!\xrightarrow{\mathtt{wr}\ \mathtt{r}_s\ \mathtt{o}\ \mathtt{sr}} (M,(R,F),D')}$$

$$\frac{\textbf{save}(R,F)=(R',F') \quad [\![\mathtt{o}]\!]_R=w \quad R''=R'\{\mathtt{r}_d \rightsquigarrow R(\mathtt{r}_s)+w\}}{(M,(R,F),D) \bullet\!\!\xrightarrow{\mathtt{save}\ \mathtt{r}_s\ \mathtt{o}\ \mathtt{r}_d} (M,(R'',F'),D)}$$

$$\frac{\textbf{restore}(R,F)=(R',F') \quad [\![\mathtt{o}]\!]_R=w \quad R''=R'\{\mathtt{r}_d \rightsquigarrow R(\mathtt{r}_s)+w\}}{(M,(R,F),D) \bullet\!\!\xrightarrow{\mathtt{restore}\ \mathtt{r}_s\ \mathtt{o}\ \mathtt{r}_d} (M,(R'',F'),D)}$$

图 3.9　Save, Restore 和 Wr 指令转移规则

set_delay(sr, w, D) 来表示在延时缓存中新加入一个延时写入操作。其中，X 是初始延时时钟周期，这里的 X 是一个 0 到 3 之间的预定义的系统参数，表示被延时的指令周期。

$$\textbf{set\_delay}(\mathtt{sr},w,D) \stackrel{\text{def}}{=} (X,\mathtt{sr},w)::D$$

save 和 restore 指令的状态转移已经在第3.1节介绍寄存器窗口机制时做了详述，这里不再累赘。帧链表 F 和寄存器文件 R 上的 **save** 和 **restore** 操作定义在图3.5中。

$$\frac{R(\mathtt{sr})=w \quad \mathtt{r}_d\in \mathrm{dom}(R)}{(M,R) \xrightarrow{\mathtt{rd}\ \mathtt{sr}\ \mathtt{r}_d} (M,R\{\mathtt{r}_d \rightsquigarrow w\})} \qquad \frac{R(\mathtt{r}_s)=w_1 \quad [\![\mathtt{o}]\!]_R=w_2 \quad \mathtt{r}_d\in \mathrm{dom}(R)}{(M,R) \xrightarrow{\mathtt{add}\ \mathtt{r}_s\ \mathtt{o}\ \mathtt{r}_d} (M,R\{\mathtt{r}_d \rightsquigarrow w_1+w_2\})}$$

$$\frac{[\![\mathtt{a}]\!]_R=l \quad M(l)=w' \quad \mathtt{r}_d\in \mathrm{dom}(R)}{(M,R) \xrightarrow{\mathtt{ld}\ \mathtt{a}\ \mathtt{r}_d} (M,R\{\mathtt{r}_d \rightsquigarrow w'\})}$$

图 3.10　部分简单指令转移规则

我们在图3.10中给出了部分简单指令的转移规则（不包括: save，restore，wr 指令），这些指令的执行只会操作寄存器文件和内存。指令 "rd sr $\mathtt{r}_d$" 表示将特殊寄存器 sr 中的值写如通用寄存器 $\mathtt{r}_d$ 中。指令 "add $\mathtt{r}_s$ o $\mathtt{r}_d$" 将通用寄存器 $\mathtt{r}_s$ 中的值和算术表达式 o 的计算结果求和后写入通用寄存器 $\mathtt{r}_d$。指令 "ld a $\mathtt{r}_d$" 将地址 l（地址表达式 a 的计算结果为 l）中存的值 w' 写入通用寄存器 $\mathtt{r}_d$。我们在图3.11中给出了算术表达式的计算方法 $[\![\mathtt{a}]\!]_R$ 和地址表达式的计算方法 $[\![\mathtt{o}]\!]_R$。这里我们用 a 表示地址运算表达式，用 o 来表示数值运算表达式，相关定义参见图6.4。

$$
[\![\circ]\!]_R \overset{\text{def}}{=} \begin{cases} R(\mathtt{r}) & \textbf{if } \circ = r \\ w & \textbf{if } \circ = w,\ -4096 \leqslant w \leqslant 4095 \\ \bot & \textbf{otherwise} \end{cases}
$$

$$
[\![\mathtt{a}]\!]_R \overset{\text{def}}{=} \begin{cases} [\![\circ]\!]_R & \textbf{if } \mathtt{a} = \circ \\ w_1 + w_2 & \textbf{if } \mathtt{a} = \mathtt{r}+\circ,\ R(\mathtt{r}) = w_1 \textbf{ and } [\![\circ]\!]_R = w_2 \\ \bot & \textbf{otherwise} \end{cases}
$$

图 3.11　表达式语义

3.3　本章小结

我们在这一章对 SPARCv8 语言进行了形式化建模，定义了它的语法和语义，并重点介绍了如何在模型中形式化地刻画 SPARCv8 的三个特性：延时跳转、特殊寄存器的延时写入和寄存器窗口机制。这一部分工作参考了之前在 Coq 辅助定义证明工具中对 SPARCv8 指令集形式化建模的工作[7]。

第 4 章　SPARCv8 汇编程序验证逻辑

本章我们将展示我们为 SPARCv8 设计的霍尔风格的程序验证逻辑，并着重介绍我们的验证逻辑是如何支持 SPARCv8 的三个特性：延时跳转、特殊寄存器的延时写入以及寄存器窗口机制的验证。最后介绍验证逻辑可靠性的证明及 Coq 辅助定理证明工具中的实现。

4.1　断言语言

$$\begin{array}{lll} \textit{(Asrt)} \quad p, q & \stackrel{\text{def}}{=} & \mathsf{emp} \mid l \mapsto w \mid \mathtt{rn} \mapsto w \mid \triangleright_t \mathtt{sr} \mapsto w \mid p\!\downarrow \mid \mathtt{cwp} \mapsto (\!| w_{id}, F |\!) \\ & \mid & p \wedge q \mid p \vee q \mid p * q \mid \mathtt{a} =_a w \mid \mathtt{o} = w \mid \forall x.\, p \mid \exists x.\, p \mid \cdots \end{array}$$

图 4.1　断言语言语法

我们在图4.1中定义断言语言的语法，并在图4.2中给出了相应的语义。断言语言的设计基于传统的分离逻辑[27]，并在此基础上扩充了描述延时缓存和寄存器窗口的断言。之前的第3.1节中我们提到了，我们将寄存器文件 $\boldsymbol{R}$ 定义为一个部分映射（partial mapping），所以，这里我们将寄存器和内存一样都视为程序资源。断言 emp 表示内存和寄存器文件文件都为空。断言 $l \mapsto w$ 描述了一块只包含地址 l 所对应的那一块内存单元，并且该内存单元中存储的值为 w。同理，因为这里我们将寄存器文件 $\boldsymbol{R}$ 和内存一样视为资源，所以我们同样定义 $\mathtt{rn} \mapsto w$ 表示寄存器文件中只有 rn 这一个寄存器，并且存储的值为 w，同时它还表示寄存器 rn 不在延时缓存中。分离连接（separation conjunction）$p * q$ 使用的是标准分离逻辑[27] 中的语义，描述了两块不相交的内存状态，一块满足断言 p，一块满足断言 q。我们定义了两个新的断言 $\triangleright_t \mathtt{sr} \mapsto w$ 和 $\mathtt{cwp} \mapsto (\!| w_{id}, F |\!)$ 分别用来描述 SPARCv8 中的延时缓存和寄存器窗口，后面将重点介绍这两个断言。我们先通过下面这个例子开始介绍描述延时写入操作的断言。

$$\begin{array}{l} \{\%\mathtt{i}_0 = 3,\ \mathtt{Y} = 5\} \\ \quad \mathtt{wr} \quad \%\mathtt{i}_0, 0, \mathtt{Y} \\ \{\%\mathtt{i}_0 = 3,\ \mathtt{Y} = 3\ ?\mathrm{or}\ 5\ ?\} \\ \quad \mathtt{rd} \quad \mathtt{Y}, \%\mathtt{i}_1 \\ \{\%\mathtt{i}_0 = 3,\ \mathtt{Y} = 3\ ?\mathrm{or}\ 5\ ?,\ \%\mathtt{i}_1 = 3\ ?\mathrm{or}\ 5\ ?\} \\ \quad \mathtt{mov} \quad 1, \%\mathtt{i}_0 \\ \quad \mathtt{add} \quad \%\mathtt{i}_0, 1, \%\mathtt{i}_1 \\ \{\%\mathtt{i}_0 = 3,\ \mathtt{Y} = 3,\ \%\mathtt{i}_1 = 3\ ?\mathrm{or}\ 5\ ?\} \end{array}$$

$$
\begin{array}{lcl}
S \vDash \mathsf{emp} & \stackrel{\text{def}}{=} & S.M = \varnothing \wedge S.Q.R = \varnothing \\
S \vDash l \mapsto w & \stackrel{\text{def}}{=} & S.M = \{l \rightsquigarrow w\} \wedge S.M.R = \varnothing \\
S \vDash \mathtt{rn} \mapsto w & \stackrel{\text{def}}{=} & S.Q.R = \{\mathtt{rn} \rightsquigarrow w\} \wedge \mathtt{rn} \notin \mathrm{dom}(S.D) \wedge S.M = \varnothing \\
S \vDash \triangleright_t \mathtt{sr} \mapsto w & \stackrel{\text{def}}{=} & \exists k, R', D'.\, 0 \leqslant k \leqslant t{+}1 \wedge (R, D) \rightrightarrows^k (R', D') \wedge \\
 & & \quad \big((M, (R', F), D') \vDash \mathtt{sr} \mapsto w\big) \wedge \mathbf{noDup}(D, \mathtt{sr}) \\
 & & \quad \text{where } S = (M, (R, F), D) \\
S \vDash p\downarrow & \stackrel{\text{def}}{=} & \exists R', D'.\, \big((M, (R', F), D') \vDash p\big) \wedge (R', D') \rightrightarrows (R, D) \\
 & & \quad \text{where } S = (M, (R, F), D) \\
S \vDash \mathtt{cwp} \mapsto (\!| w_{id}, F |\!) & \stackrel{\text{def}}{=} & \big(S \vDash \mathtt{cwp} \mapsto w_{id}\big) \wedge \exists F'.\, F \cdot F' = S.Q.F \\
S \vDash \mathtt{a} =_a w & \stackrel{\text{def}}{=} & [\![\mathtt{a}]\!]_{S.Q.R} = w \wedge \mathbf{word\_align}(w) \\
S \vDash \mathtt{o} = w & \stackrel{\text{def}}{=} & [\![\mathtt{o}]\!]_{S.Q.R} = w \\
S \vDash p_1 * q_2 & \stackrel{\text{def}}{=} & \exists S_1, S_2.\, S_1 \vDash S_1 \wedge S_2 \vDash p_2 \wedge S = S_1 \uplus S_2
\end{array}
$$

$$
S_1 \uplus S_2 \stackrel{\text{def}}{=} \begin{cases}
(M_1 \cup M_2, (R_1 \cup R_2, F), D) & \textbf{if } M_1 \perp M_2 \wedge R_1 \perp R_2 \wedge \\
 & \quad S_1 = (M_1, (R_1, F), D) \wedge S_2 = (M_2, (R_2, F), D) \\
\textbf{undefined} & \text{otherwise}
\end{cases}
$$

$$
\mathrm{dom}(D) \stackrel{\text{def}}{=} \begin{cases}
\{\mathtt{sr}\} \cup \mathrm{dom}(D') & \textbf{if } D = (t, \mathtt{sr}, w) :: D' \\
\varnothing & \textbf{if } D = \mathrm{nil}
\end{cases}
$$

$$
\mathbf{noDup}(D, \mathtt{sr}) \stackrel{\text{def}}{=} \begin{cases}
\mathtt{sr} \notin \mathrm{dom}(D') & \textbf{if } D = (t, \mathtt{sr}, w) :: D' \\
\mathtt{sr} \neq \mathtt{sr}' \wedge \mathbf{noDup}(D', \mathtt{sr}) & \textbf{if } D = (t, \mathtt{sr}', w) :: D' \\
\textbf{True} & \textbf{if } D = \mathrm{nil}
\end{cases}
$$

$$
\mathbf{word\_align}(w) \stackrel{\text{def}}{=} w \% 4 = 0
$$

图 4.2　断言语言语义

初始状态下，寄存器 $\%\mathtt{i}_0$ 的值为 3，寄存器 Y 的值为 5，我们希望通过 wr 指令，将 $\%\mathtt{i}_0$ 寄存器中的值写入 Y 寄存器中。但由于延时写入特性，这一写入操作并不一定会立即生效。在之前的章节中我们介绍到延时周期 X（$0 \leqslant X \leqslant 3$）是一个预定义的系统参数，所以，执行完 wr 指令后，如果延时周期为 0，那么 Y 寄存器的值将被修改为 3；但如果延时周期大于 0 的话，则 Y 的值仍然为 5。由于不确定延时写入操作是否生效，所以如果我们在此时希望通过 rd 指令来读取 Y 中的值，得到的结果依然是不确定的。当然，因为延时周期的最大值为 3，所以我们可以确定，至多 3 个周期之后，延时写入将生效，Y 寄存器的值将被修改为 3。那么，如何在刻画程序状态时描述由于延时写入特性所带来的不确定性？这里我们定义断言 $\triangleright_t \mathtt{sr} \mapsto w$ 来解决。

我们用断言 $\triangleright_t \mathtt{sr} \mapsto w$ 来描述一个在延时缓存中的延时写入操作，同时它也描述了寄存器文件 $\boldsymbol{R}$ 中 sr 寄存器所对应的那一块存储单元。不过，我们并不能通过该断言的语义确定当前状态下 sr 中所存储的具体数值，但可以知道 sr 中的值将在当前指令周期后的至多第 $t+1$ 个指令周期变为 w。这里，我们使用 $\_ \Rightarrow^k \_$ 表示执行 k 步延时写入的状态转移。$\triangleright_t \mathtt{sr} \mapsto w$ 同时还约束了当前延时缓存 $\boldsymbol{D}$ 中只有一个和 sr 寄存器相关的延时写入操作，所以，我们禁止了在四个指令周期（最大延时周期为 3）之内重复写同一特殊寄存器的情况。当然，这样的约束其实并不会对实际的 SPARCv8 编程产生很大的影响，因为具体的延时周期是一个系统预定义的参数，所以程序员在编程过程中出于对程序可移植性的考虑，通常会假设延时周期是最大值 3，也不会在不确定之前的写入操作是否生效的情况下重复写同一寄存器。根据该断言的语义我们可以得到如下性质：

$$\mathtt{sr} \mapsto w \implies \triangleright_t \mathtt{sr} \mapsto w \qquad\qquad \triangleright_t \mathtt{sr} \mapsto w \implies \triangleright_{t+k} \mathtt{sr} \mapsto w$$

我们同时定义断言 $p\downarrow$ 来描述执行一步延时写入转移之后的程序状态，它表示存在一个满足断言 p 的程序状态能够通过执行一步延时写入转移之后到达当前状态。由该断言的语义我们可以有如下性质：

$$(\triangleright_0 \mathtt{sr} \mapsto w)\downarrow \implies \mathtt{sr} \mapsto w \qquad\qquad (\triangleright_{t+1} \mathtt{sr} \mapsto w)\downarrow \implies \triangleright_t \mathtt{sr} \mapsto w$$

这两个性质告诉我们，首先，如果该延时写入操作最大可能的延时周期为 0，那么经过一次延时写入转移之后，我们可以确定该延时写入将会生效，即 sr 的值已经被修改为 w。而如果延时写入操作最大可能的延时周期为 $t+1$，则经过一次延时写入转移之后，可能的延时周期将减 1。除此之外，我们还可以发现，若断言 p 中不包含形如 $\triangleright_t \mathtt{sr} \mapsto w$ 的子项，那么 p 和 $p\downarrow$ 是等价的，即：$(p\downarrow) \iff p$。

引理 4.1　$(p * q)\downarrow \iff (p\downarrow) * (q\downarrow)$。

我们可以用第3.2节的引理3.1来证明引理3.1是成立的。引理4.1 告诉我们 $(\_)\downarrow$ 是可以分别作用在分离连接 $*$ 的左右两个子项上的。以上所给出的与断言 $p\downarrow$ 和 $\triangleright_t \mathtt{sr} \mapsto w$ 相关的性质可以给我们的验证工作带来极大的方便，比如我们可以使用上述性质对如下断言做化简：

$$(\triangleright_3 \mathtt{Y} \mapsto w_1 * \triangleright_0 \mathtt{wim} \mapsto w_2 * \%\mathtt{i}_0 \mapsto w_3)\downarrow$$
$$\Downarrow$$
$$(\triangleright_3 \mathtt{Y} \mapsto w_1 * \triangleright_0 \mathtt{wim} \mapsto w_2)\downarrow * (\%\mathtt{i}_0 \mapsto w_3)\downarrow$$
$$\Downarrow$$
$$(\triangleright_3 \mathtt{Y} \mapsto w_1)\downarrow * (\triangleright_0 \mathtt{wim} \mapsto w_2)\downarrow * (\%\mathtt{i}_0 \mapsto w_3)\downarrow$$
$$\Downarrow$$
$$\triangleright_2 \mathtt{Y} \mapsto w_1 * \mathtt{wim} \mapsto w_2 * \%\mathtt{i}_0 \mapsto w_3$$

我们定义断言 $\mathtt{cwp} \mapsto (\!| w_{id}, F |\!)$ 来描述寄存器窗口的状态，这里我们使用帧链表来表示除当前窗口以外的寄存器组的内容，而当前窗口的内容则直接表示在寄存器文件中（参见第3.1节中对寄存器窗口形式化建模的详述）。该断言中的 w_{id} 是当前窗口的编号，记录在寄存器 cwp 中。注意，这里的 F 并不是表示整个帧链表，F 只是完整帧链表的一个前缀（prefix）。我们在第3.1节中曾介绍过，寄存器窗口机制类似于一个循环栈，通常情况下我们并不关心整个栈的内容，只需要要描述保存了过去运行时上下文的那一部分即可，即整个帧链表的一个前缀。举一个简单的例子，如果当前程序状态中的寄存器状态为 (R, F)，且当前窗口为 w_{id}（即：$R(\mathtt{cwp}) = w_{id}$），如果帧序列 F_1 和 F_2 都是 F 的一个前缀，那么断言 $\mathtt{cwp} \mapsto (\!| w_{id}, F_1 |\!)$ 和 $\mathtt{cwp} \mapsto (\!| w_{id}, F_2 |\!)$ 都是描述当前的寄存器状态。由该断言的语义，我们可以证明性质 $\mathtt{cwp} \mapsto (\!| w_{id}, F \cdot F' |\!) \Longrightarrow \mathtt{cwp} \mapsto (\!| w_{id}, F |\!)$ 成立，这里 $F \cdot F'$ 表示两个帧链表 F 和 F' 的连接。

断言 $\mathtt{a} =_a w$ 表示地址表达式 a 在当前状态下的求值为 w，$\mathtt{o} = w$ 表示算术表达式 o 在当前状态下的求值为 w。同时，因为 a 是地址表达式，而我们所考虑的指令和数据的字长均为 32 位（4 个字节），所以要求 a 的求值结果是四字节对齐的，即地址是 4 的整数倍（使用 **word_align** 约束）。

4.2　推理规则

定义了断言语言之后，我们在这一节开始介绍推理规则的设计，设计推理规则的目的是为了证明程序满足其所对应的程序规范。所以，我们首先从程序规范入手通过图4.3 来向读者阐释我们验证逻辑中程序规范的定义。

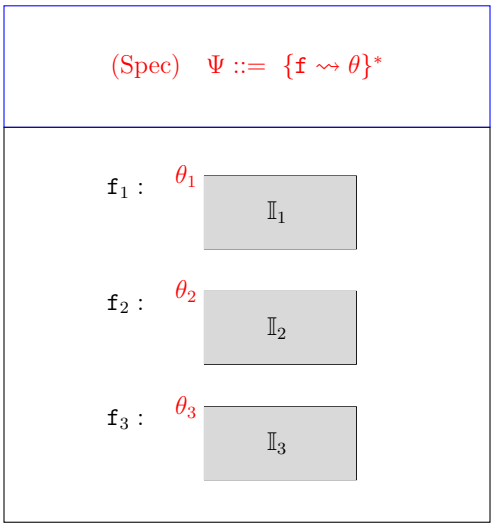

图 4.3　验证逻辑程序规范

我们在第3.1节给出了 SPARCv8 程序中指令序列（基本代码块）的定义以及从代码堆 C 中获取一个起始地址为 $\mathtt{f}$ 的指令序列的方法 $C[\mathtt{f}]$。所以，代码堆 C 也可以视为一个从地址 $\mathtt{f}$ 到以 $\mathtt{f}$ 为起始地址的指令序列的映射。我们为每个基本代码块分配对应的程序规范 θ，例如：图4.3中以 $\mathtt{f}_1$ 为起始地址的指令序列 $\mathbb{I}_1$ 的程序规范为 θ_1。这样，整个代码堆 C 的程序规范 Ψ 则可以理解为代码堆中基本代码块程序规范的集合。我们在图4.4给出验证逻辑程序规范的形式化定义。

$$
\begin{array}{llllllll}
\text{(valList)} & \iota & \in & \text{list value} & \text{(pAsrt)} & \text{fp, fq} & \in & \text{valList} \to \mathit{Asrt} \\
\text{(CdSpec)} & \theta & ::= & \text{(fp, fq)} & \text{(CdHpSpec)} & \Psi & ::= & \{\mathtt{f} \rightsquigarrow \theta\}^*
\end{array}
$$

图 4.4　程序规范

从图4.4中的定义可知，代码的程序规范集合 Ψ 将一个标签 $\mathtt{f}$ 映射到以 $\mathtt{f}$ 为起始地址的基本代码块的程序规范 θ 上，这里我们将 θ 定义为一个由前断言 fp（precondition）和后断言 fq（postcondition）组成的二元组。注意，这里前断言和后断言与我们之前定义的断言有些不同，它们带了一个参数序列 ι 作为参数，在具体的验证工作中，我们要求前断言和后断言带着相同参数序列，目的是为了通过这些相同的参数来建立起前后断言所描述状态之间的联系。

另一点值得注意的是，虽然我们为每个基本代码块都分配了一个程序规范 θ，但规范里的后断言并不是描述该基本代码块执行结束时程序的状态。相反，它描述的是该代码块*当前所在函数*执行结束时程序的状态。这一思想来源于之前的 SCAP[12] 工作，不同之处在于，我们并没有使用 SCAP 中描述前后程序状态的二元关系 g 作为程序规范，而是使用传统霍尔逻辑中前断言和后断言的方式，这么做的目的是为了复用之前一些基于霍尔三元组上的自动证明策略（例如：Coq tactic），来简化我们的验证工作。

– $\{(\text{fp}, \text{fq})\}$

```
add  %i0, %i1, %l7
add  %l7, %i2, %l7
retl
nop
```

$$
\begin{aligned}
\text{fp} &\overset{\text{def}}{=} \lambda lv.\, (\%\mathtt{i}_0 \mapsto lv[0]) * (\%\mathtt{i}_1 \mapsto lv[1]) * (\%\mathtt{i}_2 \mapsto lv[2]) \\
&\qquad * \%\mathtt{l}_7 \mapsto \_ * (\mathtt{r}_{15} \mapsto lv[3]) \\
\text{fq} &\overset{\text{def}}{=} \lambda lv.\, (\%\mathtt{i}_0 \mapsto lv[0]) * (\%\mathtt{i}_1 \mapsto lv[1]) * (\%\mathtt{i}_2 \mapsto lv[2]) \\
&\qquad * (\%\mathtt{l}_7 \mapsto lv[0] + lv[1] + lv[2]) * (\mathtt{r}_{15} \mapsto lv[3])
\end{aligned}
$$

图 4.5　一个 SPARCv8 函数规范的例子

我们通过图4.5中的例子来方便大家理解我们的程序规范，图4.5中是一个简单的 SPARCv8 函数，它将 $\%\mathtt{i}_0$，$\%\mathtt{i}_1$ 和 $\%\mathtt{i}_2$ 寄存器中的值求和后写入寄存器 $\%\mathtt{l}_7$ 中。我们定义程序规范 (fp, fq) 来描述该函数的功能。这个规范告诉我们，当我们为前断言 fp 和后断言 fq 提供相同的参数列表 lv 作为参数时，我们可以知

道，在函数结束时，寄存器 $\%\mathtt{i}_0$，$\%\mathtt{i}_1$，$\%\mathtt{i}_2$ 和 $\mathtt{r}_{15}$ 寄存器中的值不变，而寄存器 $\%\mathtt{l}_7$ 中的值为 $\%\mathtt{i}_0$、$\%\mathtt{i}_1$ 和 $\%\mathtt{i}_2$ 寄存器中值的和。$\mathtt{r}_{15}$ 在程序执行过程中负责保存函数的返回地址。我们可以发现 *lv* 在定义程序规范时主要用于建立前断言和后断言之间的联系，例如，如果没有 *lv*，我们无法描述函数执行前后，寄存器 $\mathtt{r}_{15}$ 的值不变这一事实。因为我们并不关系 *lv* 的具体内容，所以在验证一个 SPARCv8 函数满足其程序规范时，我们需要证明：对于任意的参数列表 *lv*，函数执行前后的状态满足 (fp *lv*, fq *lv*)。

确定了程序规范的形式之后，下面我们开始介绍我们为 SPARCv8 程序设计的验证推理规则。以下是我们所设计推理规则一个整体上的结构。

$$\boxed{\vdash C : \Psi} \qquad \textbf{(Well-Formed Code Heap)}$$

$$\Downarrow$$

$$\boxed{\Psi \vdash \{(p,q)\}\ \mathtt{f} : \mathbb{I}} \qquad \textbf{(Well-Formed Instruction Sequences)}$$

$$\Downarrow$$

$$\boxed{\vdash \{p\}\ \mathtt{i}\ \{q\}} \qquad \textbf{(Well-Formed Instructions)}$$

可以看到我们的推理规则设计分为三层：1. 验证整个代码堆的代码堆推理规则（**Well-Formed Code Heap**）；2. 验证基本代码块的指令序列推理规则（**Well-Formed Instruction Sequences**）；3. 验证单条简单指令的指令推理规则（**Well-Formed Instructions**）。我们依次对他们进行详细介绍。

4.2.1　代码堆推理规则

$$\frac{\text{for all } \mathtt{f} \in \mathrm{dom}(\Psi),\ \iota :\ \Psi(\mathtt{f}) = (\mathrm{fp}, \mathrm{fq}) \quad \Psi \vdash \{(\mathrm{fp}\ \iota, \mathrm{fq}\ \iota)\}\ \mathtt{f} :\ C[\mathtt{f}]}{\vdash C : \Psi}\ \textbf{(CDHP)}$$

图 4.6　代码堆验证规则

图4.6中是我们验证逻辑的顶层规则 **CDHP** 规则，我们使用该验证逻辑来验证代码堆 C。他要求对于代码堆 C 中的任意一个基本代码块，如果代码的程序规范 Ψ 中有它的程序规范，那么该基本代码块可以被验证是满足它的程序规范的，这里我们用 ι 表示程序规范中前后断言的参数列表。因为参数列表只是起到建立前后断言所描述状态之间联系的作用，相当于辅助变量，我们并不关心它的具体内容，所以这里不需要对它做任何具体的限制。

4.2.2　指令序列的推理规则

我们将在这一小节介绍指令序列的推理规则，其中包括了我们是如何支持 SPARCv8 延时跳转特性的验证以及函数调用的模块化验证。

$$\frac{\vdash \{p\downarrow\}\ \mathtt{i}\ \{p'\} \qquad \Psi \vdash \{(p',q)\}\ \mathtt{f}+4 : \mathbb{I}}{\Psi \vdash \{(p,q)\}\ \mathtt{f} : \mathtt{i};\mathbb{I}}\ \textbf{(SEQ)}$$

图 4.7　顺序推理规则

图4.7中展示的是我们的顺序推理规则（SEQ rule），该推理规则主要应用在当前指令为简单指令 i 的指令序列验证中。我们说一个以简单指令 i 开头的指令序列是良性的（Well-Formed），那么首先，指令 i 应该可以通过我们的指令推理规则的验证，并且后继指令序列 $\mathbb{I}$ 仍然可以通过指令序列推理规则的验证。注意，虽然验证当前指令序列的前断言是 p，但在验证首指令 i 时，前断言却是 $p\downarrow$。原因可以通过图3.7中给出的程序状态转移来理解，因为对于 SPARCv8 程序执行的每一步，在执行当前指令之前，首先会执行一步延时写入状态转移（$\_ \Rightarrow \_$）。由 $p\downarrow$ 断言的语义我们很容易证明：如果执行延时写入状态转移前程序状态满足 p，那么执行完延时写入状态转移后，程序状态满足 $p\downarrow$。

$$\frac{\begin{array}{c} p\downarrow \Rightarrow (\mathtt{a} =_a \mathtt{f}') \quad \mathtt{f}' \in \mathrm{dom}(\Psi) \quad \Psi(\mathtt{f}') = (\mathrm{fp}, \mathrm{fq}) \\ \vdash \{p\downarrow\downarrow\}\ \mathtt{i}\ \{p'\} \quad \exists \iota, p_r.\ (p' \Rightarrow \mathrm{fp}\ \iota * p_r) \wedge (\mathrm{fq}\ \iota * p_r \Rightarrow q) \end{array}}{\Psi \vdash \{(p,q)\}\ \mathtt{f} : \mathtt{jmp\ a};\ \mathtt{i}}\ \textbf{(JMP)}$$

图 4.8　跳转指令推理规则

延时跳转特性的处理　这里我们将跳转指令 jmp 和函数调用指令 call 做了区分。前者被视为函数内部的程序跳转；而后者则被视为函数调用，即被调用者（callee）开始执行的标志。图4.8给出了跳转指令推理规则（JMP rule），我们将跳转指令 jmp 指令和其后继指令 i 一起验证是考虑到 SPARCv8 的延时跳转特性，实际的程序跳转是发生在 i 执行结束之后的。

跳转指令的推理规则首先要求目标跳转地址是合法的，所谓合法跳转地址，即程序规范集合 Ψ 中存在以该目标跳转地址为起始地址的基本代码块的程序规范。例如：如果目标跳转地址为 f′，并且存在程序规范 (fp, fq) 满足 $\mathtt{f}' \in \mathrm{dom}(\Psi)$ 和 $\Psi(\mathtt{f}') = (\mathrm{fp}, \mathrm{fq})$，那么我们说该目标跳转地址是合法的。确定目标跳转地址是合法的之后，我们需要验证 jmp 的后继指令 i 能够通过指令推理规则的验证。假设验证 “jmp; i” 序列之前，程序状态满足断言 p，那么验证指令 i 时，程序状态应该满足 $p\downarrow\downarrow$，因为在执行 jmp 和 i 指令之前都会执行延时写入状态转移。如果指令 i 指令结束之后程序状态满足 p'，那么 p' 应该能够蕴含目标指令序列的前断言 fp ι 分离连接上断言 p_r，即：$p' \Rightarrow \mathrm{fp}\ \iota * p_r$。这里的 ι 是参数列表，而 p_r 则用来表示目标指令序列执行过程中不会访问的（既不会读取也不会修改的）程序状态。最后，因为我们说过，jmp 指令被视为函数内部的程序跳转，而程序

$$\frac{\begin{array}{c}\mathtt{f}' \in \mathrm{dom}(\Psi) \quad \Psi(\mathtt{f}') = (\mathrm{fp}, \mathrm{fq}) \quad \Psi \vdash \{(p', q)\}\ \mathtt{f}{+}8 : \mathbb{I} \\ p{\downarrow} \Rightarrow (\mathtt{r}_{15} \mapsto \_) * p_1 \quad \vdash \{(\mathtt{r}_{15} \mapsto \mathtt{f} * p_1){\downarrow}\}\ \mathtt{i}\ \{p_2\} \\ \exists \iota, p_r.\ (p_2 \Rightarrow \mathrm{fp}\ \iota * p_r) \wedge (\mathrm{fq}\ \iota * p_r \Rightarrow p') \wedge (\mathrm{fq}\ \iota \Rightarrow \mathtt{r}_{15} = \mathtt{f})\end{array}}{\Psi \vdash \{(p, q)\}\ \mathtt{f}\ :\ \mathtt{call}\ \mathtt{f}'; \mathtt{i}; \mathbb{I}}\ \textbf{(CALL)}$$

$$\frac{p{\downarrow}{\downarrow} \Rightarrow (\mathtt{r}_{15} \mapsto \mathtt{f}') * p_1 \quad \vdash \{p_1\}\ \mathtt{i}\ \{p_2\} \quad (\mathtt{r}_{15} \mapsto \mathtt{f}') * p_2 \Rightarrow q}{\Psi \vdash \{(p, q)\}\ \mathtt{f}\ :\ \mathtt{retl}; \mathtt{i}}\ \textbf{(RETL)}$$

图 4.9　函数调用/返回验证规则

规范的后断言描述的是当前函数结束时的程序状态，所以，“jmp; i”序列和目标跳转指令序列属于同一函数，而它们后断言描述的是同一程序状态，即：当前函数结束时的程序状态。因此我们还需要：$\mathrm{fq}\ \iota * p_r \Rightarrow q$。

函数调用模块化验证的支持　我们在图4.9中展示了函数调用推理规则（CALL rule）和函数返回推理规则（RETL rule）。这两条规则的设计思路来源于之前的 SCAP[12] 工作。但 SPARCv8 中的函数调用 call 指令和函数返回 retl 指令都是延时跳转指令，所以，需要做和 jmp 指令验证相同的处理，即：将延时跳转指令和其后继指令一起验证。

我们首先看函数调用指令 call 指令的验证，假设执行 call 指令前，程序状态满足断言 p，call 指令的执行会将当前代码标记 f（即当前指令的地址）保存到寄存器 $\mathtt{r}_{15}$ 中。由于延时跳转特性，此时控制流并不会立即发生转移，而是在执行完后继指令 i 之后才发生跳转。假设这里的被调用函数为 $\mathtt{f}'$ 并且指令 i 执行结束之后程序状态满足断言 p_2，我们从程序规范集合 Ψ 中取出函数 $\mathtt{f}'$ 的程序规范 (fp, fq)，我们需要保证发生函数调用时的程序状态满足函数的断言，即：$p_2 \Rightarrow \mathrm{fp}\ \iota * p_r$。此处，$\iota$ 和 p_r 与 jmp 中的一样，分别表示程序规范的参数列表和描述不会被目标函数访问的程序状态的断言。在被调用函数 $\mathtt{f}'$ 返回调用者之后，我们要求后继的指令序列 $\mathbb{I}$ 仍然可以通过我们的逻辑验证，假设验证 $\mathbb{I}$ 的前断言是 p'，我们需要函数 $\mathtt{f}'$ 的后断言能够蕴含 p'，即：$\mathrm{fq}\ \iota * p_r \Rightarrow p'$，因为函数 $\mathtt{f}'$ 执行结束的状态即为指令序列 $\mathbb{I}$ 执行的起始状态。

支持汇编函数模块化验证的关键问题在于如何保证被调用函数在执行结束之后会返回调用者？此处我们对被调用函数 $\mathtt{f}'$ 的程序规范做了约束。我们要求函数 $\mathtt{f}'$ 执行结束时，寄存器 $\mathtt{r}_{15}$ 中的内容还是发生函数调用时的保存的标识符 f（即：$\mathrm{fp}\ \iota \Rightarrow \mathtt{r}_{15} = \mathtt{f}$）。因为函数返回地址是通过 $\mathtt{r}_{15}$ 中的内容计算得来，所以，要求函数执行结束时 $\mathtt{r}_{15}$ 中保存的标识符和发生函数调用的程序点一致，就可以保证被调用者执行结束之后会返回到正确的地址执行。

对于函数返回推理规则（RETL rule），由之前的介绍我们可以知道，我们赋

$$\frac{\mathtt{sr}\mapsto \_ * p \Rightarrow (\mathtt{r}_s = w_1 \wedge \circ = w_2)}{\vdash \{\mathtt{sr}\mapsto \_ * p\}\ \mathtt{wr}\ \mathtt{r}_s \circ \mathtt{sr}\ \{(\triangleright_3 \mathtt{sr}\mapsto (w_1 \oplus w_2)) * p\}}\ \textbf{(WR)}$$

$$\frac{}{\vdash \{\mathtt{sr}\mapsto w * \mathtt{r}_d \mapsto \_\}\ \mathtt{rd}\ \mathtt{sr}\ \mathtt{r}_d\ \{\mathtt{sr}\mapsto w * \mathtt{r}_d \mapsto w\}}\ \textbf{(RD)}$$

$$\frac{\begin{array}{c} p \Rightarrow (\mathtt{r}_s = w_1 \wedge \circ = w_2) \qquad w'_{id} = \textbf{next\_cwp}(w_{id}) \qquad w \,\&\, 2^{w'_{id}} = 0 \\ p \Rightarrow \left(\mathtt{cwp}\mapsto (\!| w_{id}, F \cdot \_ \cdot \_ |\!)\right) * (\mathsf{out}\mapsto \mathrm{fm}_o) * (\mathsf{local}\mapsto \mathrm{fm}_l) * (\mathsf{in}\mapsto \mathrm{fm}_i) * p_1 \\ \left(\mathtt{cwp}\mapsto (\!| w'_{id}, \mathrm{fm}_l :: \mathrm{fm}_i :: F |\!)\right) * (\mathsf{out}\mapsto \_) * (\mathsf{local}\mapsto \_) * (\mathsf{in}\mapsto \mathrm{fm}_o) * p_1 \Rightarrow \mathtt{r}_d \mapsto \_ * p_2 \end{array}}{\vdash \{(\mathtt{wim}\mapsto w) * p\}\ \mathtt{save}\ \mathtt{r}_s \circ \mathtt{r}_d\ \{(\mathtt{wim}\mapsto w) * (\mathtt{r}_d \mapsto w_1 + w_2) * p_2\}}\ \textbf{(SAVE)}$$

where $[\mathtt{r}_i, \cdots, \mathtt{r}_{i+7}] \mapsto [w_0, \cdots, w_7] \overset{\text{def}}{=} \mathtt{r}_i \mapsto w_0 * \cdots * \mathtt{r}_{i+7} \mapsto w_7$
and out, local and in are defined in Fig. 3.5.

$$\frac{\begin{array}{c} p \Rightarrow (\mathtt{r}_s = w_1 \wedge \circ = w_2) \qquad w'_{id} = \textbf{prev\_cwp}(w_{id}) \qquad w \,\&\, 2^{w'_{id}} = 0 \\ p \Rightarrow \left(\mathtt{cwp}\mapsto (\!| w_{id}, \mathrm{fm}_1 :: \mathrm{fm}_2 :: F |\!)\right) * (\mathsf{out}\mapsto \_) * (\mathsf{local}\mapsto \_) * (\mathsf{in}\mapsto \mathrm{fm}_i) * p_1 \\ \left(\mathtt{cwp}\mapsto (\!| w'_{id}, F \cdot \_ \cdot \_ |\!)\right) * (\mathsf{out}\mapsto \mathrm{fm}_i) * (\mathsf{local}\mapsto \mathrm{fm}_1) * (\mathsf{in}\mapsto \mathrm{fm}_2) * p_1 \Rightarrow \mathtt{r}_d \mapsto \_ * p_2 \end{array}}{\vdash \{(\mathtt{wim}\mapsto w) * p\}\ \mathtt{restore}\ \mathtt{r}_s \circ \mathtt{r}_d\ \{(\mathtt{wim}\mapsto w) * (\mathtt{r}_d \mapsto w_1 + w_2) * p_2\}}\ \textbf{(RESTORE)}$$

图 4.10　部分指令推理规则

予了我们的验证逻辑直接风格（direct-style）的语义，所以我们将 `retl` 指令视为当前函数执行结束的标志。因为程序规范中后断言描述的是当前函数执行结束时程序的状态，所以，这里我们需要保证后断言 q 在执行完 `retl` 的后继指令 `i` 之后成立（考虑到 SPARCv8 的延时跳转特性）。当然，还有个额外的约束就是指令 `i` 不能修改寄存器 $\mathtt{r}_{15}$ 中的内容。因为由之前对函数调用推理规则（CALL rule）的介绍可知，为了保证被调用函数在执行结束后能够返回正确的程序点继续执行，我们要求了被调用函数执行结束时 $\mathtt{r}_{15}$ 寄存器中的内容和发生函数调用时保存在 $\mathtt{r}_{15}$ 中的标识符一致，而 `retl` 指令通过 $\mathtt{r}_{15}$ 寄存器计算返回地址。所以，`call` 指令规则中对 $\mathtt{r}_{15}$ 寄存器的约束，实际上保证了发生函数调用时保存在 $\mathtt{r}_{15}$ 寄存器中的值和被调用函数返回时，计算返回地址时所使用的 $\mathtt{r}_{15}$ 寄存器的值相同。约束 `i` 不能修改 $\mathtt{r}_{15}$ 寄存器的内容，也就保证了我们可以通过函数后断言所描述的内容确定函数是否能返回正确的程序点执行。

4.2.3　指令推理规则

图4.10中展示了部分的指令推理规则，这里类似于 `add`，`nop` 等指令的推理规则因为比较简单且和之前 ARM，x86 上相关指令的验证没什么不同，所以这里不再详述，我们主要希望通过图4.10 中的内容来介绍我们的验证逻辑是如何支持 SPARCv8 的特殊寄存器延时写入和寄存器窗口机制的。

特殊寄存器延时写入特性的支持　我们在第3章中介绍到 SPARCv8 使用指令 `wr`

来修改特殊寄存器的值，不过这个写入操作何时生效则是由一个预定义的系统参数 X （$0 \leqslant X \leqslant 3$）来决定的。这样的不确定性也给我们为 wr 指令设计推理规则带来了一定的挑战。当然，可以确定的是，至多 3 个指令周期后该延时写入操作一定会生效，因为延时周期最大为 3。所以，可以想到一种直观的解决方法是：将 wr 指令区别于其它简单指令，并将该指令的验证工作放在代码序列的验证工作中，然后将 wr 及其后继的三条指令一起验证，类似于对延时跳转指令的处理，大致形式如下：

$$\Psi \vdash \{(p,q)\}\ \mathtt{f} : \mathtt{wr}\ \mathtt{r}_s \circ \mathtt{sr};\ \mathtt{i}_1;\ \mathtt{i}_2;\ \mathtt{i}_3;\ \mathbb{I}$$

这是一种可行的方法，但问题在于无法处理多条连续 wr 指令的情况。通常情况下程序员不会在不确定对某一寄存器延时写入是否生效的情况下，重复去写该寄存器，但不排除会连续写多个不同特殊寄存器的情况。例如：

$$\mathtt{wr}\ \mathtt{r}_1, 0, \mathtt{asr}_0;\ \mathtt{wr}\ \mathtt{r}_1, 0, \mathtt{asr}_1$$

上面这个例子中，我们希望同时对 $\mathtt{asr}_0$ 和 $\mathtt{asr}_1$ 这两个特殊寄存器赋值，因为 $\mathtt{asr}_0$ 和 $\mathtt{asr}_1$ 是不同的寄存器，我们没有必要先对寄存器 $\mathtt{asr}_0$ 赋值，然后等待 3 个指令周期确定写入生效后再对 $\mathtt{asr}_1$ 寄存器赋值，随后再等待三个指令周期确保对 $\mathtt{asr}_1$ 寄存器写入的生效。更高效的方法是连续对 $\mathtt{asr}_0$ 和 $\mathtt{asr}_1$ 寄存器赋值，然后等待三个指令周期，而上述提到的验证 wr 指令的方法显然无法支持这类情况。另一个方法是依然将 wr 指令作为简单指令，但我们需要在断言中增加对延时缓存 D 的描述，如果直接描述整个 D，那么 D 中一些我们不关心的延时写入操作则显得冗余，也不利于我们程序规范的书写。

结合上述两种方法的不足之处，我们的方法是：依然将 wr 指令视为简单指令，并且只描述延时缓存中我们所关心的延时写入操作。在第4.1节中，我们定义的断言 $\triangleright_t \mathtt{sr} \mapsto w$ 就是用来表示：延时缓存中可能存在一个关于 sr 寄存器的延时写入操作，待写入的值为 w，并且至多经过 t 个指令周期，该延时写入操作一定会生效。图4.10中的写入指令规则（WR rule）是我们为 wr 指令设计的推理规则，该推理规则的前断言要求当前程序有目标寄存器的占有权，使用断言 $(\mathtt{sr} \mapsto \_)$ 表示，这个断言同时也表示目标寄存器 sr 不在延时缓存中（由4.2中断言语义可知）。执行 wr 指令之后，目标寄存器 sr 的状态为：$\triangleright_3 \mathtt{sr} \mapsto (w_1 \oplus w_2)$，表示新值 $w_1 \oplus w_2$ 在至多 3 个指令周期之后将被写入目标寄存器 sr。因为之前介绍过，延时周期最大为 3，虽然具体的延时周期是系统预定义的可能小于 3，但出于对程序可移植性的考虑，程序在编程过程中通常都会假设延时周期为最大值 3，所以，这里我们也选择 3 个指令周期作为延时写入操作可能的延时时间。SPARCv8 中同时也提供指令 rd 来读特殊寄存器的值，我们使用图4.10中的读

指令规则（RD rule）来验证 rd 指令，读指令规则在前断言中约束了在读特殊寄存器 sr 时，sr 不在延时缓存中。我们用下面一个简单例子来展示我们的写入规则的使用方法，以及可以验证连续写入多个不同特殊寄存器的情况。

$$\{\mathtt{r}_1 \mapsto w_1 * \mathtt{r}_2 \mapsto w_2 * \mathtt{asr}_0 \mapsto \_ * \mathtt{asr}_1 \mapsto \_\}$$
$$\mathtt{wr}\quad \mathtt{r}_1, 0, \mathtt{asr}_0$$
$$\{\mathtt{r}_1 \mapsto w_1 * \mathtt{r}_2 \mapsto w_2 * (\triangleright_3 \mathtt{asr}_0 \mapsto w_1) * \mathtt{asr}_1 \mapsto \_\}$$
$$\mathtt{wr}\quad \mathtt{r}_2, 0, \mathtt{asr}_1$$
$$\{\mathtt{r}_1 \mapsto w_1 * \mathtt{r}_2 \mapsto w_2 * (\triangleright_2 \mathtt{asr}_0 \mapsto w_1) * (\triangleright_3 \mathtt{asr}_1 \mapsto w_2)\}$$
$$\mathtt{add}\quad \mathtt{r}_1, \mathtt{r}_2, \mathtt{r}_2$$
$$\{\mathtt{r}_1 \mapsto w_1 * \mathtt{r}_2 \mapsto (w_1 + w_2) * (\triangleright_1 \mathtt{asr}_0 \mapsto w_1) * (\triangleright_2 \mathtt{asr}_1 \mapsto w_2)\}$$
$$\mathtt{nop}$$
$$\{\mathtt{r}_1 \mapsto w_1 * \mathtt{r}_2 \mapsto (w_1 + w_2) * (\triangleright_0 \mathtt{asr}_0 \mapsto w_1) * (\triangleright_1 \mathtt{asr}_1 \mapsto w_2)\}$$
$$\mathtt{rd}\quad \mathtt{asr}_0, \mathtt{r}_2$$
$$\{\mathtt{r}_1 \mapsto w_1 * \mathtt{r}_2 \mapsto w_1 * \mathtt{asr}_0 \mapsto w_1 * (\triangleright_0 \mathtt{asr}_1 \mapsto w_2)\}$$

寄存器窗口机制　在第3中我们介绍了 SPARCv8 使用指令 save 和 restore 来操作寄存器窗口。所以，我们在图4.10中的定义了保存指令规则（SAVE rule）和恢复指令规则（RESTORE rule）来刻画我们对 SPARCv8 中寄存器窗口机制的处理。通过第3.2中给出的 save 和 restore 的语义，我们很容易理解这两条推理规则。这里我们再结合图4.11对这两条规则做详细说明。

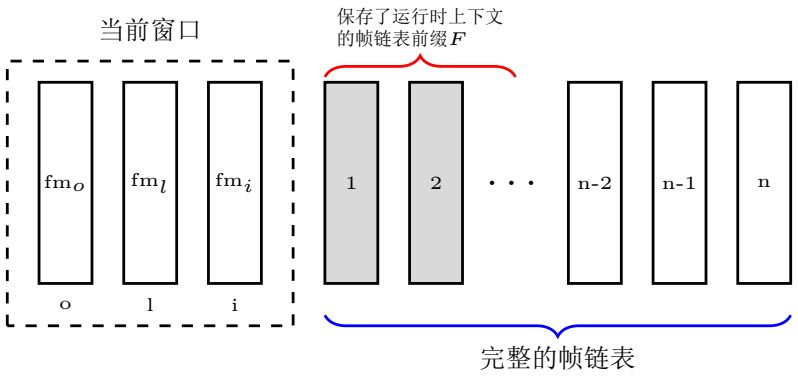

图 4.11　寄存器窗口状态

图4.11中，当前窗口中 out，local 和 in 寄存器的内容分别为：fm_o，fm_l 和 fm_i。save 指令常用于发生函数调用时保存调用者的运行时上下文，save 指令会将 local 寄存器中的内容 fm_l 和 in 寄存器中的内容 fm_i 加入帧链表的头部，而原来 out 寄存器中的内容 fm_o 则仍然保留在当前窗口中，只不过变为当前窗口 in

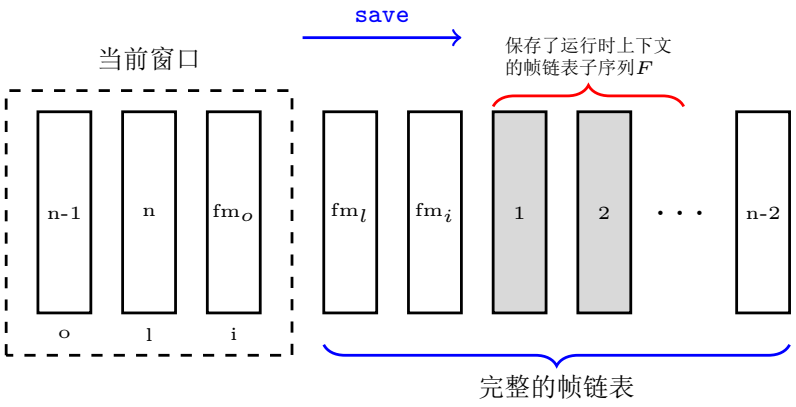

图 4.12　执行完 save 指令后的寄存器窗口状态

寄存器的内容，通常用于调用者（caller）向被调用者（callee）传递参数。根据之前的介绍，save 指令常被用于保存函数调用者的运行时上下文，对于新窗口中 out 和 local 的内容，类似于为被调用者分配的新的寄存器状态，我们并不关心它的具体内容，所以这里使用 "_" 表示不关心它的具体内容。我们用图4.12来表示执行完 save 指令之后的寄存器状态。

而对于程序状态中帧链表的描述，我们希望只描述它保存了运行时上下文的一个前缀 F （图4.11中标记为灰色的部分），因为这一部分的内容是我们所关心的，而其它窗口因为未被使用，所以我们不关心它的具体内容。不过，仅仅描述一个前缀 F 并不足够，因为 F 中保存了之前运行时的上下文，这些内容在未来是希望被恢复的，所以，我们希望我们所描述的帧链表子序列 F 在窗口旋转的过程中是稳定的（因为它代表着之前所保存的运行时上下文），即：窗口旋转前后帧链表中都存在子序列 F。比较图4.11和4.12可知，执行 save 指令会将帧链表尾部的两个帧移入当前窗口（模拟窗口的旋转），要想保证子序列 F 在窗口旋转的过程中不被破坏，必须要保证移入当前窗口的处于完整帧链表尾部的两个帧不在帧序列 F 中。所以，为了保证 F 在窗口旋转过程中是稳定的，我们在所描述帧链表的前缀 F 后面加了两个占位符 "_"，表示帧链表中的两个帧，我们并不关心它们的具体内容，但它可以保证执行窗口旋转不会对保存了之前运行时上下文的帧链表子序列 F 产生影响。保存指令规则（SAVE rule）中还包含着一些其它附加的条件，例如计算下一个窗口的编号（$w'_{id} = \mathbf{next\_cwp}(w_{id})$），判断下一个窗口是否有效（$w \,\&\, 2^{w'_{id}}$）等，都在第3章的对 SPARCv8 的形式化建模中做了详述，此处不再累赘。restore 指令常用于被调用者执行结束时恢复调用者的运行时上下文，restore 指令执行和 save 相反的操作，恢复最近一次所保存的 local 和 in 寄存器的内容 fm_1 和 fm_2，原窗口 in 寄存器的内容 fm_i 变为当前窗口 out 寄存器的内容，通常是为了方便被调用者向调用者传递返回值。

4.3　验证逻辑的可靠性

上一节我们介绍了我们的验证逻辑，我们在这一节介绍我们验证逻辑的语义及可靠性的证明。

我们首先定义指令推理规则的语义：

定义 4.1 (指令推理规则语义)

$$\vDash \{p\}\ \mathtt{i}\ \{q\} \stackrel{\text{def}}{=} \forall S.\, S \vDash p \to (\exists S'.\, (S \xrightarrow{\mathtt{i}} S') \wedge S' \vDash q)$$

指令推理规则的语义比较直观，它告诉我们如果有一个程序状态 S 满足前断言 p，那么存在一个程序状态 S'，S 可以通过指令 i 的执行转移到 S'，并且 S' 满足后断言 q。

引理 4.2 (指令推理规则的可靠性)

$$\vdash \{p\}\ \mathtt{i}\ \{q\} \Longrightarrow \vDash \{p\}\ \mathtt{i}\ \{q\}$$

我们接下来定义指令序列推理规则的语义，我们首先定义指令序列的安全性：safe_insSeq$(C, S, \mathtt{pc}, \mathtt{npc}, q, \Psi)$。

定义 4.2 (指令序列的安全性)　safe_insSeq$(C, S, \mathtt{pc}, \mathtt{npc}, q, \Psi)$ 成立，当且仅当下述全部成立：

- 如果 $C(\mathtt{pc}) = \mathtt{i}$，那么：
  - 存在 S'，$\mathtt{pc}'$，$\mathtt{npc}'$，使得 $C \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto (S', \mathtt{pc}', \mathtt{npc}')$ 成立；
  - 对于任意的 $S', \mathtt{pc}', \mathtt{npc}'$，如果 $C \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto (S', \mathtt{pc}', \mathtt{npc}')$ 成立，那么有 safe_insSeq $(C, S', \mathtt{pc}', \mathtt{npc}', q, \Psi)$ 成立；
- 如果 $C(\mathtt{pc}) = \mathtt{jmp\ a}$，那么：
  - 存在 $S', \mathtt{pc}', \mathtt{npc}'$，使得 $C \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$ 成立；
  - 对于任意的 $S', \mathtt{pc}', \mathtt{npc}'$，如果 $C \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$ 成立，那么存在 fp, fq, ι 和 p_r，使得下述内容成立：
    1. $\mathtt{npc}' = \mathtt{pc}' + 4$；$\Psi(\mathtt{pc}') = (\mathrm{fp}, \mathrm{fq})$，
    2. $S' \vDash (\mathrm{fp}\ \iota) * p_r$, $(\mathrm{fq}\ \iota) * p_r \Rightarrow q$；
- 如果 $C(\mathtt{pc}) = \mathtt{be\ f}$，那么：
  - 存在 $S', \mathtt{pc}', \mathtt{npc}'$，使得 $C \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$ 成立；
  - 对于任意的 $S', \mathtt{pc}', \mathtt{npc}'$，如果 $C \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$ 成立，那么下述成立：
    * 如果 $S.Q.R(\mathtt{z}) \neq 0$，那么存在 fp, fq, ι, p_r，使得：

1. $\mathtt{pc}' = \mathtt{f}$，$\mathtt{npc}' = \mathtt{f}+4$，$\Psi(\mathtt{f}) = (\mathrm{fp}, \mathrm{fq})$，
2. $S' \vDash (\mathrm{fp}\ \iota) * p_r$, $(\mathrm{fq}\ \iota) * p_r \Rightarrow q$；

∗ 如果 $S.Q.R(\mathtt{z}) = 0$，那么 safe_insSeq$(C, S', \mathtt{pc}', \mathtt{npc}', q, \Psi)$ 成立；

- 如果 $C(\mathtt{pc}) = \mathtt{call\ f}$，那么：
  - 存在 $S', \mathtt{pc}', \mathtt{npc}'$，使得 $C \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$ 成立；
  - 对于任意的 $S', \mathtt{pc}', \mathtt{npc}'$，如果 $C \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$ 成立，那么存在 $\mathrm{fp}, \mathrm{fq}, \iota$ 和 p_r，使得下述成立：
    1. $\mathtt{pc}' = \mathtt{f}$，$\mathtt{npc}' = \mathtt{f}+4$，$\Psi(\mathtt{f}) = (\mathrm{fp}, \mathrm{fq})$，
    2. $S' \vDash (\mathrm{fp}\ \iota) * p_r$，
    3. 对于任意的 S'，若 $S' \vDash (\mathrm{fq}\ \iota) * p_r$ 成立，则 safe_insSeq $(C, S', \mathtt{pc}+8, \mathtt{pc}+12, q, \Psi)$ 成立；
    4. 对于任意的 S'，若 $S' \vDash \mathrm{fq}\ \iota$ 成立，则 $S'.Q.R(\mathtt{r}_{15}) = \mathtt{pc}$ 成立；
- 如果 $C(\mathtt{pc}) = \mathtt{retl}$，那么：
  - 存在 $S', \mathtt{pc}', \mathtt{npc}'$，使得 $C \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$ 成立；
  - 对于任意的 $S', \mathtt{pc}', \mathtt{npc}'$，如果 $C \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$ 成立，那么下述成立：
    1. $S' \vDash q$；
    2. $\mathtt{pc}' = S'.Q.R(\mathtt{r}_{15})+8$，$\mathtt{npc}' = S'.Q.R(\mathtt{r}_{15})+12$。

指令序列的安全性的定义（定义4.2）保证了代码 C 可以从起始程序状态 $(S, \mathtt{pc}, \mathtt{npc})$ 开始安全地执行，直到当前指令序列（$C[\mathtt{pc}]$）结束，并且如果 $C[\mathtt{pc}]$ 是以函数返回指令（retl）结尾，那么后断言 q 将成立。定义4.2中的“$C \vdash \_ \longmapsto^n \_$”表示程序执行 n 步。下面我们来逐条介绍定义4.2中的各种情况：

- 如果当前指令为简单指令 i (即：$C(\mathtt{pc}) = \mathtt{i}$)，那么当前程序可以走一步；并且如果程序可以走一步，即：$C \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto (S', \mathtt{pc}', \mathtt{npc}')$，那么当前指令序列（$C[\mathtt{pc}']$）仍然是安全的。

- 如果当前指令为跳转指令 jmp a（即：$C(\mathtt{pc}) = \mathtt{jmp\ a}$），那么当前程序可以走两步（因为 jmp 是 SPARCv8 中的延时跳转特性，程序能够进行控制流的转移说明当前指令 jmp 和其后继指令都能安全执行）；并且如果程序可以走两步，即：$C \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$，那么程序控制流转移到起始地址为 $\mathtt{pc}'$ 的基本代码块继续执行，此时 $\mathtt{npc}' = \mathtt{f}+4$，并且程序规范集合 Ψ 中存在起始地址为 f 的基本代码块的程序规范，即：$\Psi(\mathtt{pc}') = (\mathrm{fp}, \mathrm{fq})$。我们要求该代码块程序规范中的前段言在当前状态 S'

下成立，并且程序规范中的后断言可以蕴含 jmp a 指令所在代码块的后断言 q。这个可以通过我们之前对 jmp 指令推理规则（JMP rule）和程序规范的介绍中理解，因为我们将 jmp 指令视为函数内部跳转，所以，断言 q 和以目标跳转地址 pc′ 为起始地址的基本代码块的后断言描述的都是当前函数执行结束时的程序状态，即同一程序点的程序状态，所以此处还要求 $(\mathrm{fq}\ \iota) * p_r \Rightarrow q$，其中 p_r 描述目标代码块执行中不会访问的程序状态。

- 当前指令为条件跳转指令的情况（即：$\boldsymbol{C}(\mathtt{pc}) = \mathtt{be\ f}$）可以视为前两种情况的结合。首先，因为 be 是延时跳转指令，所以我们要求程序可以走两步；并且，如果程序可以走两步，则根据判断条件是否成立分两种情况讨论：如果判断条件成立（$\boldsymbol{S}.\boldsymbol{Q}.\boldsymbol{R}(\mathtt{z}) \neq 0$），则需要进行控制流转移，采用和 jmp 指令对应情况相同的处理；而如果判断条件不成立（$\boldsymbol{S}.\boldsymbol{Q}.\boldsymbol{R}(\mathtt{z}) = 0$），则不需要进行控制流转移，采用和简单指令 i 对应情况相同的处理。

- 如果当前指令为函数调用指令 call（即：$\boldsymbol{C}(\mathtt{pc}) = \mathtt{call\ f}$），则程序可以走两步；并且如果程序可以走两步，那么将发生函数调用，因为我们没有把 call 指令视为基本代码块的结束指令，所以我们要求被调用函数返回后 call 指令的后继指令序列（$\boldsymbol{C}[\mathtt{pc+8}]$）仍然是安全。

- 如果当前指令为函数返回指令 retl（即：$\boldsymbol{C}(\mathtt{pc}) = \mathtt{retl}$），则程序可以走两步；并且如果程序可以走两步，即：$\boldsymbol{C} \vdash (\boldsymbol{S}, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (\boldsymbol{S}', \mathtt{pc}', \mathtt{npc}')$，此时当前函数执行结束，程序状态 $\boldsymbol{S}'$ 满足后断言 q，程序跳转到 $\mathtt{pc}' = \boldsymbol{S}.\boldsymbol{Q}(\mathtt{r}_{15})+8, \mathtt{npc}' = \boldsymbol{S}.\boldsymbol{Q}(\mathtt{r}_{15})+12$。

我们在定义4.3中基于指令序列的安全性定义（定义4.2）定义我们指令序列推理规则的语义。

定义 4.3 (指令序列推理规则语义)　$\Psi \vDash \{(p, q)\}\ \mathtt{f} : \mathbb{I}$ 成立，当且仅当，对于任意的 $\boldsymbol{C}$ 和 $\boldsymbol{S}$，如果 $\boldsymbol{C}[\mathtt{f}] = \mathbb{I}$ 和 $\boldsymbol{S} \vDash p$ 成立，那么 $\mathsf{safe\_insSeq}(\boldsymbol{C}, \boldsymbol{S}, \mathtt{f}, \mathtt{f+4}, q, \Psi)$ 成立。

引理 4.3 (指令序列推理规则的可靠性)

$$\Psi \vdash \{(p, q)\}\ \mathtt{f} : \mathbb{I} \implies \Psi \vDash \{(p, q)\}\ \mathtt{f} : \mathbb{I}$$

我们可以通过证明引理4.3成立，来证明我们指令序列推理规则的可靠性。最后我们基于指令序列推理规则语义（定义4.3）来定义代码堆推理规则的语义（定义4.4）。我们基于代码堆推理规则的语义定义 SPARCv8 验证逻辑的可靠性（定理4.4）。

定义 4.4　$\vDash \boldsymbol{C} : \Psi$ 成立，当且仅当，对于任意的 f，fp 和 fq，如果有 $\Psi(\mathtt{f}) =$

(fp, fq) 成立，那么对于任意的 ι，$\Psi \vDash \{(\text{fp}\ \iota, \text{fq}\ \iota)\}\ \texttt{f} : C[\texttt{f}]$ 成立。

定理 4.4 (验证逻辑的可靠性)

$$\vdash C : \Psi \implies \vDash C : \Psi$$

值得注意的是，通过代码堆推理规则的语义，我们知道验证逻辑的可靠性保证了通过我们验证逻辑验证的代码中，每个被赋予了程序规范的基本代码块都是安全的（即：满足定义 safe_insSeq），但是否可以保证所验证的整个程序是安全的？下面我们将介绍，我们的验证逻辑可以保证被分别验证的基本代码块连接起来所得到的完整程序仍然是安全的。我们首先定义 $\text{safe}^n(C, S, \texttt{pc}, \texttt{npc}, q, k)$ 来描述整个程序的安全性（定义4.5）。

定义 4.5 (程序安全性)　$\text{safe}^0(C, S, \texttt{pc}, \texttt{npc}, q, k)$ 永远成立。

$\text{safe}^{n+1}(C, S, \texttt{pc}, \texttt{npc}, q, k)$ 成立，当且仅当下述全部成立：

1. 如果 $C(\texttt{pc}) \in \{\texttt{i}, \texttt{jmp a}, \texttt{be f}\}$，那么：
   - 存在 $S', \texttt{pc}', \texttt{npc}'$，使得 $C \vdash (S, \texttt{pc}, \texttt{npc}) \longmapsto (S', \texttt{pc}', \texttt{npc}')$ 成立；
   - 对于任意的 $S', \texttt{pc}', \texttt{npc}'$，如果 $C \vdash (S, \texttt{pc}, \texttt{npc}) \longmapsto (S', \texttt{pc}', \texttt{npc}')$ 成立，那么 $\text{safe}^n\ (C, S', \texttt{pc}', \texttt{npc}', q, k)$ 成立；
2. 如果 $C(\texttt{pc}) = \texttt{call f}$，那么
   - 存在 $S', \texttt{pc}', \texttt{npc}'$，使得 $C \vdash (S, \texttt{pc}, \texttt{npc}) \longmapsto^2 (S', \texttt{pc}', \texttt{npc}')$ 成立；
   - 对于任意的 $S', \texttt{pc}', \texttt{npc}'$，如果 $C \vdash (S, \texttt{pc}, \texttt{npc}) \longmapsto^2 (S', \texttt{pc}', \texttt{npc}')$ 成立，那么 $\text{safe}^n(C, S', \texttt{pc}', \texttt{npc}', q, k+1)$ 成立；
3. 如果 $C(\texttt{pc}) = \texttt{retl}$，那么
   - 存在 $S', \texttt{pc}', \texttt{npc}'$，使得 $C \vdash (S, \texttt{pc}, \texttt{npc}) \longmapsto^2 (S', \texttt{pc}', \texttt{npc}')$ 成立；
   - 对于任意的 $S', \texttt{pc}', \texttt{npc}'$，如果 $C \vdash (S, \texttt{pc}, \texttt{npc}) \longmapsto^2 (S', \texttt{pc}', \texttt{npc}')$，成立，那么

     如果 $k = 0$，则

     $$S' \vDash q$$

     否则

     $$\text{safe}^n(C, S', \texttt{pc}', \texttt{npc}', q, k-1)。$$

简单来说，程序的安全性（$\text{safe}^n(C, S, \texttt{pc}, \texttt{npc}, q, k)$）保证了程序 C 从起始状态 $(S, \texttt{pc}, \texttt{npc})$ 开始执行，要么在 n 步内正常 终止，并且末状态满足断言 q；或者可以安全地执行至少 n 步。注意，这里的程序安全性定义中的参数 k 表示函数调用的层数。

定义程序安全性的难点在于如何定义程序的终止状态，即：我们希望后断言 q 成立的程序点。在之前的介绍中我们说过，我们赋予了我们的验证逻辑直接风

格（direct-style）的语义，我们将 retl 指令视为函数结束的标志，所以我们希望当前函数结束时该函数程序规范的后断言成立。但因为程序在执行过程中还会调用别的子函数，所以问题在于如何判断当前的 retl 指令是否是最外层主函数的结束点。这里，我们在定义程序安全性时用了一个参数 k 来记录函数的调用层数：

- 如果当前指令为简单指令 i，或者跳转指令 jmp，或者条件跳转指令 be，那么程序执行当前指令并不会修改函数的调用层数，因为 jmp 和 be 被视为了函数内部的跳转指令；
- 如果当前指令为函数调用指令 call，那么说明当前函数希望调用某一新的子函数，所以函数调用的层数应该加一；
- 如果当前指令为函数返回指令 retl，我们需要通过调用层数 k 来判断当前的函数返回指令是否属于最外层主函数。如果 $k = 0$，则该 retl 指令属于最外层主函数；否则，它属于最外层主函数执行过程中调用的子函数。

我们可以证明推论4.5成立，来说明我们的验证逻辑可以保证被验证的基本代码块所连接得到的完整程序是安全的。

推论 4.5 (函数安全性)　如果有 $\Psi \vDash \{(p, q)\}\,\mathtt{pc} : C[\mathtt{pc}]$，$S \vDash p$ 和 $\vDash C : \Psi$ 成立，那么 $\forall n.\ \mathsf{safe}^n(C, S, \mathtt{pc}, \mathtt{pc}{+}4, q, 0)$ 成立。

4.4　验证逻辑在 Coq 中的实现

我们在 Coq 辅助定理证明工具中实现了我们的验证逻辑[31]。表4.1 给出了我们验证逻辑实现的 Coq 代码量统计，包括：434 行 Coq 代码用于定义断言语言和推理规则；11586 行 Coq 代码用于证明验证逻辑可靠性；以及 615 行 Coq 代码用于证明函数安全性（推论4.5）。

表 4.1　验证逻辑实现 Coq 代码量统计

验证逻辑	Coq 行数
断言语言和推理规则	434
逻辑可靠性的证明	11586
函数安全性的证明	615

断言语言和推理规则　我们首先来介绍我们验证逻辑中断言语言和推理规则在 Coq 中的定义。代码4.1展示了我们在 Coq 中定义的断言语言的语法，我们使用归纳定义的方法（inductive）定义断言语言。

代码 4.1　Coq 中断言语言的语法定义

```
Inductive asrt : Type :=
| Aemp : asrt
| Amapsto : Address -> Word -> asrt
| Areg : RegName -> Word -> asrt
| Aregdly : nat -> SpReg -> Word -> asrt
| ATimReduce : asrt -> asrt
| Aframe : Word -> FrameList -> asrt
| Aconj : asrt -> asrt -> asrt
| Adisj : asrt -> asrt -> asrt
| Astar : asrt -> asrt -> asrt
| ... ...
```

在 Coq 中我们定义 “asrt” 表示断言，它的类型是 Type。其中，“Aemp” 是断言，它对应于我们之前介绍的断言 emp；“Amapsto” 是一个断言的构造子（constructor），“Amapsto l w” 对应于我们之前介绍的断言 $l \mapsto w$，其中地址 l 和值 w 在 Coq 中的类型分别被定义为：Address 和 Word；“Aregdly” 是构造描述延时写入断言的构造子，“Aregdly t sr w” 对应于之前所介绍的断言 $\triangleright_t \mathtt{sr} \mapsto w$，其中延时周期 t 在 Coq 中的类型为自然数 nat，特殊寄存器 sr 在 Coq 中的类型被定义为 SpReg。其它断言的构造子与之前所介绍断言的对应关系，这里就不再赘述。当然，直接使用上述定义的断言的构造子在 Coq 中构造断言会非常难看，我们可以使用 Coq 中提供的记号机制（notation）来增加 Coq 代码的可读性。例如代码4.2中我们利用 Notation 在 Coq 中定义了表示描述内存单元和延时写入操作断言的记号（此处单引号表示字符间的分隔符），有了该定义之后，我们就可以在编写 Coq 代码时使用 “l |-> v” 代替 “Amapsto l v”，使用 “n @ rn |==> v” 代替 “Aregdly n rn v”。定义记号时，我们同样需要指明该标记的优先级用于方便 Coq 做复合表达式的解析，例如在代码4.2中我们通过 “at level 20” 指明所定义记号的优先级的值为 20。Coq 允许所定义记号的优先级取值在 0 到 100 之间，取值越小代表该记号的优先级越高。

代码 4.2　部分与断言相关的记号

```
Notation "l |-> v" := (Amapsto l v) (at level 20).
Notation "n '@' rn |==> v" := (Aregdly n rn v)
            (at level 20).
```

定义了断言语言之后，我们来看验证逻辑的推理规则在 Coq 中的实现。我们先从用于验证单条简单指令的推理规则（**Well-Formed Instructions**）开始介绍。我们首先在代码4.3中给出了部分指令推理规则的 Coq 实现，我们同样使用归纳定义的方法（inductive）来定义我们的指令推理规则，这里重点展示的是读指令

规则（RD rule）和写入指令规则（WR rule）的 Coq 定义，他们分别对应于代码4.3中的 “rd_rule” 和 “wr_rule”。这里，Coq 代码中的 “**” 代表分离逻辑中的逻辑中的分离连接 “∗” 在 Coq 中的表示。

代码 4.3　部分指令推理规则的 Coq 实现

```
Inductive wf_ins : asrt -> ins -> asrt -> Prop :=
| ... ...
| rd_rule : forall (rsp : SpReg) v v1 (r1 : GenReg) p,
   wf_ins (rsp |=> v ** r1 |=> v1 ** p) (rd rsp r1)
          (rsp |=> v ** r1 |=> v ** p)

| wr_rule : forall (rsp : SpReg) (rs : GenReg) p v v1 v2 oexp,
  rsp |=> v ** p ==> ((Or rs) ==ₑ v1 //\\ oexp ==ₑ v2) ->
   wf_ins (rsp |=> v ** p) (wr rs oexp rsp)
          ((3 @ rsp |==> (set_spec_reg rsp (v1 xor v2))) ** p)
| ... ...
```

为了方便 Coq 代码的编写，我们同样在 Coq 中定义指令推理规则在 Coq 中的记号（代码4.4）。

代码 4.4　指令推理规则在 Coq 中的记号

```
Notation " '|-' '{{' p '}}' i '{{' q '}}' " :=
                        (wf_ins p i q) (at level 50).
```

代码4.5中给出了部分指令序列推理规则（**Well-Formed Instruction Sequences**）在 Coq 中的实现，这里着重展示了顺序推理规则（SEQ rule）在 Coq 中的实现。Coq 实现中的 “funspec” 是程序规范集合 Ψ 的类型在 Coq 中的表示，“InsSeq” 是指令序列 $\mathbb{I}$ 的类型在 Coq 中的表示。

代码 4.5　部分指令序列推理规则在 Coq 中的实现

```
Inductive wf_seq : funspec -> asrt -> Label ->
                   InsSeq -> asrt -> Prop :=
| seq_rule : forall f i I p p' q Spec,
        |- {{ p ↓ }} i {{ p' }} ->
        wf_seq Spec p' (f + ($ 4)) I q ->
        wf_seq Spec p f (i ;; I) q
| ... ...
```

我们在代码4.6中定义了表示指令推理规则的 Coq 记号

代码 4.6　指令序列推理规则在 Coq 中的记号

```
Notation " Spec '|-' '{{' p '}}' f '#' I '{{' q '}}' " :=
        (wf_seq Spec p f I q) (at level 55).
```

最后我们来介绍，代码堆推理规则（**Well-Formed Code Heap**）在 Coq 中的实现（代码4.7），我们在 Coq 中定义 “wf_cdhp” 表示代码堆的推理规则，它带两个参数：程序规范集合 “Spec”，对应于 Ψ；代码堆 “C”，对应于 C，所以代码堆的推理规则 $\vdash C : \Psi$ 在 Coq 中的表示为 “wf_cdhp Spec C”。定义中的 “LookupC C f I” 是我们之前所介绍的从代码堆中获取指定起始地址指令序列方法的 Coq 实现，对应于 $C[f] = \mathbb{I}$，其中 “I” 是指令序列 $\mathbb{I}$ 在 Coq 中的表示。关于断言语言和推理规则的语义在 Coq 中的实现，此处就不做介绍。

代码 4.7　代码堆推理规则在 Coq 中的实现

```
Definition wf_cdhp (Spec : funspec) (C : CodeHeap) :=
  forall f L fp fq,
        Spec f = Some (fp, fq) ->
        exists I, LookupC C f I /\ wf_seq Spec (fp L) f I (fq L).
```

逻辑可靠性的证明　我们在 Coq 中实现了验证逻辑可靠性的证明，代码4.8中展示的三个 Coq 中的定理，分别对应于指令推理规则、指令序列推理规则、代码堆推理规则可靠性的证明。其中，“Theorem ins_rule_sound” 对应于引理4.2，“Theorem wf_seq_sound” 对应于引理4.3，“Theorem cdhp_rule_sound” 对应于定理4.4。另外指令推理规则、指令序列推理规则、代码堆推理规则的语义在 Coq 中分别被定义为函数：“ins_sound”、“insSeq_sound” 和 “cdhp_sound”。

代码 4.8　Coq 中验证逻辑可靠性的证明

```
Theorem ins_rule_sound : forall p q i,
        |- {{ p }} i {{ q }} -> ins_sound p q i.
Proof.
  ... ...
Qed.

Theorem wf_seq_sound : forall Spec p q f I,
        wf_seq Spec p f I q ->
        insSeq_sound Spec p f I q.
Proof.
  ... ...
Qed.

Theorem cdhp_rule_sound : forall C Spec,
        wf_cdhp Spec C ->
        cdhp_sound Spec C.
Proof.
  ... ...
Qed.
```

函数安全性的证明　我们在 Coq 中实现了推论4.5函数安全性的证明，代码4.9中

的 “Theorem wf_function” 对应于推论4.5的 Coq 实现。在 Coq 实现中的 “safety” 即对应与之前介绍的程序安全性定义 safe（定义4.5）。

代码 4.9　Coq 中函数安全性的证明

```
Theorem wf_function :
  forall p q Spec S C pc I,
        insSeq_sound Spec p pc I q -> LookupC C pc I ->
        cdhp_sound Spec C -> S |= p ->
        forall n, safety n C S pc (pc + ($ 4)) q 0.
Proof.
  ... ...
Qed.
```

更多的关于 Coq 实现细节这里不再做介绍，感兴趣的读者参阅我们的 Coq 代码[31]。

4.5　本章小结

这一章，我们重点介绍了我们为 SPARCv8 汇编程序设计的程序验证逻辑。我们在第4.1节介绍了用于书写程序规范的断言语言，并重点介绍了我们为描述：延时写入操作，执行延时写入转移后的程序状态，以及帧链表所设计的断言。第4.2节介绍了我们为验证 SPARCv8 程序所设计的推理规则，包括：程序规范的形式；用于验证整个代码堆的代码堆推理规则（**Well-formed Code Heap**）；用于验证基本代码块的指令序列推理规则（**Well-formed Instruction Sequence**）；以及验证单条指令的指令推理规则（**Well-formed Instruction**）。我们还介绍了我们的推理规则是如何支持函数的模块验证，以及 SPARCv8 的三个特性：延时跳转、特殊寄存器的延时写入和寄存器窗口机制的验证。我们在第4.3节介绍了我们验证逻辑推理规则的语义以及可靠性的证明，并在第4.4节介绍了我们的验证逻辑在 Coq 中的实现。本章所介绍的验证逻辑的定义以及逻辑可靠性的证明都在 Coq 辅助定理证明工具中实现[31]。

第 5 章　上下文切换程序主要功能模块的验证

这一章将介绍我们使用第4章定义的 SPARCv8 验证逻辑验证某航天嵌入式操作系统中任务上下文切换程序主要功能模块的工作。我们将在第5.1节介绍所验证的任务上下文切换程序的程序结构，在第5.2节给出程序规范的形式化定义，并在第5.3节介绍验证工作的 Coq 实现以及我们在验证工作中是如何利用自动证明策略来简化验证工作的。

5.1　任务上下文切换程序结构

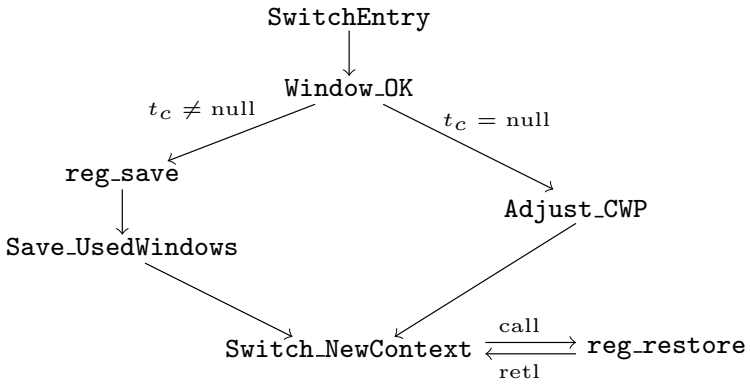

图 5.1　任务上下文切换程序结构

我们所验证的任务上下文切换程序的主要功能是保存当前任务的运行时上下文，并且恢复新任务保存在内存中的上下文。我们在图5.1中展示了我们所验证的任务上下文切换程序的程序结构，包括了用于实现该程序的各个子程序模块，箭头表示各个子模块之间的跳转关系。我们选择其中一些模块依次重点介绍来让读者对我们所验证的程序有个总体的认识。

1. SwitchEntry 是整个程序的入口，他会检查变量 SwitchFlag 来判断是否需要进行任务切换，如果需要则跳转到代码块 Window_OK 继续执行，否则直接退出。
2. Window_OK 会检查当前任务是否为null（当前任务可能是null，如果任务调度发生在当前任务被删除之后），如果当前任务是null，那么程序将跳转到 Adjust_CWP 模块去执行。Adjust_CWP 会调整当前窗口指针 `cwp`，让它指向第一个有效的窗口（即：无效窗口的上一个窗口），这么做实际上是清空循环栈（寄存器窗口）中的内容，因为当前任务已经并不存在，无

需再保存寄存器窗口中的内容。如果当前任务不是 null，那么我们会调用 reg_save 函数保存部分寄存器的内容到当前任务的任务控制块 TCB 中，然后进入代码块 Save_UsedWindow 去保存寄存器窗口中的内容（即我们模型中的帧链表 F）。

3. Save_UsedWindows 负责保存寄存器窗口中的内容（除当前窗口）到当前任务的内存栈空间内。
4. Switch_NewContext 会恢复保存在新任务任务控制块中的寄存器的值以及栈中保存的寄存器窗口的内容，并将新任务设置为当前任务。

相比与验证 x86 和 ARM 程序实现的任务上下文切换程序，这里验证的难点在于我们除了要保存寄存器文件中的内容，还需要保存寄存器窗口中内容。我们之前介绍过， SPARCv8 利用类似于循环栈的寄存器窗口机制来实现高效的运行时上下文的管理，从而避免直接将运行时上下文保存到栈中。Save_Windows 模块负责实现寄存器窗口中内容的保存，它的实现是一个循环，首先检查前一个窗口是否是有效窗口，有效则说明该窗口保存了运行时上下文，然后通过窗口旋转机制将该窗口变为当前窗口，并将其保存在对应的栈内存空间，并重复该操作直到前一个窗口为无效窗口（我们将无效窗口视为栈底）。

5.2　程序规范

我们在这一节给出程序的规范，如下所示，我们定义了该任务上下文切换程序主要功能模块的前断言和后断言。他们都带着 5 个参数：（1）当前任务的编号 t_c；（2）新任务的编号 t_n；（3）地址 SwitchFlag 中保存的值 $flag$；（4）表示当前任务上下文内容的 env；（5）表示新任务保存在内存中的上下文 nst。注意我们这里为了方便描述参数的意义将程序规范的参数列表（第4.2节图4.4中valList的定义）表示为多元组的形式。

$$
\begin{aligned}
\mathsf{a}_{pre}(t_c, t_n, flag, env, nst) \stackrel{\text{def}}{=}\ & \mathsf{Env}(env) * (\mathsf{SwitchFlag} \mapsto flag) * (\mathsf{TaskNew} \mapsto t_n) \\
& * (flag = \text{false} \vee (\mathsf{CurT}(t_c, \_, env) * \mathsf{NoCurT}(t_n, nst) \wedge |nst.F| = 2))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{a}_{post}(t_c, t_n, flag, env, nst) \stackrel{\text{def}}{=}\ & \exists env'.\ \mathsf{Env}(env') * (\mathsf{SwitchFlag} \mapsto \text{false}) * (\mathsf{TaskNew} \mapsto t_n) \\
& (flag = \text{false} \wedge \mathsf{p\_env}(env) = \mathsf{p\_env}(env') \\
& \wedge (\mathsf{CurT}(t_n, nst, env') \wedge \mathsf{p\_env}(env') = nst) \\
& * \mathsf{NoCurT}(t_c, \mathsf{p\_env}(env)))
\end{aligned}
$$

在程序规范中，我们使用 Env(env) 描述当前的寄存器状态，包括寄存器文件和寄存器窗口（帧链表）。地址 TaskNew 中保存了新任务的任务号 t_n。而地址 SwitchFlag 中保存了一个标记 $flag$ 来判断是否需要进行任务调度，如果

$$\begin{aligned}
\mathsf{Env}(env) \stackrel{\text{def}}{=}\ & ((\mathsf{global} \mapsto \mathrm{fm}_g) * (\mathsf{out} \mapsto \mathrm{fm}_o) * (\mathsf{local} \mapsto \mathrm{fm}_l) * (\mathsf{in} \mapsto \mathrm{fm}_i) * (\mathtt{Y} \mapsto v_y) \\
& * (\mathtt{cwp} \mapsto (\!| w_{id}, F |\!)) * (\mathtt{wim} \mapsto 2^n)) \wedge \mathsf{wfwin}(w_{id}, n, F) \\
& \text{where}\ \ env = (\mathrm{fm}_g, \mathrm{fm}_o, \mathrm{fm}_l, \mathrm{fm}_i, v_y, w_{id}, n, F)
\end{aligned}$$

$$\begin{aligned}
\mathsf{CurT}(t_c, nst, env) \stackrel{\text{def}}{=}\ & (\mathsf{TaskCur} \mapsto t_c) * \mathsf{context}(t_c, nst) * \mathsf{StkLoc}(\mathrm{fm}_i[6], F) \\
& \text{where}\ \ env = (\mathrm{fm}_o, \mathrm{fm}_l, \mathrm{fm}_i, v_y, w_{id}, n, F)
\end{aligned}$$

$$\begin{aligned}
\mathsf{NoCurT}(t_n, nst) \stackrel{\text{def}}{=}\ & (\mathsf{context}(t_n, nst) * \mathsf{Stack}(\mathrm{fm}_i[6], F)) \\
& \wedge (\exists\, i.\ |F| = 2{\times}i \wedge 0 \leqslant i \leqslant N-1) \\
& \text{where}\ \ nst = (\mathrm{fm}_g, \overline{v}_l, \mathrm{fm}_i, v_y, F)
\end{aligned}$$

$$\begin{aligned}
\mathsf{context}(t, nst) \stackrel{\text{def}}{=}\ & ((t{+}ofs_g) \mapsto \mathrm{fm}_g[0]) * \cdots * ((t{+}ofs_g{+}7) \mapsto \mathrm{fm}_g[7]) \\
& * ((t{+}ofs_l) \mapsto \overline{v}_l[0]) * \cdots * ((t{+}ofs_l{+}3) \mapsto \overline{v}_l[3]) \\
& * ((t{+}ofs_i) \mapsto \mathrm{fm}_i[0]) * \cdots * ((t{+}ofs_i{+}7) \mapsto \mathrm{fm}_i[7]) * ((t{+}ofs_Y) \mapsto v_y) \\
& \text{where}\ \ nst = (\mathrm{fm}_g, \overline{v}_l, \mathrm{fm}_i, v_y, F)
\end{aligned}$$

$$\mathsf{StkLoc}(l, F) \stackrel{\text{def}}{=} \begin{cases}
\mathsf{StackFrame}(l, _, _) * \mathsf{Stack}(l', F') & \textbf{if}\ \ \mathrm{fm}_2[6] = l',\ F = \mathrm{fm}_1 :: \mathrm{fm}_2 :: F' \\
\mathsf{emp} & \textbf{if}\ \ F = \mathrm{nil} \\
\mathrm{false} & \textbf{otherwise}
\end{cases}$$

$$\mathsf{Stack}(l, F) \stackrel{\text{def}}{=} \begin{cases}
\mathsf{StackFrame}(l, \mathrm{fm}_1, \mathrm{fm}_2) * \mathsf{Stack}(l', F') & \textbf{if}\ \ \mathrm{fm}_2[6] = l',\ F = \mathrm{fm}_1 :: \mathrm{fm}_2 :: F' \\
\mathsf{emp} & \textbf{if}\ \ F = \mathrm{nil} \\
\mathrm{false} & \textbf{otherwise}
\end{cases}$$

$$\begin{aligned}
\mathsf{StackFrame}(l, \mathrm{fm}_1, \mathrm{fm}_2) \stackrel{\text{def}}{=}\ & (l \mapsto \mathrm{fm}_1[0]) * \cdots * (l{+}7 \mapsto \mathrm{fm}_1[7]) \\
& * (l{+}8 \mapsto \mathrm{fm}_2[0]) * \cdots * (l{+}15 \mapsto \mathrm{fm}_2[7])
\end{aligned}$$

$$\begin{aligned}
\mathsf{p_env}(env) \stackrel{\text{def}}{=}\ & \begin{cases}
(\mathrm{fm}_g, \overline{v}_l, \mathrm{fm}_i, v_y, F) & \textbf{if}\ \ w_{id} \neq n \\
(\mathrm{fm}_g, \overline{v}_l, \mathrm{fm}_i, v_y, F \cdot (\mathrm{fm}_o :: \mathrm{nil})) & \textbf{if}\ \ w_{id} = n
\end{cases} \\
& \text{where}\ \ env = (\mathrm{fm}_g, \mathrm{fm}_o, \mathrm{fm}_l, \mathrm{fm}_i, v_y, w_{id}, n, F), \\
& \qquad \overline{v}_l = \mathrm{fm}_l[0] :: \mathrm{fm}_l[1] :: \mathrm{fm}_l[2] :: \mathrm{fm}_l[3] :: \mathrm{nil}
\end{aligned}$$

$$\begin{aligned}
\mathsf{wfwin}(w_{id}, n, F) \stackrel{\text{def}}{=}\ & ((w_{id} = n \wedge |F| = 2{\times}N{-}3) \vee \\
& (w_{id} \neq n \wedge |F| = 2{\times}((n{+}N{-}w_{id}{-}1)\%N))) \wedge 0 \leqslant w_{id}, n \leqslant N
\end{aligned}$$

图 5.2　程序规范中的辅助定义

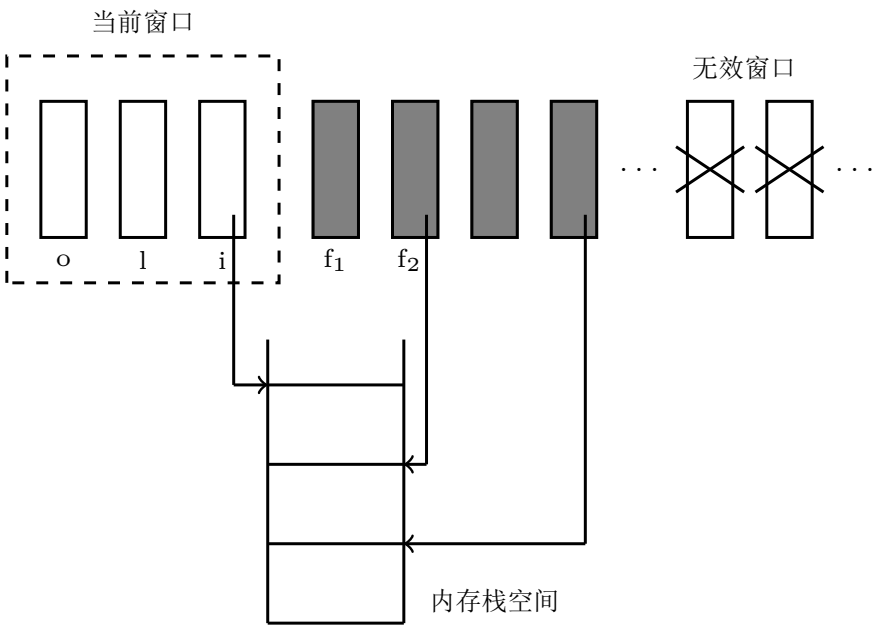

图 5.3　用于保存帧链表的栈空间

flag=false，则说明不需要进行任务调度，否则需要进行任务调度。对于需要进行任务调度的情况，我们需要描述当前任务 t_c 的状态（使用 CurT(t_c, _, *env*) 表示），即当前任务任务控制块 TCB 中用于保存运行时上下文的内存和当前任务的栈空间；以及新任务 t_n 的状态（使用 NoCurT(t_n, *nst*) 表示），这里 *nst* 表示新任务 t_n 保存在内存中的上下文，由图5.2中给出的 NoCurT(t_n, *nst*) 定义可知，我们会保存 global 寄存器，$\%l_0 \sim \%l_3$ 四个 local 寄存器，in 寄存器，Y 寄存器以及寄存器窗口 F，注意，这里 F 并不表示完整的帧链表，只是表示临时保存了任务运行时上下文，需要被上下文切换程序保存到内存中的一个帧链表的前缀。对于新任务 t_n，这里因为所验证的任务上下文切换程序只恢复寄存器窗口中栈顶窗口的内容，所以只需要描述帧链表的头两个帧即可，所以我们用 $|nst.F| = 2$ 表示所描述帧链表前缀的长度为 2。

对比前断言 a_{pre} 和后断言 a_{post} 可以发现，程序执行结束时，任务 t_n 将会变为当前任务（CurT(t_n, *nst*, *env*′），*env*′ 表示被恢复的任务 t_n 的上下文），并且保存在内存中的 t_n 的上下文将会被恢复（用 Env(*env*′) 表示），我们使用 p_env(*env*′) = *nst* 表示当前寄存器状态是新任务上下文 *nst* 恢复后的结果，图5.2中给出了 p_env(*env*′) 的定义，p_env(*env*′) 返回 *env*′ 中需要保存到内存中的值。对于原来的任务 t_c，它从当前任务变为非当前任务，进入上下文切换程序时的上下文 *env* 被保存在 t_c 的任务控制块和内存栈中（表示为断言 NoCurT(t_c, p_env(*env*))）。

图5.2中展示了我们在定义程序规范时用到的辅助定义，context(t, *nst*) 用于描述任务 t 的任务控制块中保存任务运行时上下文的内存，*nst* 是一个五元组

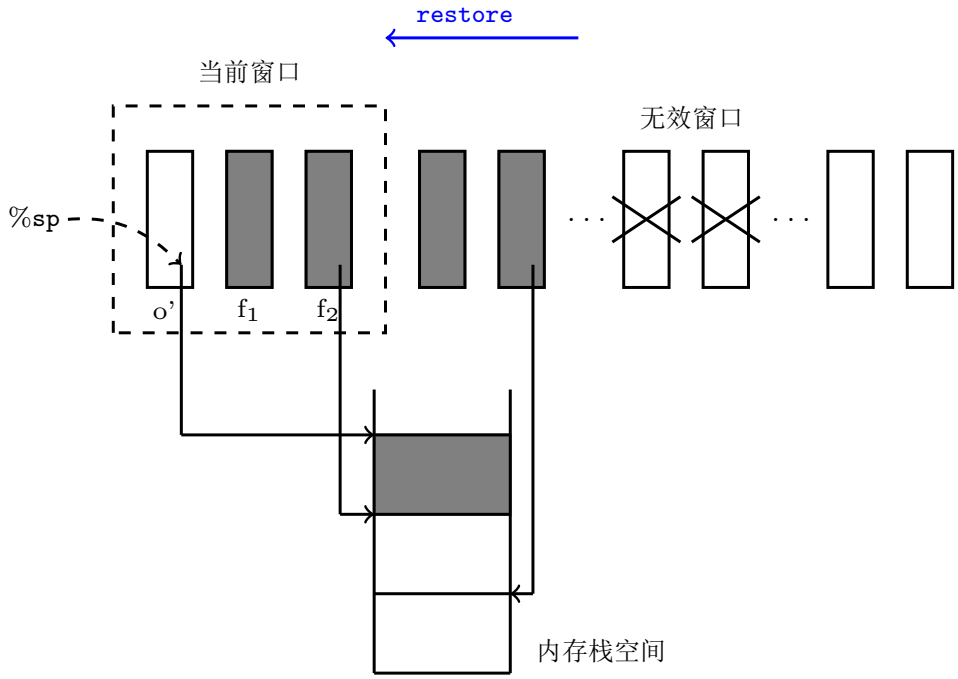

图 5.4　保存寄存器窗口中的内容到栈中

$(\mathrm{fm}_g, \overline{v}_l, \mathrm{fm}_i, v_y, F)$ 表示需要被保存的运行时上下文的内容，其中帧链表的内容 F 不保存在任务控制块中；global 寄存器的内容 fm_g 将保存在起始地址为 $(t + ofs_g)$ 的内存空间；四个 local 寄存器“$\%\mathtt{l}_0 \sim \%\mathtt{l}_3$”的内容 $\overline{v}$，保存在起始地址为 $(t+ofs_l)$ 的内存空间；in 寄存器的内容 fm_i，保存在起始地址为 $(t + ofs_i)$ 的内存空间；而地址为 $(t + ofs_Y)$ 内存单元用于保存Y寄存器的内容 v_y。StkLoc(l, F) 描述用于保存帧链表前缀 F 的栈空间，l 是栈顶地址，注意 StkLoc(l, F) 只描述栈空间不描述栈中的内容，因为这一部分内容是预留用于保存 F 的，所以原来内存中的值我们并不关心。我们用图5.3 来介绍 StkLoc(l, F) 所描述的栈空间，我们之前介绍过， SPARCv8 通过寄存器窗口机制来实现高效的运行时上下文的管理来避免直接将运行时上下文保存到栈中，而每个临时保存了当前线程运行时上下文的寄存器窗口都有一块对应的内存栈帧空间，用于在需要的时候将临时保存在寄存器窗口中的运行时上下文保存到内存中，每一个保存了运行时上下文的窗口所对应的栈帧的起始地址保存在它的上一个窗口中。我们通过图5.4 来介绍保存寄存器窗口的方法来对 StkLoc(l, F) 所描述的内存空间有更好的认识，当前一个窗口是有效时，说明该窗口保存了运行时上下文，我们通过指令 `restore` 将该窗口变为当前窗口，并将当前窗口中 local 和 in 寄存器的内容（图5.4中当前窗口标记为灰色的部分）保存到对应的栈帧中，所保存到的栈帧的位置，由当前窗口的 `%sp` 寄存器（即：$\mathtt{r}_{14}$ 寄存器）给出。Stack(l, F) 描述了起始地址为 l，保存了帧链表 F 的栈空间。在验证工作中，我们使用 StkLoc(l, F) 描述保存寄存器

窗口前的栈空间，用 Stack(l, F) 描述保存了寄存器窗口后的栈空间。此外，我们定义 wfwin(w_{id}, n, F) 来根据当前窗口编号 w_{id}，以及无效窗口编号 n 计算保存了运行时上下文的帧链表 F 的长度。我们在第3.1节中之间介绍过，寄存器窗口类似与一个循环栈，我们通常设置一个无效窗口来防止栈溢出。如果当前窗口即为无效窗口（即：$w_{id} = n$），说明发生了栈溢出，所以此时除当前窗口以外的所有寄存器窗口都保存了运行时上下文，所以，除当前窗口以外的寄存器窗口中的内容都需要被保存到栈中（即：$|F| = 2 \times N - 3$，N 为窗口总数）；如果当前窗口不是无效窗口（即：$w_{id} \neq n$），则说明从第 $(w_{id}+1)\%N$ 号窗口到第 $(n+N-1)\%N$ 号窗口（窗口号循环递增）都保存了运行时上下文，总计：$((n+N-w_{id}-1)\%N)$ 个窗口，所以保存了运行时上下文的帧链表 F 的长度为：$2 \times ((n+N-w_{id}-1)\%N)$。

5.3　Coq 验证工作

我们在证明过程中去掉了和中断处理和浮点型寄存器相关的操作，因为我们没有在建模和设计验证逻辑时考虑他们。我们共验证了约 250 行 SPARCv8 汇编代码，编写了共计 6690 行 Coq 证明脚本。表5.1 中给出了我们验证任务上下文切换程序每个模块的 Coq 证明脚本行数的统计 ①。

表 5.1　任务上下文切换程序 Coq 证明代码量统计

验证模块	Coq 证明行数统计
SwitchEntry	885
Window_OK	679
Adjust_CWP	461
Save_UsedWindows	3358
Switch_NewContext	885
reg_save	312
reg_restore	293

从 Coq 代码行数的统计中可以看出，正如我们之前所介绍的，相比与验证其它汇编指令所实现的任务上下文切换程序，这里验证的难点在于我们除了需要保存寄存器文件中的内容，还需要保存寄存器窗口中的内容，而 Save_UsedWindows

① 由于保密协议的原因，我们无法展示所验证操作系统任务上下文切换程序的完整 Coq 代码，也没有给出任务上下文切换程序每个模块的 SPARCv8 代码具体实现行数。

模块则负责保存寄存器窗口中的内容，我们需要证明：

$$\{\mathsf{StkLoc}(l, F) * \mathtt{cwp} \mapsto (\!| w_{id}, F |\!) * \cdots\}$$
$$\mathsf{Save\_UsedWindows}$$
$$\{\mathsf{Stack}(l, F) * (\exists\, w'_{id},\ \mathtt{cwp} \mapsto (\!| w'_{id}, \_ |\!)) * \cdots\}$$

即：如果执行 Save_UsedWindows 模块之前，帧链表 F 中保存了当前任务保存在寄存器窗口中的运行时上下文，那么执行完 Save_UsedWindows 模块之后帧链表序列 F 被保存到当前任务的栈内存空间中（即：断言 Stack(l, F) 成立）。我们共编写 3358 行 Coq 证明脚本来完成该模块的验证工作。

代码 5.1　在 Coq 中证明 SwitchEnty 模块的正确性

```
Theorem SwitchEntyProof :
   forall vl,
     spec |- {{ switch_entry_pre vl }}
                f0 : switch_entry_code
              {{ switch_entry_post vl }}.
Proof.
  ... ...
end.
```

代码5.1中给出了 Coq 中证明 SwitchEntry 模块代码正确性的定理“SwitchEntryProof”，该定理是下述定理 5.1在 Coq 中的表示：

定理 5.1 (SwitchEntry 模块的正确性)

$$\forall\, \iota.\, \Psi \vdash \{(\mathsf{a}_{pre}\ \iota,\ \mathsf{a}_{post}\ \iota)\}\ \mathtt{f}_0 :\ \mathsf{SwitchEntry}$$

因为 SwitchEntry 是整个任务上下文切换程序的入口代码块，所以，根据我们之前在第4.2节中对程序规范的介绍可知，程序规范的后断言描述的是当前函数结束时的程序状态，所以该代码块的程序规范即为整个任务上下文切换函数的程序规范 a_{pre} 和 a_{post}（定义在第5.2节中）。代码5.1中的“switch_entry_pre”即为前断言 a_{pre} 在 Coq 中的定义，而“switch_entry_post”是后断言 a_{post} 在 Coq 中的定义。“$\mathtt{f}_0$”是 SwitchEntry 模块的起始地址，而“switch_entry_code”是代码块 SwitchEntry 在 Coq 中的定义。我们可以通过我们所定义的指令序列推理规则和指令推理规则来证明定理 5.1成立。

使用自动证明策略简化验证工作　Cao 等人[22] 在定理证明工具 Coq 中开发了一些用于 C 程序验证的自动证明策略，其中包括自动证明策略 sep_cancel 用于断言的自动推导。断言自动推导指的是自动证明“ $p \Rightarrow p'$ ”这类命题。我们在任务上下文切换程序的验证工作中复用了自动证明策略 sep_cancel，并在此基础上

$$\frac{w = w' \quad p \Rightarrow p'}{(\mathtt{rn} \mapsto w) * p \Rightarrow (\mathtt{rn} \mapsto w') * p'}\ L3 \qquad \frac{w = w' \quad p \Rightarrow p'}{(\triangleright_t \mathtt{sr} \mapsto w) * p \Rightarrow (\triangleright_t \mathtt{sr} \mapsto w') * p}\ L4$$

$$\frac{w = w' \quad p \Rightarrow p'}{(\triangleright_0 \mathtt{sr} \mapsto w)\downarrow * p \Rightarrow (\mathtt{sr} \mapsto w') * p'}\ L5$$

$$\frac{w = w' \quad p \Rightarrow p'}{(\triangleright_{t+1} \mathtt{sr} \mapsto w)\downarrow * p \Rightarrow (\triangleright_t \mathtt{sr} \mapsto w') * p'}\ L6$$

$$\frac{w = w' \quad p \Rightarrow p'}{(\mathtt{rn} \mapsto w)\downarrow * p \Rightarrow (\mathtt{rn} \mapsto w') * p'}\ L7 \qquad \frac{w = w' \quad p \Rightarrow p'}{(l \mapsto w)\downarrow * p \Rightarrow (l \mapsto w') * p'}\ L8$$

$$\frac{p_1 \downarrow \Rightarrow p_1' \quad p_2 \downarrow \Rightarrow p_2'}{(p_1 * p_2)\downarrow \Rightarrow p_1' * p_2'}\ L9$$

图 5.5　扩充的用于断言自动推理的规则

做了一些扩展。sep_cancel 通过反复使用一些推理规则来自动消去断言蕴含式两侧断言 p 和 p' 中存在蕴含关系的子断言，例如下面的规则 $L1$ 和 $L2$：

$$\frac{p \Rightarrow p'}{p * q \Rightarrow p' * q}\ L1 \qquad \frac{w = w' \quad p \Rightarrow p'}{(l \mapsto w) * p \Rightarrow (l \mapsto w') * p'}\ L2$$

规则 $L1$ 保证了我们在证明 $p * q \Rightarrow p' * q$ 时，可以消去断言蕴涵式两侧相同的子断言，即断言 q，从而只需证明 $p \Rightarrow p'$ 即可。规则 $L2$ 保证了如果所证明的断言蕴含关系两边都有描述地址 l 的内存单元，那么只要相同内存单元中保存的值相同（即：$w = w'$），那么我们就可以消去蕴含关系两边描述地址 l 所对应内存单元的断言，只需证明 $p \Rightarrow p'$。

我们根据我们为 SPARCv8 所设计的验证逻辑，我们在图5.5 中定义了规则 $L3 \sim L9$ 作为对证明策略 sep_cancel 中所使用推理规则的扩充。规则 $L3$ 用于消去断言蕴含式两侧描述相同寄存器单元 rn 的子断言，而规则 $L4$ 用于消去断言蕴含式两侧描述对同一特殊寄存器 sr 的延时写入操作的子断言。规则 $L5 \sim L9$ 则用来处理断言蕴含式左侧存在描述执行一步延时写入转移之后程序状态的断言 $(\_)\downarrow$ 的情况，我们可以根据第4.1节中的引理4.1以及断言的语义来证明这些推理规则的可靠性。扩充了 sep_cancel 所使用的推理规则后，下述命题正确性的证明可以直接通过调用自动证明策略 sep_cancel 完成：

$$(\triangleright_3 \mathtt{Y} \mapsto w_1 * \triangleright_0 \mathtt{wim} \mapsto w_2 * \%\mathtt{i}_0 \mapsto w_3)\downarrow \Longrightarrow \triangleright_2 \mathtt{Y} \mapsto w_1 * \mathtt{wim} \mapsto w_2 * \%\mathtt{i}_0 \mapsto w_3$$

由此可见，利用自动证明策略可以大大简化我们的验证工作，避免很多繁琐而重复的工作，提高了证明效率。

5.4　本章小结

我们在这一章介绍了我们使用我们所设计的 SPARCv8 程序验证逻辑验证某航天嵌入式操作系统中任务上下文切换程序主要功能模块的验证工作。我们在第5.1节中介绍了所验证程序的结构，在第5.2节中给出了程序规范的形式化定义，最后，在第5.3节介绍了相关的 Coq 证明工作以及通过自动证明策略来简化我们的验证工作。当然，这里我们并没有验证完整的任务上下文切换程序，只验证了它的一个主要功能模块，因为我们的验证逻辑并不支持调用任务上下文切换程序（因为调用该方法时保存在 $\mathtt{r}_{15}$ 寄存器的值，与该方法返回时 `retl` 指令计算返回地址时 $\mathtt{r}_{15}$ 中的值不同），这里我们主要关注的是上下文切换程序中的寄存器窗口管理模块和一些内部函数。另外，我们所验证的任务上下文切换程序的主要功能模块，涉及到函数调用，以及 SPARCv8 的三个主要特性：延时跳转、特殊寄存器的延时写入和寄存器窗口机制，该工作很好的说明了我们的验证逻辑对函数调用和 SPARCv8 的三个特性验证的支持。相关工作的完善和扩展我们放在第6章中解决。

第 6 章　SPARCv8 程序的精化验证

我们将在这一章介绍 SPARCv8 程序的精化验证。我们在第4章介绍了我们为 SPARCv8 设计的验证逻辑，并在第5章介绍了用我们的验证逻辑验证某航天嵌入式操作系统中任务上下文切换程序的主要功能模块。不过，本研究课题的最终目标是希望证明 SPARCv8 汇编程序和其对应的抽象汇编原语之间存在 “上下文精化关系”，来完善目前的操作系统验证工作。所以，只有之前的验证逻辑其实并不足够，并且我们的验证逻辑并不支持之前所验证任务上下文切换程序的调用（因为调用该方法时保存在 $\mathtt{r}_{15}$ 寄存器的值，与该方法返回时 `retl` 指令计算返回地址时 $\mathtt{r}_{15}$ 中的值不同）。这一章，我们将介绍如何验证 SPARCv8 汇编程序和其所对应的抽象汇编原语之间的 “上下文精化关系”。

6.1　验证方法概述

本节将从整体上对我们的 SPARCv8 程序精化验证的工作做简要介绍。本工作的主要目的是希望证明 SPARCv8 汇编程序和其所对应的抽象汇编原语之间存在 “上下文精化关系”，所谓上下文精化关系，可以简单理解为如下形式：

$$\forall\, \mathbb{P}.\ \mathbb{P}[C_{\text{as}}] \sqsubseteq \mathbb{P}[\iota]$$

其中，我们用 ι 表示抽象汇编原语，C_{as} 表示抽象汇编原语 ι 的汇编代码实现，我们称 $\mathbb{P}$ 为抽象汇编原语和汇编实现的 “执行上下文” 或者调用抽象汇编原语和汇编实现的 “客户端程序”。所以，汇编原语实现 C_{as} 和汇编原语 ι 之间存在上下文精化关系，指的是对于任意的客户端程序 $\mathbb{P}$，调用汇编原语实现 C_{as} 的程序 $\mathbb{P}[C_{\text{as}}]$ 和调用抽象汇编原语 ι 的程序 $\mathbb{P}[\iota]$ 之间存在精化关系。而精化关系可以理解为程序 $\mathbb{P}[C_{\text{as}}]$ 的行为是程序 $\mathbb{P}[\iota]$ 行为的子集。这就保证了调用 C_{as} 的程序不会比调用 ι 的程序产生更多的行为，这样我们就可以在任何调用汇编实现 C_{as} 的地方使用抽象汇编原语 ι 来代替它。而想要形式化地验证汇编原语实现和汇编原语之间的 “上下文精化关系”，必须需要解决如下两个问题：

选择什么样的程序作为客户端程序？

操作系统验证框架[1] 中对 C 语言的子集进行了形式化的建模，并将 C 程序作为调用抽象汇编原语的客户端程序。所以，这里我们同样选择 C 程序作为调用抽象汇编原语的客户端程序。不过，如果同时将 C 语言程序作为调用汇编原语实现的客户端程序，则需要将 C 语言和汇编语言在统一的机器模型上进行建模，而汇编程序会操作寄存器和栈，这会导致语义变得非常复杂。所以，这里我们的

方法是：选择 C 程序编译生成的汇编程序作为调用抽象原语汇编实现的客户端程序。而汇编原语实现与汇编原语之间的上下文精化关系可以表示为如下形式：

$$\forall \mathbb{P},\ \mathbb{A}.\ \mathbb{A} \preccurlyeq \mathbb{P} \Longrightarrow \mathbb{A}[C_{\text{as}}] \sqsubseteq \mathbb{P}[\iota]$$

这里，$\mathbb{P}$ 是调用抽象汇编原语的客户端程序，我们将它称为高层客户端程序，它是由 C 语言实现的。而 $\mathbb{A}$ 是调用抽象汇编原语实现的客户端程序，我们将它称为底层客户端程序，它是汇编语言实现的，即 SPARCv8 程序。底层客户端程序 $\mathbb{A}$ 是高层客户端程序 $\mathbb{P}$ 编译生成的，我们希望编译器能够建立源程序和目标程序之间的模拟关系 $\mathbb{A} \preccurlyeq \mathbb{P}$，不过目前主要的编译器验证工作中验证的 CompCert 编译器并不支持编译生成 SPARCv8 目标代码，所以我们暂时假设二者之间存在模拟关系 $\mathbb{A} \preccurlyeq \mathbb{P}$，至于编译器能否保证该模拟关系，以后工作再去完善。

使用什么方法证明汇编实现和抽象汇编原语之间的上下文精化关系？

确定了底层和高层程序中的“客户端程序”之后，我们其次所关注的则是如何证明底层和高层程序之间的精化关系。常用的证明程序精化关系的手段是定义底层和高层程序之间的“模拟关系”来建立底层和高层程序执行过程中程序状态之间的对应关系。这里我们同样采用定义模拟关系的方法来证明底层和高层程序间的精化关系：

$$C_{\text{as}} \leqslant \iota \Longrightarrow (\forall \mathbb{P},\ \mathbb{A}.\ \mathbb{A} \preccurlyeq \mathbb{P} \Longrightarrow \mathbb{A}[C_{\text{as}}] \sqsubseteq \mathbb{P}[\iota])$$

我们定义了汇编原语实现和抽象汇编原语之间的模拟关系“$C_{\text{as}} \leqslant \iota$”，该模拟关系具有“可组合性”, 即汇编原语实现和抽象汇编原语之间的模拟关系“$C_{\text{as}} \leqslant \iota$”可以和客户端程序之间的模拟关系“$\mathbb{A} \preccurlyeq \mathbb{P}$”组合起来，得到完整的底层和高层程序之间的模拟关系。所以，证明汇编原语实现与抽象汇编原语之间存在上下文精化关系的证明思路大致如下：

$$\begin{array}{l} C_{\text{as}} \leqslant \iota \\ \quad + \qquad \Longrightarrow \mathbb{A}[C_{\text{as}}] \preccurlyeq \mathbb{P}[\iota] \Longrightarrow \mathbb{A}[C_{\text{as}}] \sqsubseteq \mathbb{P}[\iota] \\ \mathbb{A} \preccurlyeq \mathbb{P} \end{array}$$

我们首先证明汇编实现 C_{as} 和抽象汇编原语 ι 之间存在模拟关系“$C_{\text{as}} \leqslant \iota$”，再由该模拟关系与底层和高层客户端程序之间模拟关系“$\mathbb{A} \preccurlyeq \mathbb{P}$”之间的可组合性，得到完整的底层和高层程序之间的模拟关系“$\mathbb{A}[C_{\text{as}}] \preccurlyeq \mathbb{P}[\iota]$”。最终由完整的底层和高层程序之间的模拟关系得到底层和高层程序之间的精化关系。

我们将在后续章节中详细介绍验证汇编原语实现和汇编原语之间上下文精化关系的工作。第6.2节将介绍高层带抽象汇编原语的 C 语言，即：我们的

高层程序 $\mathbb{P}[\iota]$；在第 6.3节介绍底层 SPARCv8 语言，即：调用汇编原语实现的底层 SPARCv8 程序 $\mathbb{A}[C_{\text{as}}]$；并在第6.4节给出刻画最终目标的定义和精化关系 $\mathbb{A}[C_{\text{as}}] \sqsubseteq \mathbb{P}[\iota]$ 的定义；最后在第6.5节介绍我们定义的能够保证汇编原语实现和汇编原语之间上下文精化关系的模拟关系，即：本节所提到的 "$C_{\text{as}} \leqslant \iota$"，并证明了一个 SPARCv8 实现的任务上下文切换程序和 `switch` 原语之间满足我们所定义的模拟关系。

6.2　高层带抽象汇编原语的 C 语言

本节将定义带抽象汇编原语的 C 语言作为我们精化验证工作中的高层语言。

$$
\begin{array}{rrcl}
\text{(HProg)} & \mathbb{P} & ::= & \textbf{let}\ (\Gamma, \Omega)\ \textbf{in}\ \mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n \\
\text{(HCode)} & \Gamma & ::= & \{\mathtt{f}_1 \rightsquigarrow fd_1, \cdots, \mathtt{f}_n \rightsquigarrow fd_n\} \qquad \text{(FunDef)}\ fd ::= (ty, \mathcal{D}_1, \mathcal{D}_2, \mathtt{s}) \\
\text{(PrimSet)} & \Omega & ::= & \{\mathtt{f}_{\text{sw}} \rightsquigarrow (\mathtt{switch}, \text{nil})\} \qquad \text{(Prim)}\ \iota ::= \mathtt{switch} \mid \cdots \\
\text{(Stmt)} & \mathtt{s} & ::= & \mathtt{f}(\overline{\mathsf{e}}) \mid x := \mathtt{f}(\overline{\mathsf{e}}) \mid x := \mathsf{e} \mid \mathtt{s}; \mathtt{s} \mid \textbf{skip} \mid \textbf{return}\ \mathsf{e} \\
 & & \mid & \textbf{if}\,(\mathsf{b})\,\textbf{then}\,\{\mathtt{s}\}\,\textbf{else}\,\{\mathtt{s}\}\textbf{while}\,(\mathsf{b})\,\{\mathtt{s}\} \mid \textbf{print}\ \mathsf{e} \mid \cdots \\
\text{(Expr)} & \mathsf{e} & ::= & x \mid v \mid \mathsf{e} + \mathsf{e} \mid [\mathsf{e}] \\
\text{(BExpr)} & \mathsf{b} & ::= & \textbf{true} \mid \textbf{false} \mid \mathsf{e} = \mathsf{e} \mid !\,\mathsf{b} \\
\text{(DecList)} & \mathcal{D} & ::= & \text{nil} \mid (x, ty) :: \mathcal{D} \qquad \text{(Ident)}\ x \in \mathbb{Z} \\
\text{(Type)} & ty & ::= & \mathtt{Tint} \mid \mathtt{Tptr} \mid \mathtt{Tvoid} \\
\text{(Val)} & v & ::= & n \mid (b, ofs) \mid \mathtt{null} \mid \mathtt{undef} \qquad \text{where } n \in \mathbb{Z}
\end{array}
$$

图 6.1　带抽象汇编原语的 C 语言语法

我们在图6.1中定义了带抽象汇编原语的 C 语言语法。程序 $\mathbb{P}$ 包括：高层代码 Γ，抽象原语集合 Ω，以及 n 个线程的执行入口 $\mathtt{f}_1$ 至 $\mathtt{f}_n$。高层代码是一个从函数标识符到函数的部分映射（partial mapping）。C 语言中的函数被形式化定义为一个四元组：返回值类型 ty，形参列表 $\mathcal{D}_1$，局部变量列表 $\mathcal{D}_2$，以及函数体 $\mathtt{s}$。参数列表和局部变量列表都是变量名和变量类型二元组的序列。

程序语句 $\mathtt{s}$ 可以是赋值语句 $x := \mathsf{e}$，函数调用语句 $\mathtt{f}(\overline{\mathsf{e}})$、$x := \mathtt{f}(\overline{\mathsf{e}})$，条件分支语句 $\textbf{if}\,(\mathsf{e})\,\textbf{then}\,\{\mathtt{s}_1\}\,\textbf{else}\,\{\mathtt{s}_2\}$，循环语句 $\textbf{while}\,(\mathsf{e})\,\{\mathtt{s}\}$ 等等。表达式 e 可以是程序变量 x，常数 v，加运算表达式 $\mathsf{e} + \mathsf{e}$，或者地址访问表达式 $[\mathsf{e}]$。常数 v 可以是整数 n，指针 (b, ofs)，空指针 `null` 或者未定义 `undef`。这里的内存模型借鉴了 CompCert 中的内存模型[32]，内存地址被表示为内存块 b 和偏移地址 ofs 的二元组。

抽象汇编原语集合 Ω 的类型是标识符到抽象汇编原语和参数类型列表二元

$$
\begin{array}{rrcl}
\text{(HWord)} & \mathbb{W} & ::= & ((\Gamma, \Omega), \theta) \\
\text{(CState)} & \theta & ::= & (T, t, \mathcal{K}, \Sigma) \\
\text{(ThrdPool)} & T & \in & \text{ThrdId} \rightharpoonup \text{Cont} \qquad \text{(ThrdId)}\ t \in \mathbb{Z} \\
\text{(Core)} & \mathcal{K} & ::= & (\mathtt{s}, \kappa, \mathrm{E}) \qquad \text{(ContStk)}\ \kappa ::= \circ \mid \mathtt{s} \cdot \kappa \\
\text{(CMem)} & \Sigma & ::= & (\mathrm{G}, M) \qquad \text{(HMem)}\ M ::= \text{Addr} \rightharpoonup \text{Val} \\
\text{(Addr)} & l & ::= & \text{Block} \times \mathbb{Z} \qquad \text{(Block)}\ b \in \mathbb{Z} \\
\text{(HMsg)} & \alpha & ::= & \tau \mid e \mid \mathtt{call}(\mathtt{f}, \overline{v}) \qquad \text{(Evt)}\ e ::= \mathtt{out}(v) \\
\text{(SymTbl)} & \mathrm{G},\ \mathrm{E} & \in & \text{Ident} \rightharpoonup \text{Addr} \times \text{Type} \\
\text{(PStep)} & \phi\longrightarrow & \in & \text{CState} \to \text{Prim} \times \text{list Type} \to \text{CState} \to \text{Prop}
\end{array}
$$

图 6.2　程序状态及抽象原语状态转移

组的映射，这里我们暂时只考虑 `switch` 原语，所以集合 Ω 中只有 `switch` 一条抽象原语。抽象汇编原语 ι 可以是用于线程切换的 `switch` 原语，也可以是其它功能的原语不过我们这里暂时不考虑。

我们在图6.2中定义了高层机器 $\mathbb{W}$，包括：高层代码 Γ 和抽象汇编原语集合 Ω 的二元组，以及一个用于描述程序状态的四元组 θ。我们将 θ 称为 C 语言程序状态，包括：线程池 T、当前线程号 t、当前线程的程序后继 $\mathcal{K}$、以及 C 语言内存状态 Σ。程序后继是一个由当前语句 s，后继语句 κ 和局部符号表 E 组成的三元组。而 C 语言的内存状态被定义为一个由全局符号表 G 和内存 M 组成的二元组。另外，我们还定义抽象原语状态转移的类型（PStep），它被定义成一个 C 程序状态、汇编原语和参数列表二元组、C 程序状态上的三元关系。对于消息 α 和事件 e 的介绍我们放在后面操作语义的介绍中介绍。

操作语义 图6.3中展示了带抽象汇编原语的 C 语言操作语言，包括：全局操作语义（图6.3 (a)），`switch` 原语操作语义（图6.3(b)），以及部分的线程局部操作语义（图6.3 (c)）。全局操作语义中，加载规则（Load）表示程序的初始化工作，我们根据 $\mathtt{f}_1$ 至 $\mathtt{f}_n$ 这 n 个程序入口依次创建 n 个线程编号 1 至 n，我们要求这 n 个函数入口都是合法的，即都在代码 Γ 中，并选择线程 t 作为起始线程；正常执行规则（τ-Step）表示当前线程走一步，并且不产生外部可见事件，我们定义 τ 表示空消息；事件执行规则（e-Step）表示当前线程走一步，并产生外部可见事件 e，外部可见事件 e 定义在图6.2中，主要用于刻画程序的行为；抽象汇编原语执行规则（ι-Step）表示当前程序发生了汇编原语的调用，当前线程执行产生消息 $\mathtt{call}(\mathtt{f}, \overline{v})$（参见图6.2中消息 msg 的定义），表示希望调用标识符 `f` 所对应的抽象原语并且参数为 $\overline{v}$，我们从汇编原语集合 Ω 中取出 `f` 所映射到的原语 ι 和参数类型 $\overline{ty}$，我们判断参数和参数的类型是匹配的（$\mathsf{tyms}(\overline{v}, \overline{ty})$ 表示参数列表 $\overline{v}$

$$\frac{T = \{1 \rightsquigarrow \mathcal{K}_1, \cdots n \rightsquigarrow \mathcal{K}_n\} \quad T(t) = \mathcal{K} \quad \text{for all } 1 \leqslant i \leqslant n,\ \mathcal{K}_i = (\mathtt{f}_i(\text{nil}), \circ, \varnothing) \quad \Gamma(\mathtt{f}) = (\mathtt{Tvoid}, \text{nil}, \_, \_)}{(\textbf{let } (\Gamma, \Omega) \textbf{ in } \mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n, \Sigma) \overset{load}{\Longrightarrow} ((\Gamma, \Omega), (T, t, \mathcal{K}, \Sigma))} \text{(Load)}$$

$$\frac{\Gamma \vdash (\mathcal{K}, \Sigma) \xrightarrow{\tau} (\mathcal{K}', \Sigma')}{((\Gamma, \Omega), (T, t, \mathcal{K}, \Sigma)) \overset{\tau}{\Longrightarrow} ((\Gamma, \Omega), (T, t, \mathcal{K}', \Sigma'))} (\tau - \text{Step})$$

$$\frac{\Gamma \vdash (\mathcal{K}, \Sigma) \xrightarrow{e} (\mathcal{K}', \Sigma)}{((\Gamma, \Omega), (T, t, \mathcal{K}, \Sigma)) \overset{e}{\Longrightarrow} ((\Gamma, \Omega), (T, t, \mathcal{K}', \Sigma))} (e - \text{Step})$$

$$\frac{\Gamma \vdash (\mathcal{K}, \Sigma) \xrightarrow{\mathtt{call}(\mathtt{f}, \overline{v})} (\mathcal{K}', \Sigma) \quad \Omega(\mathtt{f}) = (\iota, \overline{ty}) \quad \mathsf{tyms}(\overline{v}, \overline{ty}) \quad (T, t, \mathcal{K}', \Sigma, P)\, \phi \xrightarrow{(\iota, \overline{v})} (T', t', \mathcal{K}'', \Sigma', P')}{((\Gamma, \Omega), (T, t, \mathcal{K}, \Sigma)) \overset{\tau}{\Longrightarrow} ((\Gamma, \Omega), (T', t', \mathcal{K}'', \Sigma'))} (\iota - \text{Step})$$

$$\frac{\Gamma \vdash (\mathcal{K}, \Sigma) \xrightarrow{\tau} \textbf{abort}}{((\Gamma, \Omega), (T, t, \mathcal{K}, \Sigma)) \overset{\tau}{\Longrightarrow} \textbf{abort}} \text{(Abort)}$$

(a) 全局操作语义

$$\frac{[\![\mathsf{TaskNew}]\!]_{(\varnothing, \Sigma)} = (t', 0) \quad T(t') = \mathcal{K}' \quad T' = T\{t \rightsquigarrow \mathcal{K}\} \quad t' \neq t}{(T, t, \mathcal{K}, \Sigma)\, \phi \xrightarrow{(\mathtt{switch}, \text{nil})} (T', t', \mathcal{K}', \Sigma)} \text{(Switch)}$$

(b) `switch` 原语状态转移

$$\frac{\Gamma(\mathtt{f}) = (\_, \mathcal{D}_1, \mathcal{D}_2, \mathrm{s}) \quad [\![\overline{\mathsf{e}}]\!]_{(\mathrm{E}, \Sigma)} = \overline{v} \quad \mathsf{args\_alloc}(\overline{v}, \mathcal{D}_1 \cdot \mathcal{D}_2, \mathrm{E}, \Sigma, \mathrm{E}', \Sigma')}{\Gamma \vdash ((\mathtt{f}(\overline{\mathsf{e}}), \kappa, \mathrm{E}), \Sigma) \xrightarrow{\tau} ((\mathrm{s}, \kappa, \mathrm{E}'), \Sigma')} \text{(IntCall)}$$

$$\frac{\mathtt{f} \notin \mathrm{dom}(\Gamma) \quad [\![\overline{\mathsf{e}}]\!]_{(\mathrm{E}, \Sigma)} = \overline{v}}{\Gamma \vdash ((\mathtt{f}(\overline{\mathsf{e}}), \kappa, \mathrm{E}), \Sigma) \xrightarrow{\mathtt{call}(\mathtt{f}, \overline{v})} ((\textbf{skip}, \kappa, \mathrm{E}), \Sigma)} \text{(ExtCall)}$$

$$\frac{[\![\mathsf{e}]\!]_{(\mathrm{E}, \Sigma)} = v}{\Gamma \vdash ((\textbf{print } \mathsf{e}, \kappa, \mathrm{E}), \Sigma) \xrightarrow{\mathtt{out}(v)} (\textbf{skip}, \kappa, \Sigma)} \text{(Print)}$$

$$\frac{}{\Gamma \vdash ((\textbf{skip}, \mathrm{s} \cdot \kappa, \mathrm{E}), \Sigma) \xrightarrow{\tau} ((\mathrm{s}, \kappa, \mathrm{E}), \Sigma)} \text{(Cont)}$$

(c) 部分线程局部操作语义

图 6.3　带抽象汇编原语的 C 语言操作语义

和类型列表 ty 是匹配的，定义在附录A的图A.4中），并执行抽象原语 ι 对应的状态转移（$\_\ \phi \xrightarrow{(\iota, \overline{v})}\ \_$）；中止规则（Abort）展示了程序中止的情况，如果当前线程中止，则整个程序中止。当前线程中止定义为：

$$\Gamma \vdash (\mathcal{K}, \Sigma) \xrightarrow{\tau} \textbf{abort} \overset{\text{def}}{=} \neg \exists\, \mathcal{K}', \Sigma'.\, \Gamma \vdash (\mathcal{K}, \Sigma) \xrightarrow{\tau} (\mathcal{K}', \Sigma')$$

对于抽象原语所对应的程序状态转移，因为暂时只考虑 `switch`，所以这里在图6.3 (b) 中只给出了 `switch` 原语对应的状态转移。它从全局变量 TaskNew 中获取新线程的编号 t'，执行线程的切换（此处 TaskNew 是一个预定义的全局变量）。

图6.3 (c) 中定义了线程局部操作语义来描述单条线程的执行，这里展示了部分线程局部操作语义来说明线程何时会产生外部可见事件 e，及如何处理调用汇编原语的情况（完整的线程局部操作语义参见附录A中图A.1和图A.2）。内部函数调用规则（IntCall）展示了当前线程调用内部函数的情况，我们将代码 Γ 中的函数称为内部函数，args_alloc 操作负责为实参和函数局部变量分配内存（定义在附录A的图A.3中）；当所调用的函数不在 Γ 中时（ExtCall 规则），这时会产生消息 `call(f,` $\overline{v}$`)`（`f` 是函数标记，$\overline{v}$ 是参数），从全局操作语义 6.3 (a) 中的 ι-Step 规则可知，全局程序会接受该消息，并调用抽象原语。输出指令规则（Print）展示了执行 `print` 指令会产生外部可见事件，而后继规则（Cont）展示了在当前语句为 **skip** 时，我们从后继语句 κ 取出首语句作为当前语句继续执行。执行函数调用等指令时需要涉及到为变量分配内存（例如：args_alloc）和表达式的求值（例如：$[\![\mathsf{e}]\!]_{(\mathrm{E},\Sigma)}$，$[\![\overline{\mathsf{e}}]\!]_{(\mathrm{E},\Sigma)}$），相关操作定义在图A.3和图A.4中，这里不再赘述。

6.3　底层 SPARCv8 程序

$$
\begin{array}{rrcl@{\qquad}rrcl}
\text{(LWorld)} & W & ::= & (C, \sigma, \mathtt{pc}, \mathtt{npc}) & \text{(LState)} & \sigma & ::= & (m, Q, D) \\
\text{(LMem)} & m & ::= & \mathrm{Addr} \rightharpoonup \mathrm{Val} & \text{(Rstate)} & Q & ::= & (R, F) \\
\text{(Code)} & C & \in & \mathrm{Addr} \rightharpoonup \mathrm{Comm} & \text{(RegFile)} & R & \in & \mathrm{RegName} \to \mathrm{Val} \\
\text{(FrameList)} & F & ::= & \mathrm{nil} \mid \mathrm{fm} :: F & \text{(Frame)} & \mathrm{fm} & ::= & [v_0, \cdots, v_{31}] \\
\text{(DBuf)} & D & ::= & \mathrm{nil} \mid (tc, \mathtt{sr}, v) & \text{(DlyCycle)} & tc & \in & \{0, 1, \cdots, X\} \\
\text{(Val)} & v & ::= & n \mid (b, ofs) \mid \mathtt{null} \mid \mathtt{undef} & & & & \\
\text{(Comm)} & c & ::= & \mathtt{call\ f} \mid \mathtt{jmp\ a} \mid \mathtt{retl} \mid \mathtt{print} \mid \mathtt{i} & & & & \\
\text{(LMsg)} & \beta & ::= & \tau \mid e & \text{(Event)} & e & ::= & \mathtt{out}(v)
\end{array}
$$

图 6.4　底层 SPARCv8 机器模型

我们的底层程序选择串行的 SPARCv8 机器模型。值得注意的是，在第6.2节介绍高层机器模型时，我们定义抽象线程池 T 作为表示多线程共同执行，不过在实际的机器模型中并不存在抽象线程池，所以这里我们选择串行的 SPARCv8 机器模型作为我们的底层机器模型。

我们在图6.4中定义了 SPARCv8 的机器模型，该机器模型基本上和第3.1节

$$\frac{(R,F) \rightrightarrows (R',F') \quad C \vdash ((m,(R',F),D'),\mathtt{pc},\mathtt{npc}) \overset{\beta}{\longmapsto} ((m',(R'',F'),D''),\mathtt{pc}',\mathtt{npc}')}{(C,(m,(R,F),D),\mathtt{pc},\mathtt{npc}) :\overset{\beta}{\Longrightarrow} (C,(m',(R'',F'),D''),\mathtt{pc}',\mathtt{npc}')}$$

$$\frac{(R,F) \rightrightarrows (R',F') \quad C \vdash ((m,(R',F),D'),\mathtt{pc},\mathtt{npc}) \overset{\tau}{\longmapsto} \mathbf{abort}}{(C,(m,(R,F),D),\mathtt{pc},\mathtt{npc}) :\overset{\tau}{\Longrightarrow} \mathbf{abort}}$$

(a) 程序转移规则

$$\frac{C(\mathtt{pc}) = \mathtt{i} \quad (\mathtt{i},(m,Q,D)) \bullet\!\!\longrightarrow (m',Q',D')}{C \vdash ((m,Q,D),\mathtt{pc},\mathtt{npc}) \overset{\tau}{\longmapsto} ((m',Q',D'),\mathtt{npc},\mathtt{npc}{+}4)}$$

$$\frac{C(\mathtt{pc}) = \mathtt{i} \quad (\mathtt{i},(m,Q,D)) \bullet\!\!\longrightarrow \mathbf{abort}}{C \vdash ((m,Q,D),\mathtt{pc},\mathtt{npc}) \overset{\tau}{\longmapsto} \mathbf{abort}}$$

$$\frac{C(\mathtt{pc}) = \mathtt{print} \quad R(\%\mathtt{o}_0) = v}{C \vdash ((m,Q,D),\mathtt{pc},\mathtt{npc}) \overset{\mathtt{out}(v)}{\longmapsto} ((m,Q,D),\mathtt{pc}{+}8,\mathtt{pc}{+}12)}$$

$$\frac{C(\mathtt{pc}) = \mathtt{call\ f}}{C \vdash ((m,(R,F),D),\mathtt{pc},\mathtt{npc}) \overset{\tau}{\longmapsto} ((m,(R\{\mathtt{r}_{15} \rightsquigarrow \mathtt{pc}\},F),D),\mathtt{npc},(\mathtt{f},0))}$$

(b) 控制流转移指令转移规则

图 6.5　程序语义及部分控制转移指令语义

图3.2 中形式化定义的 SPARCv8 语法和程序状态类似。所以，这里我们省去了一些与图3.2中相同的部分，例如：寄存器名（RegName），简单指令（Simplins），以及程序指针 pc 和 npc 等。寄存器文件（RegFile）此处定义为全部映射。

底层机器 W 定义为一个四元组，包括：代码堆 C、程序状态 σ、以及程序指针 pc 和 npc。程序状态 σ 是一个包含内存 m，寄存器状态 Q 和延时缓存 D 的三元组。这里的内存是地址到值的映射，与图3.2 中内存定义的不同之处在于，我们没有将地址和值都视为字（Word），而是为了方便证明采用了和高层机器模型中相同的内存定义（定义在图6.2中）。对于指令 c（Comm），这里我们在原基础上扩充了 print 指令，用于产生外部可见事件 e。真实的底层程序通过调用系统函数 print 产生输出，这里我们相当于用 print 指令代替了该系统调用。

操作语义 我们在图6.5中展示了底层 SPARCv8 程序的程序状态转移和部分控制流转移指令，延时写入状态转移（_ ⇉ _）和第3.2节中的定义相同，此处略去。而简单指令的转移规则，这里我们用 “$(\mathtt{i},\_) \bullet\!\!\longrightarrow \_$” 表示，也和第3.2节中所定义的状态转移指令类似，只不过这里因为我们将箭头上的内容视为输出，所以没有将指令 i 放在箭头 “$\bullet\!\!\longrightarrow$” 上（简单指令转移规则定义在附录B的图B.1中）。和之前的 SPARCv8 语义相比，我们在程序状态转移（定义在图6.5中）里增加了程

序中止 **abort** 的情况为了和高层程序对应，另外程序执行的每一步可能产生输出（图6.4中 msg 定义），输出可能为空（用 τ 表示），也可能是一个外部可见事件 e。

图6.5 (b) 中定义了部分控制流转移指令转移规则。如果当前指令为简单指令 i，要么执行简单指令转移规则（$(\mathtt{i}, \_) \bullet\!\!\longrightarrow \_$），要么当前指令的执行导致程序中止（**abort**），我们将指令导致程序中止定义为：

$$(\mathtt{i}, \sigma) \bullet\!\!\longrightarrow \mathbf{abort} \stackrel{\text{def}}{=} \neg\exists\, \sigma'.\, (\mathtt{i}, \sigma) \bullet\!\!\longrightarrow \sigma'$$

如果当前指令是 print 指令，则会产生一个外部可见事件 "$\mathtt{out}(v)$"。我们之前介绍了我们用 print 指令代替调用系统函数 print，而 SPARCv8 使用 "$\%o_0 \sim \%o_5$" 寄存器传递参数，所以这里 $\%o_0$ 寄存器中保存着待输出的值，而系统调用执行结束之后返回 pc+8 继续执行。这里我们没有执行具体的系统函数 print，而是发生函数调用后立即返回。call 指令的执行和之前的介绍没有区别，此处同样不做赘述。

6.4　最终目标

我们将在这一节形式化地刻画我们的最终目标，即汇编原语实现和抽象汇编原语之间存在上下文精化关系，我们首先给出底层和高层程序之间精化关系的形式化定义。

精化关系 $\sqsubseteq$ 是描述两个程序之间程序行为的关系，即：程序 A 精化程序 B 表示程序 A 不会产生比程序 B 更多的行为。这里，我们用事件路径 $\mathcal{B}$（event trace）来定义程序的行为[29]，事件路径被定义为由外部可见事件 e 组成的序列，当然也可以是空序列 nil，或者程序出错 **abort**。

$$(\mathit{EvtTrace})\ \ \mathcal{B} \stackrel{\text{def}}{=} \mathbf{abort} \mid \text{nil} \mid e :: \mathcal{B}$$

定义 6.1 (事件路径精化)

$$W \sqsubseteq \mathbb{W} \stackrel{\text{def}}{=} \forall\, \mathcal{B}.\, \mathit{Etr}(W, \mathcal{B}) \Longrightarrow \mathit{Etr}(\mathbb{W}, \mathcal{B})$$

$$\frac{W :\stackrel{\tau}{\Longrightarrow} \mathbf{abort}}{\mathit{Etr}(W, \mathbf{abort})} \qquad \frac{}{\mathit{Etr}(W, \text{nil})}$$

$$\frac{W :\stackrel{e}{\Longrightarrow} W' \quad \mathit{Etr}(W', \mathcal{B})}{\mathit{Etr}(W, e :: \mathcal{B})} \qquad \frac{W :\stackrel{\tau}{\Longrightarrow} W' \quad \mathit{Etr}(W', \mathcal{B})}{\mathit{Etr}(W, \mathcal{B})}$$

定义6.1中形式化地定义了事件路径精化关系，$W \sqsubseteq \mathbb{W}$ 表示：如果机器 W 执行有限步产生的事件路径 $\mathcal{B}$（有限步因为 Etr 是归纳定义的），那么机器 $\mathbb{W}$ 也可以执行产生。我们基于此定义，来定义底层 SPARCv8 程序和高层带抽象汇编原语的 C 程序之间的精化关系。

定义 6.2 (底层和高层程序之间的精化关系)

$$C \sqsubseteq_{\varphi,\mathbb{S}} \mathbb{P} \stackrel{\text{def}}{=} \forall \sigma_T, \sigma_s, \Sigma.\ \mathsf{initL}(\sigma_T, \{\mathtt{f}_1, \cdots, \mathtt{f}_n\}) \wedge \mathsf{wfmap}(\varphi, \mathbb{S}, \Sigma, \sigma_s, n) \Longrightarrow$$
$$\exists \theta, i \in \{1, \cdots, n\}.\ (\mathbb{P}, \Sigma) \stackrel{load}{\Longrightarrow} ((\Gamma, \Omega), \theta)$$
$$\wedge (C, \sigma_T \uplus \sigma_s, (\mathtt{f}_i, 0), (\mathtt{f}_i, 4)) \sqsubseteq ((\Gamma, \Omega), \theta)$$

where $\mathbb{P} = \textbf{let}\ (\Gamma, \Omega)\ \textbf{in}\ \mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n$,

$$\text{and}\quad \sigma_1 \uplus \sigma_2 \stackrel{\text{def}}{=} \begin{cases} (m_1 \uplus m_2, Q, D) & \textbf{if}\ \sigma_1 = (m_1, Q, D), \\ & \sigma_2 = (m_2, Q, D),\ m_1 \perp m_2 \\ \textbf{undefined} & \textbf{otherwise} \end{cases}$$

定义6.2形式化地定义了底层和高层程序之间的精化关系，我们定义 initL（参见定义6.3）来描述初始情况下底层程序每个线程的私有内存，定义 wfmap（参见定义6.3）来刻画底层和高层程序线程的共享内存之间的对应关系。定义6.2是说对于任意的底层和高层程序状态，如果底层程序状态可以分为不相交的两部分：包含线程私有内存空间的程序状态 σ_T 和包含所有线程的共享内存的程序状态 σ_s，并且 σ_T 满足一定的初始化约束，而 σ_s 和高层程序状态 Σ 存在一定的对应关系，则底层和高层机器之间存在事件路径精化关系。其中，$\mathbb{S}$ 是高层程序共享内存的地址集合，而 φ 是高层和底层程序内存之间内存地址上的一个映射关系：$\varphi \in \text{Addr} \rightharpoonup \text{Addr}$。定义6.3 给出了 initL 和 wfmap 的形式化定义。该定义中用到的描述程序初始状态的辅助定义参见图6.6。

定义 6.3

$$\mathsf{initL}(\sigma_T, \{\mathtt{f}_1, \cdots, \mathtt{f}_n\}) \stackrel{\text{def}}{=} \exists\ m_1, \cdots, m_n, i \in \{1, \cdots, n\}.$$
$$m_1 \uplus \cdots \uplus m_n \uplus \{\mathsf{TaskCur} \rightsquigarrow (i, 0)\} \subseteq m$$
$$\wedge\, \mathsf{initCurT}(i, m_i, \mathtt{f}_i, Q) \wedge D = \text{nil}$$
$$\wedge (\forall\ j \in \{1, \cdots, n\},\ i \neq j.\ \mathsf{initRdy}(j, m_j, \mathtt{f}_j))$$
$$\text{where}\ \ \sigma = (m, Q, D)$$

$$\mathsf{wfmap}(\varphi, \mathbb{S}, \Sigma, \sigma_s, n) \stackrel{\text{def}}{=} \mathsf{initM}(\varphi, \mathbb{S}, \Sigma, \sigma_s) \wedge [\varphi]_{\{1,\cdots,n\}}$$

定义 initL 要求：（1）初始状态下的 n 个线程，每个线程在底层程序内存中拥有自己的私有内存，依次记为：m_1 至 m_n，地址为 TaskCur 的内存单元保存的当前线程的线程编号 i；（2）我们定义 initCurT 来描述初始状态当前线程 i 的私有内存 m_i 和初始状态下的寄存器状态 Q，初始状态下 Q 中的栈指针 %sp 指向当前线程 i 的初始栈空间的首地址（初始栈空间属于该线程的私有内存），而当前窗口指针 cwp 指向无效窗口的下一个，而帧链表 $Q.F$ 的内容我们不关心；（3）

$$\mathsf{initRdy}(t, m, \mathtt{f}) \stackrel{\text{def}}{=} \exists\, b.\ \mathsf{initT}(t, \mathtt{f}, b) = m$$

$$\mathsf{initCurT}(t, m, \mathtt{f}, Q) \stackrel{\text{def}}{=} \exists\, b.\ \mathsf{initT}(t, \mathtt{f}, b) = m \wedge \mathsf{initR}(Q.R, b)$$

$$\begin{aligned}\mathsf{initT}(t, \mathtt{f}, b) \stackrel{\text{def}}{=}\ & \{[(t, ofs_g), \cdots, (t, ofs_g{+}7)] \rightsquigarrow \mathtt{undef}\} \uplus \{[(b,0), \cdots, (b,15)] \rightsquigarrow \mathtt{undef}\} \\ & \uplus \{(t, ofs_Y) \rightsquigarrow \mathtt{undef},\ (t, ofs_{ret}) \rightsquigarrow (\mathtt{f}, -8),\ (t, ofs_{sp}) \rightsquigarrow (b, 0)\}\end{aligned}$$

$$\begin{aligned}\mathsf{initR}(R, b) \stackrel{\text{def}}{=}\ & \exists\, n.\ R(\mathtt{cwp}) = \mathbf{next\_cwp}(n) \wedge R(\mathtt{wim}) = 2^n \wedge 0 \leqslant n < N \\ & \wedge\, \mathtt{r}_{15} = \mathtt{undef} \wedge R(\%\mathtt{sp}) = (b, 0)\end{aligned}$$

$$\begin{aligned}\mathsf{initM}(\varphi, \mathbb{S}, \Sigma, m) \stackrel{\text{def}}{=}\ & \mathsf{range}(\mathrm{G}) \subseteq \mathbb{S} \subseteq \mathrm{dom}(M) \wedge \varphi\{\{\mathbb{S}\}\} \subseteq \mathrm{dom}(m) \\ & \wedge\, \mathsf{Inv}(\varphi|_{\mathbb{S}}, M, m) \wedge \mathsf{closed}(\mathbb{S}, M) \qquad \text{where } \Sigma = (\mathrm{G}, M)\end{aligned}$$

$$[\varphi]_{\{t_1,\cdots,t_n\}} \stackrel{\text{def}}{=} \forall\, t \in \{t_1, \cdots, t_n\},\ ofs.\ (t, ofs) \in \mathrm{dom}(\varphi) \Longrightarrow \varphi(t, ofs) = (t, ofs)$$

$$\mathsf{range}(\mathrm{E}) \stackrel{\text{def}}{=} \{l \mid \exists x,\ l,\ ty.\ \mathrm{E}(x) = (l, ty)\} \qquad \varphi|_{\mathbb{S}} \stackrel{\text{def}}{=} \{(l, \varphi(l)) \mid l \in (\mathbb{S} \cap \mathrm{dom}(\varphi))\}$$

$$v_1 \overset{\varphi}{\hookrightarrow} v_2 \stackrel{\text{def}}{=} (v_1 \neq \mathrm{Addr} \wedge v_1 = v_2) \vee (v_1, v_2 \in \mathrm{Addr} \wedge \varphi(v_1) = v_2)$$

$$\mathsf{Inv}(\varphi, M, m) \stackrel{\text{def}}{=} \forall\, l, l'.\ (l \in \mathrm{dom}(M) \wedge \varphi(l) = l') \Longrightarrow (l' \in \mathrm{dom}(m) \wedge M(l) \overset{\varphi}{\hookrightarrow} m(l'))$$

$$\mathsf{closed}(\mathbb{S}, M) \stackrel{\text{def}}{=} \forall\, l,\ l'.\ (l \in \mathbb{S} \wedge l' = M(l)) \Longrightarrow l' \in \mathbb{S}$$

图 6.6　程序状态初始化相关定义

初始状态下，延时缓存 D 为空；（4）而对于非当前线程 j，其私有内存 m_j 中保存了线程 j 起始状态下的运行时上下文，其中用于保存栈指针的内存单元（偏移地址为 ofs_{sp}）保存了起始栈空间的首地址，而保存返回地址的内存单元（偏移地址为 ofs_{ret}）的初始值为 $(\mathtt{f}_j, -8)$，因为 `retl` 指令返回 $\mathtt{r}_{15}{+}8$ 的程序点，所以该内存单元中的值被赋予 $\mathtt{r}_{15}$ 寄存器后，执行 `retl` 指令程序将跳转到入口函数 $\mathtt{f}_j$ 的起始点开始执行，我们定义 initRdy 来描述线程 j 的初始私有内存 m_j。定义 wfmap 对底层和高层程序状态和映射关系 φ 之间做了约束，我们定义 initM 表明初始状态下底层和高层程序的共享内存之间的对应关系，$\mathbb{S}$ 代表高层程序线程共享内存的地址集合。此外，我们这里因为直接将高层线程 t 对应与底层程序起始地址为 $(t, 0)$ 的任务控制块，所以 $[\varphi]_{\{1,\cdots,n\}}$ 表示将高层内存中的任务控制块直接映射到底层程序相同的地址空间。

定理6.1形式化地刻画了我们的最终目标，它描述了汇编原语实现和抽象汇编原语之间的上下文精化关系，即调用汇编原语实现 C_{as} 的底层程序和调用 Ω 中抽象汇编原语的高层程序之间存在精化关系。抽象原语集合 Ω 定义在第6.2节图6.2中，目前只包含一个抽象汇编原语 `switch`。代码堆 C_{as} 中是抽象汇编原语 `switch` 的 SPARCv8 实现，由于空间原因我们将它定义在附录D的图D.1中。

定理 6.1 (最终目标)　对于任意的 C，Γ，$\mathtt{f}_1$，…，$\mathtt{f}_n$，$\mathbb{S}$ 和 φ，若下述成立：

1. $C \preccurlyeq_{\varphi,\mathbb{S}} \Gamma$；
2. Safe(**let** (Γ, Ω) **in** $\mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n, \mathbb{S}$)；
3. ReachClose($\Gamma, \mathbb{S}$)；

则 $C \uplus C_{\text{as}} \sqsubseteq_{\varphi,\mathbb{S}}$ **let** (Γ, Ω) **in** $\mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n$ 成立。

定理6.1中前提 1 描述的是调用汇编原语实现的底层客户端程序 C 和调用抽象汇编原语的高层客户端程序 Γ 之间的关系。C 是由高层代码 Γ 编译生成的，我们希望编译器可以建立源程序和目标程序之间的模拟关系 $C \preccurlyeq_{\varphi,\mathbb{S}} \Gamma$。关于该模拟关系的详细介绍和形式化定义参见附录C中第C.3节。

前提 2 要求高层程序是安全 Safe 的（见定义C.1），这样可以排除高层程序非法调用抽象原语的情况，也就是说我们不保证不安全地调用抽象原语的源程序与编译后的调用抽象原语实现的目标程序之间存在精化关系。前提 3 中 ReachClose（见定义C.2）保证了代码 Γ 中的函数执行不会访问自己局部和共享的内存以外的内存空间，即：不会访问其它方法的私有内存。Safe 和 ReachClose 的定义参考了并发 C 程序编译器的验证工作[33]，详细介绍参见附录第C.1节。

6.5　汇编实现和抽象汇编原语之间的模拟关系

我们在本章的第6.1节中介绍到我们通过定义汇编原语实现和其所对应的抽象汇编原语之间的模拟关系来证明他们之间的上下文精化关系。所以，我们将在这一节介绍该模拟关系。

底层和高层程序状态的对应关系　定义模拟关系的关键是定义底层和高层程序执行过程中程序状态的对应关系。图6.7中定义了底层程序调用汇编实现时的程序状态 σ 和高层程序调用汇编原语时的程序状态 θ 之间的对应关系 "$\sigma \sim_{(\mu,\overline{v})} \theta$"。其中，$\overline{v}$ 是调用汇编原语时的参数，而 μ 是一个三元组：

$$\mu \stackrel{\text{def}}{=} (\varphi, \mathbb{S}, d) \qquad \text{where } \mathbb{S}, d \in \mathcal{P}(\text{Addr})$$

映射关系 φ 与地址集合 $\mathbb{S}$ 和之前介绍的一样，分别代表高层和底层程序内存地址之间的映射关系和高层程序共享内存的地址集合，而地址集合 d 表示当前底层汇编原语实现方法的局部内存的地址集合，初始状态时局部内存包括：当前线程 TCB 中用于保存线程运行时上下文的内存和调用者暴露给该方法的内存。由 "$\sigma \sim_{(\mu,\overline{v})} \theta$" 的定义可知：底层程序的内存状态 m 可以分为四部分：保存线程运行时上下文的内存 m_T，当前方法调用者暴露给该方法的内存 m_l，所有线程的共享内存 m_s，以及保存当前线程编号 t 的地址为 TaskCur 的内存单元。图6.7中定义 $T \sim_t m_T$ 刻画了线程池和底层内存之间的对应关系，可以理解为 m_T 中包括

$$\Delta ::= ((\iota, \overline{v}), \theta) \mid (\bot, \theta) \qquad \frac{\theta \,\phi\xrightarrow{(\iota,\overline{v})}\, \theta'}{((\iota, \overline{v}), \theta) \Longmapsto (\bot, \theta')}$$

$$\frac{m_T = m_1 \uplus m_2 \quad \mathrm{dom}(m_1) = \mathsf{DomCtx}(t) \quad t \in \mathrm{dom}(T) \quad T\backslash\{t\} \sim m_2}{T \sim_t m_T}$$

$$\frac{T_1 \sim m_1 \quad T_2 \sim m_2}{T_1 \uplus T_2 \sim m_1 \uplus m_2} \qquad \frac{m \rightarrowtail_t Q}{\{t \twoheadrightarrow \mathcal{K}\} \sim m}$$

$$\mathsf{DomCtx}(t) ::= \{(t, ofs_g), \cdots, (t, ofs_g + 7), (t, ofs_Y), (t, ofs_{ret}), (t, ofs_{sp})\}$$

$$\frac{\begin{array}{l}\sigma = (m, (R, F), \mathrm{nil}) \quad \theta = (T, t, \mathcal{K}, \Sigma) \\ m_T \uplus m_l \uplus m_s \uplus \{\mathsf{TaskCur} \rightsquigarrow (t, 0)\} \subseteq m \\ T \sim_t m_T \quad \mathsf{caller\_explore}(R, F, \overline{v}, m_l) \quad \mathsf{initM}(\varphi|_{\mathbb{S}}, \mathbb{S}, \Sigma, m_s) \\ \mu = (\varphi|_{\mathbb{S}}, \mathbb{S}, \mathrm{dom}(m_l) \cup \mathsf{DomCtx}(t)) \quad [\varphi]_{\mathrm{dom}(T)}\end{array}}{\sigma \sim_{(\mu,\overline{v})} \theta}$$

图 6.7　辅助状态及底层和高层程序状态的对应关系

了当前线程 t 任务控制块 TCB 中用于保存线程运行时上下文的内存（用于保存 global，%sp 和 $\mathtt{r}_{15}$ 寄存器），以及其他线程在被调度时需要被恢复的上下文所保存在的内存空间，并且保存每个线程运行时上下文的内存空间不相交。函数 DomCtx(t) 返回线程 t 任务控制块 TCB 中保存运行时上下文的内存的地址集合，我们定义 $m \rightarrowtail_t Q$ [①]表示可以从内存 m 中恢复线程 t 运行时的寄存器状态 Q。我们定义 caller_explore 来描述当前方法的调用者暴露给当前方法的内存，调用者暴露给当前方法的内存包括：用于传递参数的内存和用于保存其它寄存器窗口的内存（caller_explore 的形式化定义和详细说明见附录第C.2节）。

模拟关系　我们在这一节定义汇编原语实现和汇编原语之间的模拟关系，我们首先在定义6.4中定义汇编原语实现和汇编原语集合之间的模拟关系；之后，再在定义6.5和定义6.6中给出单个汇编原语实现和其对应汇编原语之间的模拟关系。

定义 6.4 (实现与原语集合的模拟)

$$C_{\mathrm{as}} \leqslant_{\varphi,\mathbb{S}} \Omega \stackrel{\mathrm{def}}{=} \forall\, \mathtt{f}, \iota, \overline{ty}.\ \Omega(\mathtt{f}) = (\iota, \overline{ty}) \Longrightarrow (C_{\mathrm{as}}, \mathtt{f}) \leqslant_{\varphi,\mathbb{S}} (\iota, \overline{ty})$$

定义 6.5 (实现与原语的模拟)

$$\begin{aligned}(C_{\mathrm{as}}, \mathtt{f}) \leqslant_{\varphi,\mathbb{S}} (\iota, \overline{ty}) \stackrel{\mathrm{def}}{=}\ & \forall\, \sigma, \theta, d, \overline{v}.\ (\sigma \sim_{(\mu,\overline{v})} \theta \wedge \mathsf{tyms}(\overline{v}, \overline{ty})) \Longrightarrow \\ & (\theta \,\phi\xrightarrow{(\iota,\overline{v})}\, \mathbf{abort} \vee (C_{\mathrm{as}}, \sigma, (\mathtt{f}, 0), (\mathtt{f}, 4)) \leqslant^0_{(\mu,\sigma)} ((\iota, \overline{v}), \theta)) \\ & \text{where}\ \ \mu = (\varphi|_{\mathbb{S}}, \mathbb{S}, d)\end{aligned}$$

[①] $m \rightarrowtail_t Q$ 的定义参见附件 C的图C.3

定义6.4中定义了汇编实现精化汇编原语集合 $C_{\text{as}} \leqslant_{\varphi,\mathbb{S}} \Omega$，它是说，对于任何一个函数标记 f，如果我们汇编原语集合中存在其对应的汇编原语 ι（$\Omega(\mathtt{f}) = (\iota, \overline{ty})$），那么我们有 $(C_{\text{as}}, \mathtt{f}) \leqslant_{\varphi,\mathbb{S}} (\iota, \overline{ty})$，其中 f 也是汇编原语 ι 的实现函数存储在 C_{as} 中代码块的块号。定义 6.5 给出了汇编实现精化汇编原语的定义 $(C_{\text{as}}, \mathtt{f}) \leqslant_{\varphi,\mathbb{S}} (\iota, \overline{ty})$，该定义是说对于任意的底层程序状态 σ，高层程序状态 θ，地址集合 d 和参数 $\overline{v}$，如果底层和高层程序状态之间满足 $\sigma \sim_{(\mu,\overline{v})} \theta$（这里 $\mu = (\varphi|_{\mathbb{S}}, \mathbb{S}, d)$），并且参数 $\overline{v}$ 和类型 $\overline{ty}$ 是匹配的（$\mathsf{tyms}(\overline{v}, \overline{ty})$，定义在图A.4中）。那么要么高层程序调用抽象汇编原语会中止（**abort**），或者抽象原语实现和抽象原语之间存在定义6.6中模拟关系。该定义也说明我们不保证高层程序在不合法调用抽象汇编原语的情况下，抽象汇编原语和其具体实现之间存在模拟关系。

我们在图6.7中定义了辅助状态 Δ，用于定义抽象原语实现和抽象原语之间的模拟关系（定义6.6），Δ 是一个二元组包括高层程序状态 θ，以及抽象汇编原语 ι 和参数 $\overline{v}$ 构成的二元组，或者 $\bot$（表示抽象原语已经执行）。定义 6.6 给出了汇编原语实现和汇编原语之间的模拟关系，该定义中用到的辅助定义见图6.8。

定义 6.6　$(C, \sigma, \mathtt{pc}, \mathtt{npc}) \preccurlyeq^{k}_{(\mu,\sigma_0)} \Delta$ 成立，当且仅当，有 $\mathtt{pc} \in \mathrm{dom}(C)$ 成立，并且下述全部成立：

1. 若 $C(\mathtt{pc}) = \{\mathtt{i}, \mathtt{jmp\ a}, \mathtt{be\ f}\}$，则
   - 存在 σ'，$\mathtt{pc}'$，$\mathtt{npc}'$，使得 $(C, \sigma, \mathtt{pc}, \mathtt{npc}) :\overset{\tau}{\Longrightarrow} (C, \sigma', \mathtt{pc}', \mathtt{npc}')$
   - 任意的 $\sigma', \mathtt{pc}', \mathtt{npc}'$，如果 $(C, \sigma, \mathtt{pc}, \mathtt{npc}) :\overset{\tau}{\Longrightarrow} (C, \sigma', \mathtt{pc}', \mathtt{npc}')$，则 $(\sigma, \sigma') \vDash G_\mu$，$(C, \sigma, \mathtt{pc}, \mathtt{npc}) \preccurlyeq^{k}_{(\mu,\sigma_0)} \Delta$ 成立；
2. 若 $C(\mathtt{pc}) = \mathtt{call\ f}$，则
   - 存在 σ'，$\mathtt{pc}'$，$\mathtt{npc}'$，使得 $(C, \sigma, \mathtt{pc}, \mathtt{npc}) :\overset{\tau}{\Longrightarrow}{}^{2} (C, \sigma', \mathtt{pc}', \mathtt{npc}')$
   - 任意的 σ'，$\mathtt{pc}'$，$\mathtt{npc}'$，如果 $(C, \sigma, \mathtt{pc}, \mathtt{npc}) :\overset{\tau}{\Longrightarrow}{}^{2} (C, \sigma', \mathtt{pc}', \mathtt{npc}')$，则 $(\sigma, \sigma') \vDash G_\mu$，$(C, \sigma, \mathtt{pc}, \mathtt{npc}) \preccurlyeq^{k+1}_{(\mu,\sigma_0)} \Delta$ 成立；
3. 若 $C(\mathtt{pc}) = \mathtt{retl}$，则
   - 存在 σ'，$\mathtt{pc}'$，$\mathtt{npc}'$，使得 $(C, \sigma, \mathtt{pc}, \mathtt{npc}) :\overset{\tau}{\Longrightarrow}{}^{2} (C, \sigma', \mathtt{pc}', \mathtt{npc}')$
   - 任意的 σ'，$\mathtt{pc}'$，$\mathtt{npc}'$，
     如果 $(C, \sigma, \mathtt{pc}, \mathtt{npc}) :\overset{\tau}{\Longrightarrow}{}^{2} (C, \sigma', \mathtt{pc}', \mathtt{npc}')$，则 $(\sigma, \sigma') \vDash G_\mu$，并且
     若 $k = 0$，则存在 Δ'，使得：
     $$\Delta \Longmapsto \Delta' \text{ 和 } \mathbf{wfctx}(\mu, (\sigma_0, \Delta), (\sigma', \Delta'), \mathtt{pc}', \mathtt{npc}') \text{ 成立；}$$
     或者　$(C, \sigma', \mathtt{pc}', \mathtt{npc}') \preccurlyeq^{k-1}_{(\mu,\sigma_0)} \Delta$。

定义 6.6 中，k 表示函数调用的层数；μ 是一个三元组 $(\varphi, \mathbb{S}, d)$, 分别表示底层和高层程序共享内存的地址映射 φ，高层程序的共享内存 $\mathbb{S}$，以及当前底层程

$$(\sigma, \sigma') \vDash G_\mu \overset{\text{def}}{=} \sigma.m \overset{\text{dom}(\sigma.m)-d'}{=\!=\!=} \sigma'.m$$
$$\text{where } \mu = (\varphi, \mathbb{S}, d),\ d' = \varphi\{\{\mathbb{S}\}\} \cup d \cup \{\mathsf{TaskCur}\}$$

$$\mathbf{wfctx}(\mu, (\sigma_0, \Delta), (\sigma', \Delta'), \mathtt{pc}', \mathtt{npc}') \overset{\text{def}}{=}$$
$$\Sigma.M \overset{\text{dom}(\Sigma.M)-\mathbb{S}}{=\!=\!=} \Sigma'.M \wedge \mathsf{Inv}(\varphi, \Sigma'.M, \sigma'.m) \wedge \mathsf{closed}(\mathbb{S}, \Sigma'.M)$$
$$\wedge (\mathbf{fret}((\sigma_0, \Delta), (\sigma', \Delta')) \vee \mathbf{ctxsw}(\mu, (\sigma_0, \Delta), (\sigma', \Delta'))) \wedge \mathtt{pc}' = \mathsf{ptr}{+}8 \wedge \mathtt{npc}' = \mathsf{ptr}{+}12$$
$$\text{where } \mu = (\varphi, \mathbb{S}, d),\ \mathsf{ptr} = \sigma'.Q.R(\mathtt{r}_{15})$$

$$\mathbf{ctxsw}((\sigma_0, \Delta), (\sigma', \Delta')) \overset{\text{def}}{=} m' \rightarrowtail_{t'} Q' \wedge Q \rightarrowtail_t m' \wedge m'(\mathsf{TaskCur}) = (t', 0)$$
$$\wedge\, t \neq t' \wedge T' = T\{t \nrightarrow \mathcal{K}\} \wedge T(t') = \mathcal{K}'$$
$$\text{where } \mu = (\varphi, \mathbb{S}, d),\ \sigma_0 = (m, Q, \text{nil}),\ \sigma = (m', Q', \text{nil}),$$
$$\Delta = (\_, (T, t, \mathcal{K}, \Sigma)),\ \Delta' = (\bot, (T', t', \mathcal{K}', \Sigma'))$$

$$\mathbf{fret}((\sigma_0, \Delta), (\sigma', \Delta')) \overset{\text{def}}{=} Q \propto (Q', m') \wedge \Sigma.M(\mathsf{TaskCur}) = t$$
$$\text{where } \sigma_0 = (m, Q, \text{nil}),\ \sigma = (m', Q', \text{nil}),$$
$$\Delta = (\_, (T, t, \mathcal{K}, \Sigma)),\ \Delta' = (\bot, (T, t, \mathcal{K}, \Sigma'))$$

$$Q \rightarrowtail_t m' \overset{\text{def}}{=} (\exists Q'.\, m' \rightarrowtail_t Q') \wedge (\forall Q'.\, (m' \rightarrowtail_t Q') \Longrightarrow (Q \propto (Q', m')))$$

图 6.8　汇编实现和汇编原语间模拟关系中的辅助定义

序的局部内存空间地址集合 d；σ_0 是汇编原语实现函数被调用时的起始状态。

我们通过图6.9来对该模拟关系有个直观的认识，当底层程序调用汇编原语实现时，高层程序调用汇编原语，在底层程序执行汇编原语的实现方法时，高层程序一直不走，直到底层程序中汇编原语实现方法执行结束。另外我们要求底层程序执行的每一步都满足保证 G_μ，即：底层程序每一步只修改自己的局部内存，共享内存以及记录当前线程编号的地址为 TaskCur 的内存单元，G_μ 保证了底层程序当前方法的执行不修改其他线程的*私有内存*（*私有内存*中保存了线程的运行时上下文）。注意，这里虽然不允许修改其它线程的*私有内存*，但允许读这些内存，所以仍然可以从其它线程的*私有内存*中恢复它被保存的运行时上下文。

我们重点来看模拟关系中的第 3 条，如果当前指令为 `retl`，则程序可以走两步，并且如果程序可以走两步并且 $k = 0$，说明当前方法已经执行结束，否则继续有模拟关系成立。当前方法执行结束时，假设底层程序状态为 $(C, \sigma', \mathtt{pc}', \mathtt{npc}')$，那么高层程序可以执行一步从状态 Δ 到状态 Δ'，并且底层和高层程序的状态转移之间满足约束关系 $\mathbf{wfctx}(\mu, (\sigma_0, \Delta), (\sigma', \Delta'), \mathtt{pc}', \mathtt{npc}')$，这里 σ_0 是调用汇编原语实现时的起始状态。**wfctx** 要求的抽象原语的执行只会修改高层程序的共享内存以及修改之后高层程序的共享内存还是 “封闭的”（closed, 定义在图6.6中），并且底层实现执行结束和高层抽象原语执行结束后共享内存之间仍然存在关系 Inv（定义在图 6.6 中），另外针对该抽象汇编原语是否会导致线程的切换，我们

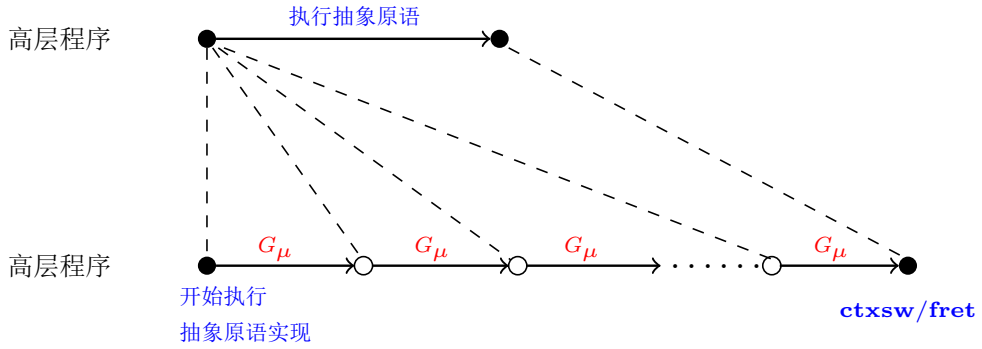

图 6.9　汇编实现和汇编原语之间的模拟关系

分两种情况讨论：（1）如果没有发生线程切换，则底层和高层的状态转移之间满足 **fret**，**fret** 要求汇编原语实现函数执行前后寄存器状态满足 SPARCv8 中函数调用惯例对函数调用前后寄存器状态的要求 $Q \propto (Q', m')$ [①]，并且当前线程的线程号不变；（2）如果发生了线程切换（当前线程由 t 变为 t'），则底层和高层程序状态转移之间满足 **ctxsw**，在 **ctxsw** 中，我们使用 $m' \mapsto_{t'} Q'$（定义在图C.3中）表示从内存 m' 中恢复线程 t' 的运行时上下文 Q'，而 $Q \mapsto_t m'$ 表示线程 t 的运行时上下文 Q 被保存在了内存 m' 中。通常情况下我们不会保存所有寄存器的内容到内存中，但函数调用惯例中要求函数调用前后保持不变的寄存器必须被保存，所以，如果保存运行时上下文时的寄存器状态为 Q，而恢复该线程运行时上下文后的寄存器状态为 Q'，内存状态为 m'，那么二者之间应该有 $Q \propto (Q', m')$。

`switch` 原语实现和原语之间满足模拟关系 我们在定义6.6中定义了汇编原语实现和汇编原语之间的模拟关系，我们需要证明 C_{as} 中 `switch` 原语的 SPARCv8 实现和 `switch` 原语之间存在该模拟关系。

引理 6.2　$(C_{\text{as}}, \mathtt{f}_{\text{sw}}) \leqslant_{\varphi,\mathbb{S}} (\mathtt{switch}, \text{nil})$

引理6.2的正确性保证了 `switch` 原语的 SPARCv8 实现与 `switch` 原语之间存在我们定义的原语实现和原语之间的精化关系。具体的证明过程参见附录D。

实现和原语之间的模拟关系保证上下文精化关系　我们首先定义完整的底层和高层程序之间的全局模拟关系，全局模拟关系用于证明底层和高层程序之间的精化关系。而汇编原语实现和汇编原语之间的模拟关系具有可组合性，它可以和底层和高层客户端程序之间的模拟关系组合起来，得到全局模拟关系。

[①] 图C.3中给出了 $Q \propto (Q', m')$ 的形式化定义，更多的有关 SPARCv8 中的函数调用惯例和函数前用前后寄存器状态的关系，我们在附录第C.2节做了详细的介绍并给出了形式化的定义。

定义 6.7 (全局模拟关系)

$$C \preccurlyeq_{\varphi,\mathbb{S}} \mathbb{P} \stackrel{\text{def}}{=} \forall \sigma_T, \sigma_s, \Sigma.\ \mathsf{initL}(\sigma_T, \{\mathtt{f}_1, \cdots, \mathtt{f}_n\}) \wedge \mathsf{wfmap}(\varphi, \mathbb{S}, \Sigma, \sigma_s, n) \Longrightarrow \exists \theta, i \in \{1, \cdots, n\}.\ (\mathbb{P}, \Sigma) \overset{load}{\Longrightarrow} ((\Gamma, \Omega), \theta) \wedge (C, \sigma_T \uplus \sigma_s, (\mathtt{f}_i, 0), (\mathtt{f}_i, 4)) \preccurlyeq ((\Gamma, \Omega), \theta)$$

where $\mathbb{P} = \textbf{let}\ (\Gamma, \Omega)\ \textbf{in}\ \mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n$

定义 6.8　若 $W \preccurlyeq \mathbb{W}$ 成立，则下述全部成立：

1. 若 $W :\overset{\tau}{\Longrightarrow} W'$，则存在 $\mathbb{W}'$，使得 $\mathbb{W} \overset{\tau}{\Longrightarrow}^* \mathbb{W}'$ 和 $W' \preccurlyeq \mathbb{W}'$ 成立；
2. 若 $W :\overset{e}{\Longrightarrow} W'$，则存在 $\mathbb{W}', \mathbb{W}''$ 使得 $\mathbb{W} \overset{\tau}{\Longrightarrow}^* \mathbb{W}'$，$\mathbb{W}' \overset{e}{\Longrightarrow} \mathbb{W}''$ 和 $W' \preccurlyeq \mathbb{W}''$ 成立；
3. 若 $W :\overset{\tau}{\Longrightarrow} \textbf{abort}$，则 $\mathbb{W} \overset{\tau}{\Longrightarrow}^* \textbf{abort}$ 成立。

我们在定义6.7和6.8中给出了全局模拟关系，该模拟关系描述了完整的底层程序和高层程序之间的关系，该模拟关系保证了如果底层程序可以走一步而且不产生外部可见事件，那么高层程序也可以走 0 步或多步也不产生外部可见事件，并且二者之间还能继续建立模拟关系；如果底层程序走一步产生了外部可见事件 e，那么高层程序也可以走多步后产生同样的事件 e，并且二者之间还可以继续建立模拟关系；如果底层程序执行中止，那么高层程序也会中止。

引理 6.3 (可组合性)　对于任意的 C，Γ，$\mathtt{f}_1$，…，$\mathtt{f}_n$，$\mathbb{S}$，φ，C_{as}，Ω，若下述成立：

1. $C \preccurlyeq_{\varphi,\mathbb{S}} \Gamma$, $C_{\text{as}} \preccurlyeq_{\varphi,\mathbb{S}} \Omega$;
2. $\mathsf{Safe}(\textbf{let}\ (\Gamma, \Omega)\ \textbf{in}\ \mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n, \mathbb{S})$;
3. $\mathsf{ReachClose}(\Gamma, \mathbb{S})$;

则 $C \uplus C_{\text{as}} \preccurlyeq_{\varphi,\mathbb{S}} \textbf{let}\ (\Gamma, \Omega)\ \textbf{in}\ \mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n$ 成立。

引理 6.3 的正确性给出了汇编原语实现和汇编原语之间的模拟关系与客户端程序之间模拟关系的可组合性。

最终目标的证明　我们首先证明引理6.4成立。

引理 6.4 (全局模拟蕴含精化关系)

$$C \uplus C_{\text{as}} \preccurlyeq_{\varphi,\mathbb{S}} \textbf{let}\ (\Gamma, \Omega)\ \textbf{in}\ \mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n \Longrightarrow C \uplus C_{\text{as}} \sqsubseteq_{\varphi,\mathbb{S}} \textbf{let}\ (\Gamma, \Omega)\ \textbf{in}\ \mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n$$

我们根据引理 6.3 模拟关系的可组合性，引理6.4全局模拟蕴含精化关系，以

及引理6.2中任务上下文切换程序的 SPARCv8 实现和 switch 原语之间满足模拟关系给出最终目标定理6.1的证明。

证明　我们需要证明对于任意的 C，Γ，$\mathtt{f}_1$，…, $\mathtt{f}_n$，$\mathbb{S}$ 和 φ，若下述成立：

1. $C \preccurlyeq_{\varphi,\mathbb{S}} \Gamma$；
2. Safe(**let** (Γ, Ω) **in** $\mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n$)；
3. ReachClose($\Gamma, \mathbb{S}$)；

则下述成立：

$$C \uplus C_{\text{as}} \sqsubseteq_{\varphi,\mathbb{S}} \textbf{let } (\Gamma, \Omega) \textbf{ in } \mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n \tag{g}$$

我们首先对目标 (g) 应用全局模拟蕴含精化关系（引理6.4），

$$C \uplus C_{\text{as}} \preccurlyeq_{\varphi,\mathbb{S}} \textbf{let } (\Gamma, \Omega) \textbf{ in } \mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n \tag{g1}$$

我们对目标 (g1) 应用模拟关系的可组合性（引理6.3），则需证明下述成立：

$$C \preccurlyeq_{\varphi,\mathbb{S}} \Gamma \tag{g1.1}$$

$$C_{\text{as}} \preccurlyeq_{\varphi,\mathbb{S}} \Omega \tag{g1.2}$$

$$\text{Safe}(\textbf{let } (\Gamma, \Omega) \textbf{ in } \mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n) \tag{g1.3}$$

$$\text{ReachClose}(\Gamma, \mathbb{S}) \tag{g1.4}$$

目标 (g1.1) (g1.3) (g1.4) 由已知条件可得，我们需要证明 (g1.2) 成立。我们根据定义 6.4将 (g1.2) 展开，则我们需要证明：对于任意的 $\mathtt{f}$，ι，$\overline{ty}$，如果有 $\Omega(\mathtt{f}) = (\iota, \overline{ty})$，则：$(C_{\text{as}}, \mathtt{f}) \preccurlyeq_{\varphi,\mathbb{S}} (\iota, \overline{ty})$ 成立。

我们对 Ω 中的每条抽象汇编原语分情况讨论，不过目前 Ω 中只有一条 switch 原语（定义在6.1中），所以我们考虑这一种情况即可，即我们需要证明：

$$(C_{\text{as}}, \mathtt{f}_{\text{sw}}) \preccurlyeq_{\varphi,\mathbb{S}} (\mathtt{switch}, \text{nil}) \tag{g2}$$

目标 (g2) 可以通过引理6.2得证，证毕。

□

6.6　本章小结

这一章介绍了我们在证明汇编实现和抽象汇编原语之间“上下文精化关系”上所做的一些工作。我们首先在第6.1节给出了验证汇编实现和抽象汇编原语之间上下文精化关系的技术路线和证明思路。并在第6.2节和6.3节中定义了高层带抽象汇编原语的 C 语言和底层 SPARCv8 程序。在第6.4节给出了精化关系的定义和最终目标。最后，第6.5节介绍了我们定义的汇编实现和抽象汇编原语之间

的模拟关系，该模拟关系具有可组合性，并且可以保证汇编实现和抽象汇编原语之间的上下文精化关系，我们证明了一个 SPARCv8 实现的线程上下文切换函数和 `switch` 原语之间存在该模拟关系。目前这一部分工作还没有在 Coq 辅助定理证明工具中实现，也缺少证明汇编原语实现和汇编原语之间精化关系的验证逻辑，这些留到以后去完善。

第 7 章　本文总结和未来工作

本文针对目前的操作系统验证工作中缺少对操作系统内嵌汇编代码 SPARCv8 程序验证的问题，做了如下工作：

- 为 SPARCv8 汇编程序设计了一套霍尔风格的程序验证逻辑，该验证逻辑能够支持 SPARCv8 汇编程序中函数调用的模块化验证以及 SPARCv8 的三个主要特性：延时跳转、特殊寄存器的延时写入，以及寄存器窗口机制的验证。
- 我们赋予了验证规则直接风格的（direct-style）、基于语义的解释，将 SPARCv8 的 `retl` 指令视为当前函数结束的标志，并在此基础上建立了验证逻辑的可靠性。以方便未来将 SPARCv8 的验证工作与之前 C 程序的验证工作链接起来。
- 我们用我们的验证逻辑验证了某航天嵌入式操作系统的任务上下文切换程序的主要功能模块。该模块的实现涉及到内部函数的调用，以及 SPARCv8 的三个主要特性，很好地说明了我们的验证逻辑对函数调用以及 SPARCv8 特性验证的支持。不过，我们的验证逻辑并不支持任务上下文切换程序执行结束时，返回地址的切换，这些工作放到精化验证工作中去完善。
- 我们提出了一种能够保证 SPARCv8 汇编程序与其所对应的汇编原语之间上下文精化关系的模拟关系。并证明了一个简化版的 SPARCv8 实现的任务上下文切换程序和抽象任务调度原语 `switch` 之间满足该模拟关系。

未来工作

- *对并发 SPARCv8 程序验证的支持*

  本文所提出的验证逻辑是针对串行的 SPARCv8 汇编程序设计的，而现在大多数操作系统是运行在并发和多核环境下的，我们希望在未来扩展我们的验证逻辑，使它支持对并发 SPARCv8 汇编程序的验证。目前，已经有很多关于并发程序的验证工作，例如并发分离逻辑[34-35] 以及基于 “依赖-保证” 的程序推理[36]，并且这些方法已经在汇编代码的验证工作中得到运用[37-38]，为我们未来工作的扩展提供了理论基础和参考。

- *完善和扩展 SPARCv8 程序和其所对应抽象原语之间精化关系的验证工作*

  本文在第6章中提出了一种可以保证 SPARCv8 程序和其所对应抽象原语之间精化关系的模拟关系。这一章的内容还没有在 Coq 辅助定理证明工具中实现，需要未来去完善。另外，该模拟关系主要是为了解决验证任务上下

文切换程序和 switch 原语之间上下文精化关系提出来的，缺乏对如任务创建、中断管理等相关抽象原语的支持。另外，我们还希望扩展本文所提出的 SPARCv8 程序验证逻辑，使它支持精化验证。

- 使用一些自动证明策略提高 SPARCv8 程序的验证效率

  本文采用前后断言的方法来刻画程序规范，目的是为了复用之前操作系统验证框架中的一些自动证明策略[22]。本文工作复用了之前验证工作中的一些自动证明策略（例如：sep_cancel，参见5.3节的介绍）。不过，之前工作中开发的自动证明策略是基于 C 程序验证逻辑的，很大一部分工作难以直接复用，我们希望在后续的工作中开发更多基于 SPARCv8 验证逻辑的证明策略。另外，Berdine[39-40] 等人提出了一些分离逻辑上的自动推理方法，我们也希望在未来工作中将这些技术应用到 SPARCv8 程序验证工作中。

参 考 文 献

[1] XU F, FU M, FENG X, et al. A practical verification framework for preemptive os kernels [C]//CAV. 2016: 59-79.

[2] KLEIN G, ELPHINSTONE K, HEISER G, et al. seL4: Formal Verification of an OS Kernel [C]//SOSP. 2009: 207-220.

[3] GU R, KOENIG J, RAMANANANDRO T, et al. Deep specifications and certified abstraction layers[C]//POPL. 2015: 595-608.

[4] XU F, FU M, FENG X, et al. A practical verification framework for preemptive os kernels [C]//CAV. 2016: 59-79.

[5] SPARCv8[EB/OL]. https://gaisler.com/doc/sparcv8.pdf.

[6] ZHE H, SANAN D, TIU A, et al. An Executable Formalisation of the SPARCv8 Instruction Set Architecture: A Case Study for the LEON3 Processor[C]//FM. 2016.

[7] WANG J, FU M, QIAO L, et al. Formalizing SPARCv8 Instruction Set Architecture in Coq [C]//SETTA. 2017.

[8] The Coq Development Team: The Coq proof assistant[EB/OL]. http://coq.inria.fr.

[9] YU D, NADEEM A H, SHAO Z. Building certified libraries for PCC : Dynamic storage allocation[J]. Science of Computer Programming, 2004, 50(1-3):101-127.

[10] TAN G, APPEL A W. A compositional logic for control flow[C]//VMCAI. 2006.

[11] NECULA G C, LEE P. Safe Kernel Extensions Without Run-Time Checking[C]//Proc.2nd USENIX Symp. on Operating System Design and Impl. 1996: 229-243.

[12] FENG X, SHAO Z, VAYNBERG A, et al. Modular Verification of Assembly Code with Stack-Based Control Abstractions[C]//PLDI. 2006.

[13] APPEL A W. Foundational proof-carrying code[C]//Proc. 16th Annual IEEE Symposium on Logic in Computer Science. 1998: 85-97.

[14] MORRISETT G, CRARY K, GLEW N, et al. Talx86:a realistic typed assembly language[C]// 1999 ACM SIGPLAN Workshop on Compiler Support for System Software. 1996: 25-35.

[15] MORRISETT G, WALKER D, CRARY K, et al. From System F to typed assembly language [C]//POPL. 1998: 85-97.

[16] MYREEN M O, GORDON M J. Hoare logic for realistically modelled machine code[C]// Proc. 13th International Conference on Tools and Algorithms for Construction and Analysis of Systems. 2007.

[17] YANG J, HAWBLITZEL C. Safe to the last instruction: automated verification of a type-safe operating system[C]//PLDI. 2010: 99-110.

[18] NI Z, YU D, SHAO Z. Using XCAP to Certify Realistic Systems code: Machine context management[C]//TPHOLs. 2007.

[19] NI Z, SHAO Z. Certified Assembly Programming with Embedded Code Pointers[C]//POPL. 2006: 320-333.

[20] GUO Y, FENG X, SHAO Z, et al. Modular verification of concurrent thread management[C]// APLAS. 2012: 315-331.

[21] GU R, SHAO Z, KIM J, et al. Certified Concurrent Abstraction Layers[C]//PLDI. 2018: 646-661.

[22] CAO J, FU M, FENG X. Practical Tactics for Verifying C Programs in Coq[C]//CPP. 2015: 97-108.

[23] APPEL A W. Tactics for Separation Logic[EB/OL]. 2006. http://www.cs.princeton.edu/~appel/papers/septacs.pdf.

[24] MCCREIGHT A. Practical Tactics for Separation Logic[J]. 2009:343-358.

[25] SPARC[EB/OL]. https://en.wikipedia.org/wiki/SPARC.

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

[27] REYNOLDS J C. Separation Logic: A Logic for SharedMutable Data Structures[C]//LICS. 2002: 55-74.

[28] LEROY X. A formally verified compiler back-end[J]. Journal of Automated Reasoning, 2009, 43(4):363-446.

[29] LIANG H, FENG X, FU M. A rely-guarantee-based simulation for verifying concurrent program transformations[C]//POPL. 2012: 455-468.

[30] STEWART G, BERINGER L, CUELLAR S, et al. Compositional Compcert[C]//POPL. 2015: 275-287.

[31] Program logic for SPARCv8 implementation in Coq (project code)[EB/OL]. https://github.com/jpzha/VeriSparc.

[32] LEROY X, BLAZY S. Formal verification of a C-like memory model and its uses for verifying program transformations[J]. Journal of Automated Reasoning, 2018, 41(1):1-31.

[33] JIANG H, LIANG H, XIAO S, et al. Towards Certified Separate Compilation for Concurrent Program[M]//PLDI. 2019.

[34] BROOKS S. A Semantics for Concurrent Separation Logic[J]. Lecture Notes in Computer Science, 2004, 3170.

[35] O'HEARN P W. Resources, Concurrency and Local Reasoning[J]. Theoretical Computer Science, 2007, 375(1-3):271-307.

[36] XU Q, DE ROEVER W P, HE J. The Relay-Guarantee Method for Verifying Shared Variable

Concurrent Programs[J]. Formal Aspects of Computing, 1997, 9(2):149-174.

[37] FENG X, SHAO Z. Modular Verification of Concurrent Assembly Code with Dynamic Thread Creation and Termination[C]//ICFP. 2005: 254-267.

[38] FENG X, SHAO Z, GUO Y, et al. Certifying low-level programs with hardware interrupts and preemptive threads[C]//PLDI. 2008: 170-182.

[39] BERDINE J, CALCAGNO C, O'HEARN P. Symbolic execution with separation logic[C]// APLAS. 2005.

[40] BERDINE J, CALCAGNO C, O'HEARN P. Smallfoot: Modular automatic assertion checking with separation logic[C]//FMCO. 2005.

附录 A　线程局部操作语义

我们在这一部分给出了线程局部操作语义，线程局部操作语义中用到的例如内存分配 args_alloc，表达式计算 $[\![\_]\!]_{\_}$ 都在第6.2中给出了定义，此处就不再重复定义。

$$\frac{\Gamma(\mathtt{f}) = (\_, \mathcal{D}_1, \mathcal{D}_2, \mathtt{s}) \quad [\![\overline{\mathsf{e}}]\!]_{(\mathrm{E},\Sigma)} = \overline{v} \quad \mathsf{args\_alloc}(\overline{v}, \mathcal{D}_1 \cdot \mathcal{D}_2, \mathrm{E}, \Sigma, \mathrm{E}', \Sigma')}{\Gamma \vdash ((\mathtt{f}(\overline{\mathsf{e}}), \kappa, \mathrm{E}), \Sigma) \xrightarrow{\tau} ((\mathtt{s}, \kappa, \mathrm{E}'), \Sigma')}$$

$$\frac{\mathtt{f} \notin \mathrm{dom}(\Gamma) \quad [\![\overline{\mathsf{e}}]\!]_{(\mathrm{E},\Sigma)} = \overline{v}}{\Gamma \vdash ((\mathtt{f}(\overline{\mathsf{e}}), \kappa, \mathrm{E}), \Sigma) \xrightarrow{\mathtt{call}(\mathtt{f}, \overline{v})} ((\mathbf{skip}, \kappa, \mathrm{E}), \Sigma)}$$

$$\frac{\begin{array}{c}\Gamma(\mathtt{f}) = (ty, \mathcal{D}_1 \cdot \mathcal{D}_2, \mathtt{s}) \quad \langle x \rangle_{(\mathrm{E},\Sigma)} = (\_, ty) \quad [\![\overline{\mathsf{e}}]\!]_{(\mathrm{E},\Sigma)} = \overline{v} \\ \mathsf{args\_alloc}(\overline{v}, \mathcal{D}_1 \cdot \mathcal{D}_2, \mathrm{E}, \Sigma, \mathrm{E}', \Sigma')\end{array}}{\Gamma \vdash ((x := \mathtt{f}(\overline{\mathsf{e}}), \kappa, \mathrm{E}), \Sigma) \xrightarrow{\tau} ((\mathtt{s}, (x := \_) \cdot \kappa, \mathrm{E}'), \Sigma')}$$

$$\frac{\begin{array}{c}[\![\mathsf{e}]\!]_{(\mathrm{E},\Sigma)} = v \quad \langle x \rangle_{(\mathrm{E},\Sigma)} = (l, ty) \quad \mathsf{tym}(v, ty) \\ \Sigma = (\mathrm{G}, M) \quad M' = M\{l \rightsquigarrow v\} \quad \Sigma' = (\mathrm{G}, M')\end{array}}{\Gamma \vdash ((x := \mathsf{e}, \kappa, \mathrm{E}), \Sigma) \xrightarrow{\tau} ((\mathbf{skip}, \kappa, \mathrm{E}), \Sigma')}$$

$$\frac{}{\Gamma \vdash ((\mathtt{s}_1; \mathtt{s}_2, \kappa, \mathrm{E}), \Sigma) \xrightarrow{\tau} ((\mathtt{s}_1, \mathtt{s}_2 \cdot \kappa, \mathrm{E}), \Sigma)}$$

$$\frac{[\![\mathsf{b}]\!]_{(\mathrm{E},\Sigma)} = \mathbf{true}}{\Gamma \vdash ((\mathbf{if}\,(\mathsf{b})\,\mathbf{then}\,\{\mathtt{s}_1\}\,\mathbf{else}\,\{\mathtt{s}_2\}, \kappa, \mathrm{E}), \Sigma) \xrightarrow{\tau} ((\mathtt{s}_1, \kappa, \mathrm{E}), \Sigma)}$$

$$\frac{[\![\mathsf{b}]\!]_{(\mathrm{E},\Sigma)} = \mathbf{false}}{\Gamma \vdash ((\mathbf{if}\,(\mathsf{b})\,\mathbf{then}\,\{\mathtt{s}_1\}\,\mathbf{else}\,\{\mathtt{s}_2\}, \kappa, \mathrm{E}), \Sigma) \xrightarrow{\tau} ((\mathtt{s}_2, \kappa, \mathrm{E}), \Sigma)}$$

图 A.1　线程局部操作语义 I

$$\frac{[\![\mathsf{b}]\!]_{(\mathrm{E},\Sigma)} = \mathbf{true}}{\Gamma \vdash ((\mathbf{while}\,(\mathsf{b})\,\{\mathtt{s}\},\kappa,\mathrm{E}),\Sigma) \xrightarrow{\tau} ((\mathtt{s},(\mathbf{while}\,(\mathsf{b})\,\{\mathtt{s}\})\cdot\kappa,\mathrm{E}),\Sigma)}$$

$$\frac{[\![\mathsf{b}]\!]_{(\mathrm{E},\Sigma)} = \mathbf{false}}{\Gamma \vdash ((\mathbf{while}\,(\mathsf{b})\,\{\mathtt{s}\},\kappa,\mathrm{E}),\Sigma) \xrightarrow{\tau} ((\mathbf{skip},\kappa,\mathrm{E}),\Sigma)}$$

$$\frac{[\![\mathsf{e}]\!]_{(\mathrm{E},\Sigma)} = v}{\Gamma \vdash ((\mathbf{return}\ \mathsf{e},(x := \_)\cdot\kappa,\mathrm{E}),\Sigma) \xrightarrow{\tau} ((x := v,\kappa,\mathrm{E}),\Sigma)}$$

$$\frac{[\![\mathsf{e}]\!]_{(\mathrm{E},\Sigma)} = v \qquad \kappa \neq (x := \_)\cdot\kappa'}{\Gamma \vdash ((\mathbf{return}\ \mathsf{e},\kappa,\mathrm{E}),\Sigma) \xrightarrow{\tau} ((\mathbf{skip},\kappa,\mathrm{E}),\Sigma)}$$

$$\frac{[\![\mathsf{e}]\!]_{(\mathrm{E},\Sigma)} = v}{\Gamma \vdash ((\mathbf{print}\ \mathsf{e},\kappa,\mathrm{E}),\Sigma) \xrightarrow{\mathtt{out}(v)} (\mathbf{skip},\kappa,\Sigma)}$$

$$\frac{}{\Gamma \vdash ((\mathbf{skip},\mathtt{s}\cdot\kappa,\mathrm{E}),\Sigma) \xrightarrow{\tau} ((\mathtt{s},\kappa,\mathrm{E}),\Sigma)}$$

图 A.2　线程局部操作语义 II

$$\begin{aligned}
&\mathsf{args_alloc}(v::\overline{v},(x,ty)::\mathcal{D},\mathrm{E},\Sigma,\mathrm{E}',\Sigma') \stackrel{\text{def}}{=}\\
&\quad \exists b.\,\mathsf{fresh}(b,M) \wedge M' = M\{(b,0) \rightsquigarrow v\} \wedge x \notin \mathrm{dom}(\mathrm{E})\\
&\qquad \wedge \mathsf{args_alloc}(\overline{v},\mathcal{D},\mathrm{E}\{x \rightsquigarrow ((b,0),ty)\},(\mathrm{G},M'),\mathrm{E}',\Sigma')\\
&\quad \text{where } \Sigma = (\mathrm{G},M)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{args_alloc}(\mathrm{nil},(x,ty)::\mathcal{D},\mathrm{E},\Sigma,\mathrm{E}',\Sigma') \stackrel{\text{def}}{=}\\
&\quad \exists b.\,\mathsf{fresh}(b,M) \wedge M' = M\{(b,0) \rightsquigarrow \mathtt{undef}\} \wedge x \notin \mathrm{dom}(\mathrm{E})\\
&\qquad \wedge \mathsf{args_alloc}(\mathrm{nil},\mathcal{D},\mathrm{E}\{x \rightsquigarrow ((b,0),ty)\},(\mathrm{G},M'),\mathrm{E}',\Sigma')\\
&\quad \text{where } \Sigma = (\mathrm{G},M)
\end{aligned}$$

$$\mathsf{args\_alloc}(\mathrm{nil},\mathrm{nil},\mathrm{E},\Sigma,\mathrm{E}',\Sigma') \stackrel{\text{def}}{=} \mathrm{E} = \mathrm{E}' \wedge \Sigma = \Sigma'$$

$$\mathsf{fresh}(b,M) \stackrel{\text{def}}{=} \forall ofs.\,(b,ofs) \notin \mathrm{dom}(M)$$

图 A.3　与内存分配相关的操作定义

$$
[\![\overline{\mathsf{e}}]\!]_{(\mathrm{E},\Sigma)} \stackrel{\text{def}}{=} \begin{cases} v :: \overline{v} & \textbf{if } \overline{\mathsf{e}} = \mathsf{e} :: \overline{\mathsf{e}}',\ [\![\mathsf{e}]\!]_{(\mathrm{E},\Sigma)} = v,\ [\![\mathsf{e}]\!]_{(\mathrm{E},\Sigma)} = \overline{v} \\ \mathrm{nil} & \textbf{if } \overline{\mathsf{e}} = \mathrm{nil} \\ \bot & \textbf{otherwise} \end{cases}
$$

$$
[\![\mathsf{e}]\!]_{(\mathrm{E},\Sigma)} \stackrel{\text{def}}{=} \begin{cases} v & \textbf{if } \mathsf{e} = x,\ \Sigma = (\_, M), \\ & (l, ty) = \mathrm{E}(x),\ M(l) = v,\ \mathsf{tym}(v, ty) \\ v & \textbf{if } \mathsf{e} = x,\ x \notin \mathrm{dom}(\mathrm{E}),\ \Sigma = (\mathrm{G}, M) \\ & (l, ty) = \mathrm{G}(x),\ M(l) = v,\ \mathsf{tym}(v, ty) \\ v & \textbf{if } \mathsf{e} = v \\ v_1 + v_2 & \textbf{if } \mathsf{e} = \mathsf{e}_1 + \mathsf{e}_2,\ [\![\mathsf{e}_1]\!]_{(\mathrm{E},\Sigma)} = v_1,\ [\![\mathsf{e}_2]\!]_{(\mathrm{E},\Sigma)} = v_2 \\ v & \textbf{if } \mathsf{e} = [\mathsf{e}],\ [\![\mathsf{e}]\!]_{(\mathrm{E},\Sigma)} = l,\ \Sigma.M(l) = v \\ \bot & \textbf{otherwise} \end{cases}
$$

$$
\mathsf{tyms}(\overline{v}, \overline{ty}) \stackrel{\text{def}}{=} \begin{cases} \mathsf{tym}(v, ty) \wedge \mathsf{tyms}(\overline{v'}, \overline{ty}') & \textbf{if } \overline{v} = v :: \overline{v}', \overline{ty} = ty :: \overline{ty}' \\ \textbf{True} & \textbf{if } \overline{v} = \mathrm{nil}, \overline{ty} = \mathrm{nil} \\ \textbf{False} & \textbf{otherwise} \end{cases}
$$

$$
\langle x \rangle_{(\mathrm{E},\Sigma)} \stackrel{\text{def}}{=} \begin{cases} \mathrm{E}(x) & \textbf{if } x \in \mathrm{dom}(\mathrm{E}) \\ \mathrm{G}(x) & \textbf{if } x \notin \mathrm{dom}(\mathrm{E}), \\ & \quad \Sigma = (\mathrm{G}, \_) \\ \bot & \textbf{otherwise} \end{cases}
\qquad
\mathsf{tym}(v, ty) \stackrel{\text{def}}{=} \begin{cases} \textbf{true} & \textbf{if } v = n,\ ty = \texttt{Tint} \\ \textbf{true} & \textbf{if } v = l \text{ or } \texttt{null}, \\ & \quad ty = \texttt{Tptr} \\ \textbf{false} & \textbf{otherwise} \end{cases}
$$

$$
[\![\mathsf{b}]\!]_{(\mathrm{E},\Sigma)} \stackrel{\text{def}}{=} \begin{cases} \textbf{true} & \textbf{if } \mathsf{b} = \textbf{true} \\ \textbf{false} & \textbf{if } \mathsf{b} = \textbf{false} \\ v_1 = v_2 & \textbf{if } \mathsf{b} = (\mathsf{e}_1 = \mathsf{e}_2),\ [\![\mathsf{e}_1]\!]_{(\mathrm{E},\Sigma)} = v_1, [\![\mathsf{e}_2]\!]_{(\mathrm{E},\Sigma)} = v_2 \\ \neg bv & \textbf{if } [\![\mathsf{b}]\!]_{(\mathrm{E},\Sigma)} = bv \\ \bot & \textbf{otherwise} \end{cases}
$$

$$
\mathsf{getTyls}(\mathcal{D}) \stackrel{\text{def}}{=} \begin{cases} ty :: \mathsf{getTyls}(\mathcal{D}') & \textbf{if } \mathcal{D} = (x, ty) :: \mathcal{D}' \\ \mathrm{nil} & \textbf{if } \mathcal{D} = \mathrm{nil} \end{cases}
$$

图 A.4　表达式的运算及类型匹配

附录 B　简单指令转移规则

我们在图B.1中给出了第6.3中的简单指令转移规则（$(\mathtt{i}, \_) \bullet\!\!\rightarrow \_$）。其中 **save** 和 **restore** 的定义和第3.1 中图3.5中 **save** 和 **restore** 的定义相同。

$$\frac{R(\mathtt{r}_s) = v_1 \quad [\![\mathtt{o}]\!]_R = v_2 \quad R' = R\{\mathtt{r}_d \rightsquigarrow v_1 + v_2\}}{(\mathtt{add}\ \mathtt{r}_s\ \mathtt{o}\ \mathtt{r}_d, (m, (R, F), D)) \bullet\!\!\rightarrow (m, (R', F), D)}$$

$$\frac{[\![\mathtt{a}]\!]_R = (b, ofs) \quad m(b, ofs) = v \quad R' = R\{\mathtt{r}_d \rightsquigarrow v\}}{(\mathtt{ld}\ \mathtt{a}\ \mathtt{r}_d, (m, (R, F), D)) \bullet\!\!\rightarrow (m, (R', F), D)}$$

$$\frac{R(\mathtt{r}_s) = v \quad [\![\mathtt{a}]\!]_R = (b, ofs) \quad (b, ofs) \in \mathrm{dom}(m) \quad m' = m\{(b, ofs) \rightsquigarrow v\}}{(\mathtt{st}\ \mathtt{r}_s\ \mathtt{a}, (m, (R, F), D)) \bullet\!\!\rightarrow (m', (R, F), D)}$$

$$\frac{}{(\mathtt{nop}, (m, Q, D)) \bullet\!\!\rightarrow (m, Q, D)}$$

$$\frac{R(\mathtt{sr}) = v \quad R' = R\{\mathtt{r}_d \rightsquigarrow v\}}{(\mathtt{rd}\ \mathtt{sr}\ \mathtt{r}_d, (m, (R, F), D)) \bullet\!\!\rightarrow (m, (R', F), D)}$$

$$\frac{R(\mathtt{r}_s) = v_1 \quad [\![\mathtt{o}]\!]_R = v_2 \quad v = v_1 \mathbf{xor} v_2 \quad D' = \mathbf{set\_delay}(\mathtt{sr}, v, D)}{(\mathtt{wr}\ \mathtt{r}_s\ \mathtt{o}\ \mathtt{sr}, (m, (R, F), D)) \bullet\!\!\rightarrow (m, (R, F), D')}$$

$$\frac{\mathbf{save}(R, F) = (R', F') \quad [\![\mathtt{o}]\!]_R = v \quad R'' = R'\{\mathtt{r}_d \rightsquigarrow R(\mathtt{r}_s) + v\}}{(\mathtt{save}\ \mathtt{r}_s\ \mathtt{o}\ \mathtt{r}_d, (m, (R, F), D)) \bullet\!\!\rightarrow (m, (R'', F'), D)}$$

$$\frac{\mathbf{restore}(R, F) = (R', F') \quad [\![\mathtt{o}]\!]_R = v \quad R'' = R'\{\mathtt{r}_d \rightsquigarrow R(\mathtt{r}_s) + v\}}{(\mathtt{restore}\ \mathtt{r}_s\ \mathtt{o}\ \mathtt{r}_d, (m, (R, F), D)) \bullet\!\!\rightarrow (m, (R'', F'), D)}$$

$$[\![\mathtt{o}]\!]_R ::= \begin{cases} R(\mathtt{r}) & \textbf{if } \mathtt{o} = \mathtt{r} \\ v & \textbf{if } \mathtt{o} = v \\ \bot & \textbf{otherwise} \end{cases} \qquad [\![\mathtt{a}]\!]_R ::= \begin{cases} [\![\mathtt{o}]\!]_R & \textbf{if } \mathtt{a} = \mathtt{o} \\ v_1 + v_2 & \textbf{if } \mathtt{a} = \mathtt{r} + \mathtt{o}, R(\mathtt{r}) = v_1, [\![\mathtt{o}]\!]_R = v_2 \\ \bot & \textbf{otherwise} \end{cases}$$

图 B.1　简单指令转移规则

附录 C　SPARCv8 精化验证内容补充

C.1　程序执行的安全性与封闭型

定理6.1中的前提 2 和前提 3 分别要求高层程序是安全的（Safe），并且每个高层函数的执行都只会访问自己的局部内存和所有线程的共享内存（ReachClose）。

定义 C.1 (安全性)　$\mathsf{Safe}(\mathbb{W}) \stackrel{\text{def}}{=} \neg\exists \mathcal{B}.\, \mathit{Etr}(\mathbb{W}, \mathcal{B} :: \mathbf{abort})$；
$\mathsf{Safe}(\mathbb{P}, \Sigma) \stackrel{\text{def}}{=} (\exists\, \mathbb{W}.\, (\mathbb{P}, \Sigma) \stackrel{load}{\Longrightarrow} \mathbb{W}) \wedge (\forall\, \mathbb{W}.\, ((\mathbb{P}, \Sigma) \stackrel{load}{\Longrightarrow} \mathbb{W}) \Longrightarrow \mathsf{Safe}(\mathbb{W}))$；
$\mathsf{Safe}(\mathbb{P}, \mathbb{S}) \stackrel{\text{def}}{=} \forall\, \Sigma.\, (\mathrm{dom}(\Sigma.M) = \mathbb{S} \wedge \mathsf{closed}(\mathbb{S}, \Sigma)) \Longrightarrow \mathsf{Safe}(\mathbb{P}, \Sigma)$。

定义 C.2 (封闭性)　$\mathsf{ReachClose}(\Gamma, \mathbb{S})$ 成立，当且仅当，对于任意的 $\mathtt{f}$，Σ，ty，$\mathcal{D}_1$，$\mathcal{D}_2$ 和 $\overline{v}$，如果下述成立：

1. $\Gamma(\mathtt{f}) = (ty, \mathcal{D}_1, \mathcal{D}_2, \mathtt{s})$；
2. $\mathbb{S} \subseteq \mathrm{dom}(\Sigma.M)$，$\mathsf{closed}(\mathbb{S}, \Sigma.M)$

则：$\mathsf{RC}((\mathtt{f}(\overline{v}), \circ, \varnothing), \Sigma, \Gamma, \mathbb{S})$ 成立。
若 $\mathsf{RC}(\mathcal{K}, \Sigma, \Gamma, \mathbb{S})$，则对于任意的 Σ' 和 α，如果有 $\mathsf{R}(\Sigma.M, \Sigma'.M, \mathsf{range}(\mathcal{K}.\mathrm{E}), \mathbb{S})$，$\mathsf{closed}(\mathbb{S}, \Sigma'.M)$，并且对于任意的 $\mathcal{K}'$，Σ''，若 $\Gamma \vdash (\mathcal{K}, \Sigma') \xrightarrow{\alpha} (\mathcal{K}', \Sigma'')$，则：$\Sigma'.M \stackrel{\mathrm{dom}(\Sigma'.M) - \mathbb{S} - \mathsf{range}(\mathcal{K}.\mathrm{E})}{\Longrightarrow\!\!\!=\!\!\!=} \Sigma''.M$，$\mathsf{closed}(\mathbb{S}, \Sigma''.M)$，$\mathsf{RC}(\mathcal{K}', \Sigma'', \Gamma, \mathbb{S})$ 成立。

定义C.1和C.2分别给出了程序安全性和代码执行封闭性的定义。安全性的定义比较容易理解，即程序在执行过程中不会发生中止 **abort**。而封闭性则是说代码 Γ 中的任意函数 $\mathtt{f}$ 的执行都满足 RC 性质。$\mathsf{RC}(\mathcal{K}, \Sigma, \Gamma, \mathbb{S})$ 则是说程序从状态 $(\mathcal{K}, \Sigma)$ 开始执行，任何一步都不会访问除自己局部和共享内存以外的内存空间，即使在执行过程中与环境发生交互，只要环境对内存的修改满足 R，并且与环境交互后共享内存还是封闭的。我们在图C.1中给出了定义 ReachClose 中用到的辅助定义，其中 range(E) 表示求符号表 E 的值域，即符号表 E 中变量所对应的内存单元地址集合，$M \stackrel{d}{=\!=} M'$ 表示内存 M 和 M' 中地址集合 d 中存的内容相同。$\mathsf{R}(M, M', d, \mathbb{S})$ 则表示，我们要求环境不能修改地址集合 d 中的内存单元，并且也不会释放内存中 d 和 $\mathbb{S}$ 中的地址所对应的内存单元。

$$M \stackrel{d}{=\!=} M' \stackrel{\text{def}}{=} \forall\, l \in d.\, l \notin (\mathrm{dom}(M) \cup \mathrm{dom}(M')) \vee l \in (\mathrm{dom}(M) \cap \mathrm{dom}(M')) \wedge M(l) = M'(l)$$

$$\mathsf{R}(M, M', d, \mathbb{S}) \stackrel{\text{def}}{=} m \stackrel{d}{=\!=} m' \wedge (d \cup \mathbb{S}) \subseteq \mathrm{dom}(m') \qquad d \in \mathcal{P}(\mathrm{Addr})$$

图 C.1　ReachClose 定义中用到的辅助定义

C.2　SPARCv8 中的函数调用惯例

SPARCv8 中，当发生函数调用时，调用者会暴露一部分内存给被调用者，主要包括两部分：用于*传递实参*的栈空间和用于*保存寄存器窗口*的栈空间。图C.2中定义 caller_explore 来描述这调用者暴露给被调用者的栈空间，其中，我们用 Arguments 描述用于*传递实参*的栈空间，而定义 $\Downarrow$ 来描述用于*保存寄存器窗口*的栈空间。对于参数，前 6 个参数通过 $\%\mathrm{o}_0 \sim \%\mathrm{o}_5$ 传递，而其它参数则依次保存在栈空间偏移地址为 16 的内存单元（前 16 个内存单元用于当前窗口的 local 和 in 寄存器）。

$$\frac{R(\%\mathtt{sp}) = (b, 0) \quad (R, F) \Downarrow \mathrm{dom}(m') \quad \mathsf{wdfmls}(R, F) \quad \mathsf{Arguments}(b, \overline{v}, m_{\mathrm{args}}, 0) \quad m_l = m_{\mathrm{args}} \uplus m'}{\mathsf{caller\_explore}(R, F, \overline{v}, m_l)}$$

$$\frac{d = \{(b,0), \cdots, (b,15)\} \uplus d_2 \quad R(\%\mathtt{sp}) = (b,0) \quad \mathbf{win\_valid}(R(\mathtt{cwp}), R) \quad \mathbf{restore}(R, F) = (R', F') \quad (R', F') \Downarrow d_2}{(R, F) \Downarrow d} \qquad \frac{\neg\mathbf{win\_valid}(R(\mathtt{cwp}), R)}{(R, F) \Downarrow \varnothing}$$

$$\frac{\overline{v} = v :: \overline{v}' \quad R(\%\mathrm{o}_j) = v \quad \mathsf{Arguments}(b, \overline{v}', m', j+1) \quad 0 \leqslant j < 6 \quad \mathrm{dom}(m) = \{(b, 17+i)\} \uplus \mathrm{dom}(m')}{\mathsf{Arguments}(b, \overline{v}, m, j)}$$

$$\frac{\overline{v} = v :: \overline{v}' \quad \mathsf{Arguments}(b, \overline{v}', m', j+1) \quad j \geqslant 6 \quad m = \{(b, 16+j) \rightsquigarrow v\} \uplus m'}{\mathsf{Arguments}(b, \overline{v}, m, j)} \qquad \frac{}{\mathsf{Arguments}(b, \mathrm{nil}, \varnothing, j)}$$

$$\mathsf{wdfmls}(R, F) \stackrel{\text{def}}{=} \exists\, w_{id}, v_i.\ R(\mathtt{cwp}) = w_{id} \wedge R(\mathtt{wim}) = 2^{v_i} \quad w_{id} \neq v_i \wedge 0 \leqslant w_{id}, v_i < N \wedge |F| = 2N-3$$

图 C.2　调用者暴露为被调用者的栈空间

我们再定义函数调用前后寄存器状态之间的关系 "$Q \propto (Q', m)$"（定义在图C.3中），SPARCv8 函数调用前后寄存器状态的关系，其实就是函数调用前后寄存器窗口之间的关系。我们用寄存器文件 R 和帧链表 F 来表示 SPARCv8 中的寄存器窗口。我们依次比较寄存器状态 Q 和 Q' 的各个窗口，如果 Q 和 Q' 的

$$\frac{Q = (R, F) \quad Q' = (R', F') \quad \mathsf{wdfmls}(R, F) \quad \mathsf{wdfmls}(R', F') \quad (R, F) \propto (R', F', m)}{Q \propto (Q', m)}$$

$$\frac{R([\mathsf{local}]) = R'([\mathsf{local}]) \quad R([\mathsf{in}]) = R'([\mathsf{in}]) \quad R(\%\mathtt{sp}) = R'(\%\mathtt{sp}) \quad R(\mathtt{r}_{15}) = R'(\mathtt{r}_{15})}{R \propto R'}$$

$$\frac{\begin{array}{c}\mathbf{win\_valid}(R(\mathtt{cwp}), R) \quad \mathbf{win\_valid}(R'(\mathtt{cwp}), R') \\ R(\%\mathtt{sp}) = R'(\%\mathtt{sp}) = (b, 0) \quad \{(b, 0), \cdots, (b, 15)\} \subseteq \mathrm{dom}(m') \\ \mathbf{restore}(R, F) = (R_1, F_1) \quad \mathbf{restore}(R', F') = (R'_1, F'_1) \quad R \propto R' \\ (R_1, F_1) \propto (R'_1, F'_1, m \backslash \{(b, 0), \cdots, (b, 15)\})\end{array}}{(R, F) \propto (R', F', m')}$$

$$\frac{\begin{array}{c}\mathbf{win\_valid}(R(\mathtt{cwp}), R) \quad R(\%\mathtt{sp}) = (b, 0) \quad \neg\mathbf{win\_valid}(R'(\mathtt{cwp}), R') \\ m'([(b, 0), \cdots, (b, 7)]) = R([\mathsf{local}]) \quad m'([(b, 8), \cdots, (b, 15)]) = R([\mathsf{in}]) \\ \mathbf{restore}(R, F) = (R_1, F_1) \quad (R_1, F_1) \propto (R', F', m\{(b, 0), \cdots, (b, 15)\})\end{array}}{(R, F) \propto (R', F', m')}$$

$$\frac{\neg\mathbf{win\_valid}(R(\mathtt{cwp}), R) \quad \neg\mathbf{win\_valid}(R'(\mathtt{cwp}), R')}{(R, F) \propto (R', F', m')}$$

$$\frac{\begin{array}{c}R([\mathsf{global}]) = m([(t, ofs_g), \cdots, (t, ofs_g + 7)]) \\ R(\mathtt{cwp}) = \mathbf{next\_cwp}(n) \quad R(\mathtt{wim}) = 2^n \quad 0 \leqslant n < N \\ R(\%\mathtt{sp}) = (b, 0) \quad R([\mathsf{local}]) = m([(b, 0), \cdots, (b, 7)]) \quad R([\mathsf{in}]) = m([(b, 8), \cdots, (b, 15)]) \\ R(\%\mathtt{sp}) = m(t, ofs_{sp}) \quad R(\mathtt{r}_{15}) = m(t, ofs_{ret})\end{array}}{m \rightarrowtail_t (R, F)}$$

$$\begin{array}{lcll} m([l_1, \cdots, l_n]) & \stackrel{\text{def}}{=} & [m(l_1), \cdots, m(l_n)] & \\ m\{[l_1, \cdots, l_n] \rightsquigarrow \overline{v}\} & \stackrel{\text{def}}{=} & m\{l_1 \rightsquigarrow v_1, \cdots, l_n \rightsquigarrow v_n\} & \text{where } \overline{v} = v_1 :: \cdots :: v_n \end{array}$$

图 C.3　SPARCv8 函数调用前后寄存器状态和从内存恢复寄存器状态

当前窗口都是有效的，那么寄存器文件 R 和 R' 之间满足关系 "$R \propto R'$"，如果 Q' 的窗口无效，则说明 Q 的当前窗口的内容已经被保存到内存中。

另外我们还定义了 $m \rightarrowtail_t (R, F)$ 表示从内存 m 中恢复线程 t 运行时的寄存器

状态。我们从线程 t 的任务控制块 TCB 中恢复 global，%sp 和 $\mathtt{r}_{15}$ 寄存器，从栈空间恢复 local 和 in 寄存器。

C.3　客户端程序模拟关系

我们将调用汇编原语实现的底层代码 C 和调用抽象汇编原语的高层程序 Γ 称为*客户端程序*。我们希望 C 是由 Γ 编译生成，并且编译器能够保存二者之间的一个模拟关系 $C \preccurlyeq_{\varphi,\mathbb{S}} \Gamma$。不过目前主要的编译器验证工作中验证的 CompCert 编译器并不支持编译生成 SPARCv8 目标代码，所以我们暂时假设二者之间存在模拟关系 $C \preccurlyeq_{\varphi,\mathbb{S}} \Gamma$，我们在定义C.3中定义该模拟关系，至于编译器能否保证该模拟关系，放在未来工作中完善。

定义 C.3 (客户端程序模拟关系)　$C \preccurlyeq_{\varphi,\mathbb{S}} \Gamma$ 成立，当且仅当，对于任意的 f，ty，$\mathcal{D}_1$，$\mathcal{D}_2$，s，$\overline{v}$，Σ，R 和 F，如果下述成立：

1. $\Gamma(\mathtt{f}) = (ty, \mathcal{D}_1, \mathcal{D}_2, \mathtt{s})$，$\mathsf{tyms}(\overline{v}, \mathsf{getTyls}(\mathcal{D}_1))$；
2. $m_l \uplus m_s \subseteq m$；
3. $\mathsf{caller\_explore}(R, F, \overline{v}, m_l)$，$\mathsf{initM}(\varphi, \mathbb{S}, \Sigma, m_s)$；

则下述成立：（这里 $\mu = (\varphi|_{\mathbb{S}}, \mathbb{S}, \mathrm{dom}(m_l))$）

$$(C, (m, (R, F), \mathrm{nil}), (\mathtt{f}, 0), (\mathtt{f}, 4)) \preccurlyeq^0_\mu (\Gamma, (\mathtt{f}(\overline{v}), \circ, \varnothing), \Sigma)$$

我们在定义C.4中给出 $(C, \sigma, \mathtt{pc}, \mathtt{npc}) \preccurlyeq^k_\mu (\Gamma, \mathcal{K}, \Sigma)$ 的定义。

定义 C.4　若 $(C, \sigma, \mathtt{pc}, \mathtt{npc}) \preccurlyeq^k_\mu (\Gamma, \mathcal{K}, \Sigma)$ 成立，则 $\mathtt{pc} \in \mathrm{dom}(C)$，并且下述全部成立（这里 $\mu = (\varphi, \mathbb{S}, d)$）：

1. 对于任意的 σ'，$\mathtt{pc}'$ 和 $\mathtt{npc}'$，
   若 $C(\mathtt{pc}) \notin \{\mathtt{call\ f}, \mathtt{retl}\}$，$(C, \sigma, \mathtt{pc}, \mathtt{npc}) :\overset{\tau}{\Longrightarrow} (C, \sigma', \mathtt{pc}', \mathtt{npc}')$，则存在 $\mathcal{K}'$，Σ'，μ'，使得下述成立：
   (a) $\sigma.m \overset{\mathrm{dom}(\sigma.m)-d-\varphi\{\{\mathbb{S}\}\}}{=\!=\!=\!=\!=\!=\!=\!=\!=} \sigma'.m \wedge \mu' = (\varphi, \mathbb{S}, d \cup (\mathrm{dom}(\sigma'.m) - \mathrm{dom}(\sigma.m)))$;
   (b) $\Gamma \vdash (\mathcal{K}, \Sigma) \overset{\tau}{\longrightarrow}^* (\mathcal{K}', \Sigma')$;
   (c) $(C, \sigma', \mathtt{pc}', \mathtt{npc}') \preccurlyeq^k_{\mu'} (\Gamma, \mathcal{K}', \Sigma')$;
2. 对于任意的 $\mathtt{pc}'$ 和 $\mathtt{npc}'$，若 $(C, \sigma, \mathtt{pc}, \mathtt{npc}) :\overset{e}{\Longrightarrow} (C, \sigma, \mathtt{pc}', \mathtt{npc}')$，则存在 $\mathcal{K}'$，$\mathcal{K}''$，Σ'，使得下述成立:
   (a) $\Gamma \vdash (\mathcal{K}, \Sigma) \overset{\tau}{\longrightarrow}^* (\mathcal{K}', \Sigma'), \Gamma \vdash (\mathcal{K}', \Sigma') \overset{e}{\longrightarrow} (\mathcal{K}'', \Sigma')$;
   (b) $(C, \sigma, \mathtt{pc}', \mathtt{npc}') \preccurlyeq^k_\mu (\Gamma, \mathcal{K}'', \Sigma')$
3. 对于任意的 σ'，$\mathtt{pc}'$ 和 $\mathtt{npc}'$，
   若 $C(\mathtt{pc}) = \mathtt{call\ f}$，$(C, \sigma, \mathtt{pc}, \mathtt{npc}) :\overset{\tau}{\Longrightarrow}^2 (C, \sigma', \mathtt{pc}', \mathtt{npc}')$，那么存在

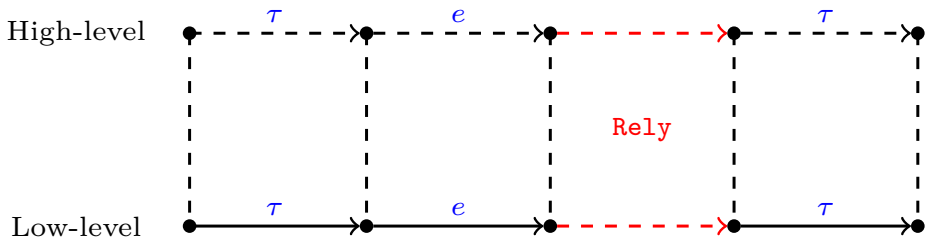

图 C.4　高层和底层客户端程序模拟关系

$\mathcal{K}', \Sigma', \mu', \mathtt{i}$，使得下述成立：

(a) $C(\mathtt{npc}) = \mathtt{i}$；

(b) $\sigma.m \xrightarrow{\mathrm{dom}(\sigma.m)-d-\varphi\{\{\mathbb{S}\}\}} \sigma'.m \wedge \mu' = (\varphi, \mathbb{S}, d \cup (\mathrm{dom}(\sigma'.m) - \mathrm{dom}(\sigma.m)))$；

(c) $\Gamma \vdash (\mathcal{K}, \Sigma) \xrightarrow{\tau}{}^{*} (\mathcal{K}', \Sigma')$；

(d) 下述性质之一成立：

- 要么 $(C, \sigma', \mathtt{pc}', \mathtt{npc}') \preccurlyeq_{\mu}^{k+1} (\Gamma, \mathcal{K}', \Sigma')$；
- 或者存在 $\mathcal{K}''$，$\overline{v}$ 和 m_l，使得：
  - $\Gamma \vdash (\mathcal{K}', \Sigma') \xrightarrow{\mathtt{call(f,}\overline{v}\mathtt{)}} (\mathcal{K}'', \Sigma')$；
  - $\mathsf{LG}(\sigma, \Sigma', \overline{v}, m_l, \mu)$；
  - 对于任意的 σ'，Σ''，$\mathtt{pc}'$ 和 $\mathtt{npc}'$，
    若 $\mathsf{Rely}(\mu, (\mathcal{K}''.\mathrm{E}, \Sigma', \Sigma''), (d - \mathrm{dom}(m_l), \sigma, \sigma'), \mathtt{pc}', \mathtt{npc}')$，则：
    $(C, \sigma', \mathtt{pc}', \mathtt{npc}') \preccurlyeq_{\mu}^{k} (\Gamma, \mathcal{K}'', \Sigma'')$；

4. 对于任意的 σ'，$\mathtt{pc}'$ 和 $\mathtt{npc}'$，
若 $C(\mathtt{pc}) = \mathtt{retl}$，$(C, \sigma, \mathtt{pc}, \mathtt{npc}) :\xrightarrow{\tau}{}^{2} (C, \sigma', \mathtt{pc}', \mathtt{npc}')$，那么存在 $\mathcal{K}'$，Σ'，μ'，$\mathtt{i}$，使得：
   - $C(\mathtt{npc}) = \mathtt{i}$；
   - $\sigma.m \xrightarrow{\mathrm{dom}(\sigma.m)-d-\varphi\{\{\mathbb{S}\}\}} \sigma'.m \wedge \mu' = (\varphi, \mathbb{S}, d \cup (\mathrm{dom}(\sigma'.m) - \mathrm{dom}(\sigma.m)))$；
   - $\Gamma \vdash (\mathcal{K}, \Sigma) \xrightarrow{\tau}{}^{*} (\mathcal{K}', \Sigma')$；
   - 若 $k = 0$，则 $\mathcal{K} = (\mathbf{skip}, \circ, \_)$；否则 $(C, \sigma', \mathtt{pc}', \mathtt{npc}') \preccurlyeq_{\mu'}^{k-1} (\Gamma, \mathcal{K}', \Sigma')$。

5. 若 $(C, \sigma, \mathtt{pc}, \mathtt{npc}) :\xrightarrow{\tau} \mathbf{abort}$，
或者 $C(\mathtt{pc}) \in \{\mathtt{call\ f}, \mathtt{retl}\}$，$(C, \sigma, \mathtt{pc}, \mathtt{npc}) :\xrightarrow{\tau}{}^{2} \mathbf{abort}$，
则 $\Gamma \vdash (\mathcal{K}, \Sigma) \xrightarrow{\tau}{}^{*} \mathbf{abort}$。

我们首先在定义C.3中给出客户端程序之间的模拟关系 $C \preccurlyeq_{\varphi,\mathbb{S}} \Gamma$，它是说对于任何一个函数入口 f，如果它在代码 Γ 中有对应的函数，并且参数 $\overline{v}$ 与该函数参数列表 D_1 中参数类型匹配（$\mathsf{tyms}(\overline{v}, \mathsf{getTyls}(D_1))$），那么对于任意

$$\begin{aligned}\mathsf{LG}(\sigma, \Sigma, \overline{v}, m_l, \mu) \stackrel{\text{def}}{=}\ & D = \text{nil} \wedge \mathsf{Inv}(\varphi, \Sigma.M, m) \wedge \mathsf{closed}(\mathbb{S}, \Sigma.M) \\ & \wedge m_l \subseteq m \wedge \text{dom}(m_l) \subseteq d \wedge \mathsf{caller\_explore}(R, F, \overline{v}, m_l) \\ & \text{where } \sigma = (m, (R, F), D),\ \mu = (\varphi, \mathbb{S}, d)\end{aligned}$$

$$\begin{aligned}\mathsf{Rely}(\mu, (\mathrm{E}, \Sigma, \Sigma'), (d, \sigma, \sigma'), \mathtt{pc}, \mathtt{npc}) \stackrel{\text{def}}{=}\ & \mathsf{R}(M, M', \mathsf{range}(\mathrm{E}), \mathbb{S}) \wedge \mathsf{R}(m, m', d, \varphi\{\{\mathbb{S}\}\}) \\ & \wedge \mathsf{Inv}(\varphi, M', m') \wedge \mathsf{closed}(\mathbb{S}, M') \wedge \mathsf{ConvG}(\sigma, \sigma', \mathtt{pc}, \mathtt{npc}) \\ & \text{where } \mu = (\varphi, \mathbb{S}, d'),\ M = \Sigma.M,\ M' = \Sigma'.M,\ m = \sigma.m,\ m' = \sigma'.m\end{aligned}$$

$$\begin{aligned}\mathsf{ConvG}(\sigma, \sigma', \mathtt{pc}', \mathtt{npc}') \stackrel{\text{def}}{=}\ & Q \propto (Q', m') \wedge \mathtt{pc}' = Q'.R(\mathtt{r}_{15})+8 \wedge \mathtt{npc}' = Q'.R(\mathtt{r}_{15})+12 \\ & \text{where } \sigma = (m, Q, \text{nil}),\ \sigma' = (m', Q', \text{nil})\end{aligned}$$

图 C.5　与环境交互相关的依赖和保证

的底层和高层程序内存（m 和 Σ），如果底层内存 m 可以分为两部分：调用者暴露给被调用者的内存 m_l 以及共享内存 m_s，并且共享内存和高层程序状态之间满足 $\mathsf{initM}(\varphi, \mathbb{S}, \Sigma, m_s)$，$m_l$ 与当前寄存器状态 (R, F) 和参数 $\overline{v}$ 满足关系 $\mathsf{caller\_explore}(R, F, \overline{v}, m_l)$，那么我们可以建立底层和高层程序之间的模拟关系 $(C, (m, (R, F), \text{nil}), (\mathtt{f}, 0), (\mathtt{f}, 4)) \preccurlyeq^0_\mu (\Gamma, (\mathtt{f}(\overline{v}), \circ, \varnothing), \Sigma)$。该模拟关系中的 k 用于记录函数的调用层数，初始状态为 0；而这里的 μ 是一个三元组，包括高层和底层程序共享内存地址之间的映射关系 $\varphi|_{\mathbb{S}}$，高层程序共享内存地址集合 $\mathbb{S}$，以及局部内存地址集合，起始状态下局部内存只有该方法的调用者所暴露的内存地址集合 $\text{dom}(m_l)$， SPARCv8 中调用者暴露给被调用者的栈空间包括：保存实参的内存空间和用于保存寄存器窗口的内存空间，caller_explore 用来描述调用者暴露给被调用者的内存空间。

我们在定义C.4中定义了模拟关系 $(C, \sigma, \mathtt{pc}, \mathtt{npc}) \preccurlyeq^k_\mu (\Gamma, \mathcal{K}, \Sigma)$，参数 k 用来记录函数调用层数。我们通过图C.4来对该模拟关系有一个大概的认识，即如果底层程序走一步不产生外部可见事件，则高层程序可以走 0 步或多步也不产生外部可见事件；如果底层程序走一步产生外部可见事件 e，则高层程序可以走多步产生同样的外部可见事件 e；如果它们与外部环境发生交互，并且环境对程序状态的改变满足 Rely （定义在图C.5中），那么底层和高层程序之间可以继续建立模拟关系。这里的依赖条件 Rely 要求环境不会修改底层和高层程序的局部内存（注意调用者暴露给被调用者的内存 m_l 不是局部内存），并且环境执行结束之后，底层和高层程序内存之间仍满足关系 Inv，并且高层内存还是 “封闭的”（closed）, 另外因为底层程序还包含寄存器状态，所以我们要求环境执行前后底层程序状态满足函数调用惯例 ConvG。ConvG 约束了与环境交互前后寄存器状态的变化（假设与环境交互前后寄存器状态分别为 Q 和 Q'），我们用 $Q \propto (Q', m')$ 表示该约束，因为环境可能将当前方法临时保存在寄存器窗口中的运行时上下文保存到内存中，所以约束需要带上环境执行后的内存状态 m'。另外环境执行结束后

的程序指针 pc′ 和 npc′ 分别为环境执行前保存在 r_{15} 寄存器中的值加 8 和加 12，更详细的介绍以及 $Q \propto (Q', m')$ 的定义参见第C.2节图C.3。

对于底层程序执行的每一步，我们要求它不会修改除当前方法的局部内存和共享内存以外的存储单元，我们用 $\sigma.m \overset{\mathrm{dom}(\sigma.m)-d-\varphi\{\{\mathbb{S}\}\}}{=\!=\!=} \sigma'.m$ 表示该约束，其中 d 是该方法局部内存空间的地址集合，记录在模拟关系的参数 μ 中，并且如果程序执行一步分配了新的内存空间，因为新分配的内存空间属于当前方法的局部内存，所以集合 d 会增大（新分配内存的地址集合为：$\mathrm{dom}(\sigma'.m) - \mathrm{dom}(\sigma.m)$）；底层程序的共享内存地址集合可以通过高层程序共享内存地址集合 $\mathbb{S}$ 经过映射 φ 作用后所得到（表示为 $\varphi\{\{\mathbb{S}\}\}$）。

对于当前指令是 call 指令的情况，底层程序执行两步之后可能有两种情况，要么调用的是内部函数，则高层程序走 0 步或多步后可以继续建立模拟关系；要么调用抽象原语（假设函数标识符是 f），则高层程序走多步后会产生消息 $\mathtt{call}(\mathtt{f}, \overline{v})$，则此时底层和高层程序会与环境发生交互，我们要求与环境发生交互时，底层和高层程序状态之间满足保证 LG，该保证是说，与环境发生交互时，高层和底层内存满足关系 Inv，底层程序的延时队列为空，并且当前方法作为调用者会暴露一部分自己的局部内存给被调用者；此外，如果环境对程序状态的修改满足依赖 Rely，那么与环境交互结束后底层和高层程序之间还能继续建立模拟关系。

对于当前指令是 retl 的情况，如果函数调用层数 k 是 0，那么底层程序走两步之后，该方法结束，通过高层程序走 0 步或多步后也会结束（即 $\mathcal{K} = (\mathbf{skip}, \circ, \_)$），否则函数调用层数减一，底层和高层程序之间继续可以建立模拟关系。另外模拟关系中还给出了底层程序中止（**abort**）的情况，如果底层程序走一步中止，或者当前指令是 call 或 retl，并且走两步后程序中止，则对应的，高层程序也会走多步后中止。

附录 D　switch 原语实现和原语之间满足模拟关系

在证明引理6.2的正确性之前,我们首先给出线程上下文切换程序的 SPARCv8 实现。图D.1给出了 switch 汇编原语实现的主函数，而图D.2 和图D.3中给出的主函数中所调用的内部函数实现。另外代码中涉及到的例如：L0_OFFSET，SP_OFFSET 等描述偏移地址的常量的定义在此处略去。

```
    f_sw :
1       set             TaskCur, %o0
2       ld              [%o0], %o1
3       add             %o1, CONTEXT_OFFSET, %o1
4       call            reg_save
5       nop

    save_windows :
6       call            window_save
7       nop
8       cmp             (1 << (cwp + 1) mod N), wim
9       bz              switch_new_task
10      nop
11      restore
12      jmp             save_windows
13      nop

    switch_new_task :
14      set             TaskCur, %o0
15      set             TaskNew, %o1
16      ld              [%o1], %o1
17      st              %o1, [%o0]
18      add             %o1, CONTEXT_OFFSET,%o1
19      call            reg_restore
20      nop
21      call            window_restore
22      nop

23      retl
24      nop
```

图 D.1　switch 汇编原语实现主函数

```
window_save：
    st          %l0, [%sp + L0_OFFSET]
    st          %l1, [%sp + L1_OFFSET]
    st          %l2, [%sp + L2_OFFSET]
    st          %l3, [%sp + L3_OFFSET]
    st          %l4, [%sp + L4_OFFSET]
    st          %l5, [%sp + L5_OFFSET]
    st          %l6, [%sp + L6_OFFSET]
    st          %l7, [%sp + L7_OFFSET]
    st          %i0, [%sp + I0_OFFSET]
    st          %i1, [%sp + I1_OFFSET]
    st          %i2, [%sp + I2_OFFSET]
    st          %i3, [%sp + I3_OFFSET]
    st          %i4, [%sp + I4_OFFSET]
    st          %i5, [%sp + I5_OFFSET]
    st          %i6, [%sp + I6_OFFSET]
    st          %i7, [%sp + I7_OFFSET]
    retl
    nop

window_restore：
    ld          [%sp + L0_OFFSET], %l0
    ld          [%sp + L1_OFFSET], %l1
    ld          [%sp + L2_OFFSET], %l2
    ld          [%sp + L3_OFFSET], %l3
    ld          [%sp + L4_OFFSET], %l4
    ld          [%sp + L5_OFFSET], %l5
    ld          [%sp + L6_OFFSET], %l6
    ld          [%sp + L7_OFFSET], %l7
    ld          [%sp + I0_OFFSET], %i0
    ld          [%sp + I1_OFFSET], %i1
    ld          [%sp + I2_OFFSET], %i2
    ld          [%sp + I3_OFFSET], %i3
    ld          [%sp + I4_OFFSET], %i4
    ld          [%sp + I5_OFFSET], %i5
    ld          [%sp + I6_OFFSET], %i6
    ld          [%sp + I7_OFFSET], %i7
    retl
    nop
```

图 D.2　内部函数实现 I

```
reg_save：
    st          %sp, [%o1 + SP_OFFSET]
    st          %o7, [%o1 + RET_OFFSET]

    st          %o2, [%o1 + Y_OFFSET]
    st          %g0, [%o1 + G0_OFFSET]
    st          %g1, [%o1 + G1_OFFSET]
    st          %g2, [%o1 + G2_OFFSET]
    st          %g3, [%o1 + G3_OFFSET]
    st          %g4, [%o1 + G4_OFFSET]
    st          %g5, [%o1 + G5_OFFSET]
    st          %g6, [%o1 + G6_OFFSET]
    st          %g7, [%o1 + G7_OFFSET]

    retl
    nop

reg_restore：
    ld          [%o1 + O6_OFFSET], %sp
    ld          [%o1 + RET_OFFSET], %o7

    ld          [%o1 + Y_OFFSET], %o2
    ld          [%o1 + G0_OFFSET], %g0
    ld          [%o1 + G1_OFFSET], %g1
    ld          [%o1 + G2_OFFSET], %g2
    ld          [%o1 + G3_OFFSET], %g3
    ld          [%o1 + G4_OFFSET], %g0
    ld          [%o1 + G5_OFFSET], %g1
    ld          [%o1 + G6_OFFSET], %g2
    ld          [%o1 + G7_OFFSET], %g3

    retl
    nop
```

图 D.3　内部函数实现 II

我们将在这里证明引理6.2的正确性。

证明　我们根据定义6.5将目标展开可知，我们需要证明：对于任意的 m，m_T，m_l，m_s，$\overline{v}$，R，F，$\mathcal{K}$，σ，如果下述成立：

1. $\mathsf{tyms}(\overline{v}, \mathrm{nil})$，$[\varphi]_{\mathrm{dom}(T)}$；
2. $m_T \uplus m_l \uplus m_s \uplus \{\mathsf{TaskCur} \rightsquigarrow (t, 0)\} \subseteq m$；
3. $T \sim_t m_T$，$\mathsf{caller\_explore}(R, F, \overline{v}, m_l)$，$\mathsf{initM}(\varphi|_{\mathbb{S}}, \mathbb{S}, \Sigma, m_s)$；
4. $\mu = (\varphi|_{\mathbb{S}}, \mathbb{S}, \mathrm{dom}(m_l) \cup \mathsf{DomCtx}(t))$，$\sigma = (m, (R, F), \mathrm{nil})$；

则：要么 $(T, t, \mathcal{K}, \Sigma)\, \phi \xrightarrow{(\mathtt{switch}, \overline{v})} \mathbf{abort}$；

或者 $(C_{\mathrm{as}}, \sigma, (\mathtt{f}_{\mathrm{sw}}, 0), (\mathtt{f}_{\mathrm{sw}}, 4)) \leqslant^0_{(\mu,\sigma)} ((\mathtt{switch}, \overline{v}), (T, t, \mathcal{K}, \Sigma))$。

已知要么 $(T, t, \mathcal{K}, \Sigma)\, \phi \xrightarrow{(\mathtt{switch}, \overline{v})} \mathbf{abort}$；或者 $\neg(T, t, \mathcal{K}, \Sigma)\, \phi \xrightarrow{(\mathtt{switch}, \overline{v})} \mathbf{abort}$。我们分情况讨论：

（1）如果 $(T, t, \mathcal{K}, \Sigma)\, \phi \xrightarrow{(\mathtt{switch}, \overline{v})} \mathbf{abort}$ 成立，则我们的证明目标成立。

（2）如果 $\neg(T, t, \mathcal{K}, \Sigma)\, \phi \xrightarrow{(\mathtt{switch}, \overline{v})} \mathbf{abort}$ 成立，则我们知道存在 T'，t'，$\mathcal{K}$，Σ，使得：

$$(T, t, \mathcal{K}, \Sigma)\, \phi \xrightarrow{(\mathtt{switch}, \overline{v})} (T', t', \mathcal{K}', \Sigma') \tag{D.1}$$

我们由 $\mathsf{tyms}(\overline{v}, \mathrm{nil})$ 成立可知：$\overline{v} = \mathrm{nil}$。我们根据 switch 原语的语义展开可知下述成立：

$$[\![\mathsf{TaskNew}]\!]_{(\varnothing, \Sigma)} = (t', 0) \tag{D.2}$$

$$T(t') = \mathcal{K}',\ T' = T\{t \rightsquigarrow \mathcal{K}\},\ t \neq t' \tag{D.3}$$

$$\Sigma = \Sigma' \tag{D.4}$$

我们由 $T \sim_t m_T$ 以及 (6.3) 和 (6.4) 可知，线程池 T 中至少存在两个线程：当前线程 t 和需要被调度到的线程 t'，而内存 m_T 中包含了当前线程 TCB 中用于保存当前线程 t 上下文的内存以及保存了 t' 运行时上下文的内存空间，则我们可以证明存在 Q'，p_r（p_r 描述我们不关心的内存）使得下述成立：

$$\begin{aligned}(m, (R, F), \mathrm{nil}) \vDash\ & \mathsf{Env}(R, F) * \mathsf{context}(t, \_) * \mathsf{StkLoc}(R, F) * \\ & \mathsf{NoCurT}(t', Q') * \mathsf{TaskCur} \mapsto (t, 0) * \mathsf{TaskNew} \mapsto (t', 0) * p_r\end{aligned} \tag{D.5}$$

其中，我们用断言 $\mathsf{Env}(R, F)$ 描述当前寄存器状态，用断言 $\mathsf{context}(t, \_)$ 描述当前线程 t 任务控制块中用于保存该线程运行时上下文的内存空间；断言 $\mathsf{StkLoc}(R, F)$ 用于描述调用者暴露给被调用方法的栈空间，即 caller_explore 关系描述的内存 m_l；断言 $\mathsf{NoCurT}(t', Q')$ 用于描述保存了线程 t' 运行时上下文 Q' 的内存空间 (此

处断言和第5.2节中同名的断言的定义不同，此处断言的定义参见图D.5)。

下面我们需要证明：$(C_{\text{as}}, \sigma, (\mathtt{f}_{\text{sw}}, 0), (\mathtt{f}_{\text{sw}}, 4)) \leqslant^{0}_{(\mu,\sigma)} ((\mathtt{switch}, \text{nil}), (T, t, \mathcal{K}, \Sigma))$ 成立（$\sigma = (m, (R, F), \text{nil})$），定义6.6中的模拟关系要求底层程序执行的每一步都满足保证 G_{μ}，即底层程序的执行只会修改自己的局部内存（局部内存包括：调用者暴露给该方法的栈空间和当前线程 TCB 保存运行时上下文的内存），共享内存以及记录当前线程编号的 TaskCur 地址空间。我们定义保证条件 G_{sw} 刻画该保证：

$$G_{\text{sw}} \overset{\text{def}}{=} \left\{ \begin{array}{l} \text{context}(t, \_) * \text{StkLoc}(R, F) * \text{TaskCur} \mapsto \_ \\ \text{context}(t, \_) * \text{StkLoc}(R, F) * \text{TaskCur} \mapsto \_ \end{array} \right\} * \text{Id} \tag{D.6}$$

这里 G_{sw} 要求底层实现只修改当前线程的局部内存空间（$\text{context}(t, \_) * \text{StkLoc}(R, F)$ 所描述的内存空间）；以及地址 TaskCur 所对应的内存单元，该保证比 G_{μ} 强，我们可以证明 (6.8) 成立，因为 G_{μ} 允许修改共享内存，而 G_{sw} 保证共享内存不变。说明如果底层程序执行每一步满足 G_{sw}，那一定也满足 G_{μ}：

$$G_{\text{sw}} \Longrightarrow G_{\mu} \tag{D.7}$$

图D.4中给出线程上下文切换程序从起始状态开始执行，一些重要程序节点的程序状态的断言描述（每段代码修改的部分用红色标记）：

- 我们首先执行程序第 1~5 行，该代码段将当前寄存器文件中的 global，$\mathtt{r}_{15}$ 和 %sp 寄存器保存到当前线程 t 的 TCB 中，该过程只修改断言 $\text{context}(t, \_)$ 描述的内存空间，满足保证 G_{sw}。
- 执行程序的第 6~13 行，该代码段将当前窗口的 local、in 寄存器，以及其它临时保存了运行时上下文的寄存器窗口的内容保存到内存栈空间（即调用者暴露给上下文切换程序的栈空间）。该过程修改 $\text{StkLoc}(R, F)$ 描述的内存空间，满足保证 G_{sw}。
- 再执行程序的第 14~17 行，该代码段修改当前任务为 t'，只修改地址 TaskCur 对应内存单元，满足保证 G_{sw}。
- 执行代码段 18~22 行，该段代码恢复新任务 t' 保存在内存中的当前窗口的内容（包括：global，$\mathtt{r}_{15}$，%sp，local 和 in 寄存器）。此部分代码执行只从断言 $\text{NoCurT}(t', Q')$ 描述内存中读数据，不修改内存，满足保证 G_{sw}。

最后执行第 23，24 行代码，底层原语实现函数执行结束，假设执行 retl 指令前底层程序状态为 σ'。此时由模拟关系，我们首先需要证明高层程序可以走一步，即：

$$\exists\, \Delta'.\ ((\mathtt{switch}, \text{nil}), (T, t, \mathcal{K}, \Sigma)) \longmapsto \Delta' \tag{g1}$$

根据 (6.3)~(6.4) 可知，存在 $\Delta' = (T\{t \rightsquigarrow \mathcal{K}\}, t', \mathcal{K}', \Sigma)$ 是的 (g1) 成立。另外，我们还需要证明执行底层执行原语实现对程序状态的修改和高层执行抽象原语对程序状态的修改之间满足关系 **wfctx**，因为这里发生了线程切换，所有由图6.8 中 **wfctx** 的定义，我们需要证明下述成立（假设执行完 23，24 行的 `retl` 和 `nop` 指令后程序计数器为 $\mathtt{pc}'$ 和 $\mathtt{npc}'$）：

$$\Sigma.M \overset{\mathrm{dom}(\Sigma.M)-\mathbb{S}}{=\!=} \Sigma.M \tag{g2}$$

$$\mathsf{Inv}(\varphi|_{\mathbb{S}}, \Sigma.M, \sigma'.m) \tag{g3}$$

$$\mathsf{closed}(\mathbb{S}, \Sigma.M) \tag{g4}$$

$$\mathbf{ctxsw}(\mu, (\sigma, \Delta), (\sigma', \Delta')) \tag{g5}$$

$$\mathtt{pc}' = \sigma'.Q.R(\mathtt{r}_{15})+8, \mathtt{npc}' = \sigma'.Q.R(\mathtt{r}_{15})+12 \tag{g6}$$

其中目标 (g2) 由 "$\overset{\mathrm{dom}(\Sigma.M)-\mathbb{S}}{=\!=}$" 的定义（定义在图C.1中）可以显然成立；(g3) (g4) 可根据条件中的 $\mathsf{initM}(\varphi|_{\mathbb{S}}, \mathbb{S}, \Sigma, m_s)$，以及底层程序的执行没有修改其共享内存 m_s 得到；(g6) 可由 `retl` 和 `nop` 的语义得证；我们下面证明 (g5) 成立，由图6.8 中 **ctxsw** 的定义展开，我们需要证明下述成立：

$$\sigma'.m \rightarrowtail_{t'} \sigma'.Q \tag{g5.1}$$

$$(R, F) \rightarrowtail_{t} \sigma'.m \tag{g5.2}$$

$$\sigma'.m(\mathsf{TaskCur}) = (t', 0) \tag{g5.3}$$

$$t \neq t',\ T\{t \rightsquigarrow \mathcal{K}\} = T\{t \rightsquigarrow \mathcal{K}\},\ T(t') = \mathcal{K}' \tag{g5.4}$$

$$\sigma'.D = \mathrm{nil} \tag{g5.5}$$

目标 (g5.4) 可由 (6.3)~(6.4) 得证。因为该任务上下文切换程序执行过程中不修改特殊寄存器，所以如果程序执行前延时缓存为空，那么程序执行结束后延时缓存也为空，所以目标 (g5.5) 成立。

由程序结束时断言 $\mathsf{Env}(Q'') * (\mathsf{NoCurT}(t', Q') \wedge Q'' \propto Q')$ 成立，我们知道新线程 t' 保存在内存中的运行时上下文已经被恢复，即目标 (g5.1) 成立。

由程序结束时断言 $\mathsf{context}(t, (R([\mathsf{global}]), R(\%\mathtt{sp}), R(\mathtt{r}_{15}))) * \mathsf{Stack}(R, F)$ 成立，可知线程 t 的运行时上下文已经保存到内存中，即我们有 $(R, F) \rightarrowtail_{t} \sigma'.m$，目标 (g5.2) 成立。

目标 (g5.3) 可根据程序结束后断言 $\mathsf{TaskCur} \mapsto (t', 0)$ 成立得证。

□

f_{sw} :

$\{\ \mathsf{Env}(R,F) * \mathsf{context}(t,\_) * \mathsf{StkLoc}(R,F) * \mathsf{NoCurT}(t',Q') * \mathsf{TaskCur} \mapsto (t,0) * \mathsf{TaskNew} \mapsto (t',0)\ \}$

```
1     set     TaskCur, %o0
2     ld      [%o0], %o1
3     add     %o1, CONTEXT_OFFSET, %o1
4     call    reg_save
5     nop
```

$\{\ \mathsf{Env}(R\{\%o_0 \rightsquigarrow \mathsf{TaskCur}\}\{\%o_1 \rightsquigarrow (t, \mathtt{CONTEXT\_OFFSET})\}, F) * \mathsf{context}(t,(R([\mathsf{global}]), R(\%\mathtt{sp}), R(\mathtt{r}_{15}))) * \mathsf{StkLoc}(R,F) * \mathsf{NoCurT}(t',Q') * \mathsf{TaskCur} \mapsto (t,0) * \mathsf{TaskNew} \mapsto (t',0)\ \}$

save_windows :

```
6     call    window_save
7     nop
8     cmp     (1 << (cwp + 1) mod N), wim
9     bz      switch_new_task
10    nop
11    restore
12    jmp     window_save
13    nop
```

$\{\ \exists Q''.\, \mathsf{Env}(Q'') * \mathsf{context}(t,(R([\mathsf{global}]), R(\%\mathtt{sp}), R(\mathtt{r}_{15}))) * \mathsf{Stack}(R,F) * \mathsf{NoCurT}(t',Q') * \mathsf{TaskCur} \mapsto (t,0) * \mathsf{TaskNew} \mapsto (t',0)\ \}$

switch_new_task :

```
14    set     TaskCur, %o0
15    set     TaskNew, %o1
16    ld      [%o1], %o1
17    st      %o1, [%o0]
```

$\{\ \exists Q''.\, \mathsf{Env}(Q'') * \mathsf{context}(t,(R([\mathsf{global}]), R(\%\mathtt{sp}), R(\mathtt{r}_{15}))) * \mathsf{Stack}(R,F) * \mathsf{NoCurT}(t',Q') * \mathsf{TaskCur} \mapsto (t',0) * \mathsf{TaskNew} \mapsto (t',0)\ \}$

```
18    add     %o1, CONTEXT_OFFSET, %o1
19    call    reg_restore
20    nop
21    call    window_restore
22    nop
```

$\{\ \exists Q''.\, \mathsf{Env}(Q'') * \mathsf{context}(t,(R([\mathsf{global}]), R(\%\mathtt{sp}), R(\mathtt{r}_{15}))) * \mathsf{Stack}(R,F) * (\mathsf{NoCurT}(t',Q') \wedge Q'' \propto Q') * \mathsf{TaskCur} \mapsto (t',0) * \mathsf{TaskNew} \mapsto (t',0)\ \}$

```
23    retl
24    nop
```

图 D.4　线程上下文切换程序主要程序节点程序状态描述

$$\mathsf{Env}(R,F) \overset{\text{def}}{=} (\mathsf{global}\mapsto R([\mathsf{global}]) * \mathsf{out}\mapsto R([\mathsf{out}]) * \mathsf{local}\mapsto R([\mathsf{local}]) * \mathsf{in}\mapsto R([\mathsf{in}]) * \mathtt{wim}\mapsto R(\mathtt{wim}) * \mathtt{cwp}\mapsto (\!| R(\mathtt{cwp}), F |\!)) \wedge \mathsf{wdwp}(R,F)$$

$$\mathsf{context}(t,(\overline{v},v_{\text{sp}},v_{\text{ret}})) \overset{\text{def}}{=} [(t,\mathtt{G0\_OFFSET}),\cdots,(t,\mathtt{G7\_OFFSET})]\mapsto\overline{v} * (t,\mathtt{SP\_OFFSET})\mapsto v_{\text{sp}} * (t,\mathtt{RET\_OFFSET})\mapsto v_{\text{ret}}$$
$$\text{where}\quad [l_0,\cdots,l_{n-1}] \overset{\text{def}}{=} l_0\mapsto\overline{v}[0] * \cdots\ l_{n-1}\mapsto\overline{v}[n-1]$$

$$\mathsf{NoCurT}(t,Q) \overset{\text{def}}{=} (\mathsf{context}(t,(R([\mathsf{global}]),R(\%\mathtt{sp}),R(\mathtt{r}_{15}))) * \mathsf{StackFrame}(b,R([\mathsf{local}]),R([\mathsf{in}]))) \wedge \mathsf{wdwp}(R)$$
$$\text{where}\quad Q=(R,F),\ R(\%\mathtt{sp})=(b,0)$$

$$\mathsf{Stack}(R,F) \overset{\text{def}}{=} \mathsf{Stack}'(b,w_{id},n,R([\mathsf{local}])::R([\mathsf{in}])::F) \wedge \mathsf{wdwp}(R)$$
$$\text{where}\quad R(\mathtt{cwp})=w_{id},\ R(\mathtt{wim})=2^n,\ R(\%\mathtt{sp})=(b,0)$$

$$\mathsf{StkLoc}(R,F) \overset{\text{def}}{=} \mathsf{StkLoc}'(b,w_{id},n,R([\mathsf{local}])::R([\mathsf{in}])::F) \wedge \mathsf{wdwp}(R)$$
$$\text{where}\quad R(\mathtt{cwp})=w_{id},\ R(\mathtt{wim})=2^n,\ R(\%\mathtt{sp})=(b,0)$$

$$\mathsf{Stack}'(b,w_{id},n,F) \overset{\text{def}}{=} \begin{cases} \mathsf{StackFrame}(b,\text{fm}_1,\text{fm}_2) * \mathsf{Stack}'(b,(w_{id}+1)\bmod N,n,F') & \textbf{if } \text{fm}_2[6]=(b',0),\ w_{id}\neq n,\ F=\text{fm}_1::\text{fm}_2::F' \\ \mathsf{emp} & \textbf{if } w_{id}=n \\ \mathsf{false} & \textbf{otherwise} \end{cases}$$

$$\mathsf{StkLoc}'(b,w_{id},n,F) \overset{\text{def}}{=} \begin{cases} \mathsf{StackFrame}(b,\_,\_) * \mathsf{StkLoc}'(b,(w_{id}+1)\bmod N,n,F') & \textbf{if } \text{fm}_2[6]=(b',0),\ w_{id}\neq n,\ F=\text{fm}_1::\text{fm}_2::F' \\ \mathsf{emp} & \textbf{if } w_{id}=n \\ \mathsf{false} & \textbf{otherwise} \end{cases}$$

$$\mathsf{StackFrame}(b,\text{fm}_1,\text{fm}_2) \overset{\text{def}}{=} (b,0)\mapsto\text{fm}_1[0] * \cdots * (b,7)\mapsto\text{fm}_1[7] * (b,8)\mapsto\text{fm}_2[0] * \cdots * (b,15)\mapsto\text{fm}_2[7]$$

$$\mathsf{wdwp}(R) \overset{\text{def}}{=} \exists\, w_{id},n.\ R(\mathtt{cwp})=w_{id}\wedge R(\mathtt{wim})=2^n\wedge n\neq w_{id} \wedge 0\leqslant n,w_{id}\leqslant N-1\wedge|F|=2\times N-3$$

$$(\sigma,\sigma')\vDash\mathsf{Id} \overset{\text{def}}{=} \sigma=\sigma' \qquad (\sigma,\sigma')\vDash\left\{\begin{array}{c}p\\p'\end{array}\right\} \overset{\text{def}}{=} \sigma\vDash p\wedge\sigma\vDash p'$$

$$g\Longrightarrow g' \overset{\text{def}}{=} \forall\,\sigma,\sigma'.(\sigma,\sigma')\vDash g\Longrightarrow(\sigma,\sigma')\vDash g'$$

图 D.5　描述程序状态时一些辅助断言定义

致　谢

我在科大度过了愉快而充实的三年，首先我要感谢我的导师冯新宇老师，是他认真、严谨、细致、负责的工作态度和崇高的学术素养和追求在一直激励着我不断努力，不断提高，不断改进。他总是教导我做事要态度认真，精益求精。在他的督促和教导下，我不仅学会了很多做人的道理，也改正了以前做事拖沓、不求甚解的坏毛病，端正了科研态度，提高了科研水平。

感谢苏州 407 软件安全实现的小伙伴们：蒋瀚如、张啸然、李朝晖、何春晖、肖思扬以及其他已经毕业了的师兄们，感谢你们在我“嘴馋”的时候陪我一起“吃吃喝喝”；感谢你们在我无聊的时候陪我打游戏；更感谢你们在我遇到困难和问题的时候给我帮助和支持。

感谢我在合肥的室友：陈冲、陈浩和程梦卓同学。虽然研一之后我就离开合肥去了苏州，但在短短一年的相处中，感谢你们在学习和生活中给我的帮助，怀念和你们一起上课、吃饭、追剧和吐槽的时光。

感谢母校对我的培养，感谢一路上给予我鼓励和支持的家人、同学和朋友，愿母校越来越好，各位前程似锦。

在读期间发表的学术论文与取得的研究成果

已发表论文

1. **Junpeng Zha**, Xinyu Feng, Lei Qiao. Modular Verification of SPARCv8 Code. In: Ryu S. (eds) Programming Languages and Systems. APLAS 2018. Lecture Notes in Computer Science, vol 11275. Springer, Cham. （CCF C 类会议）