Modular Verification of SPARCv8 Code

Junpeng Zha[1] and Xinyu Feng[2]

[1] School of Computer Science and Technology
University of Science and Technology of China
jpzha@mail.ustc.edu.cn
[2] State Key Laboratory for Novel Software Technology
Nanjing University
xyfeng@nju.edu.cn

Abstract. Inline assembly code is common in system software to interact with the underlying hardware platforms. Safety and correctness of the assembly code is crucial to guarantee the safety of the whole system. In this paper we propose a practical Hoare-style program logic for verifying SPARC assembly code. The logic supports modular reasoning about the main features of SPARCv8 ISA, including delayed control transfers, delayed writes to special registers, and register windows. We have successfully applied it to verify a context switch module in a realistic embedded OS kernel. All of the formalization and proofs have been mechanized in Coq.

1 Introduction

Operating system kernel is the most foundational software of computer systems. Because of a number of interactions with hardware existence, many important components in OS kernel are built by low-level assembly code, like context switch, trap enable/disable and so on. However, current OS kernel verification [**?**, **?**, **?**] works either skip or do not present a program logic for the assembly code verification. And the existing efforts for assembly code verification like PCC [**?**] and TAL [**?**, **?**] usually care about addressing the safety properties of programs, instead of their functional behavior. In this paper, we present a novel and practical Hoare-style program logic for SPARCv8 code [**?**]. Our logic is designed on a realistically modelled machine code and supports modular, local, function call/return, and main features of SPARCv8 code verification.

SPARC (Scalable Processor ARChitecture) is a CPU instruction set architecture with high-performance and great flexibility [**?**]. It has been widely used in various processors for workstations, embedded system, and space mission. Designing a practical Hoare-style program logic for SPARCv8 code isn't an easy work because we will meet many challenges as following :

– SPARCv8 has some special features which are different with other assembly language. (1) SPARCv8 uses the "register windows" mechanism to reduce the cost of context saving and restoring, instead of operating on memory directly.

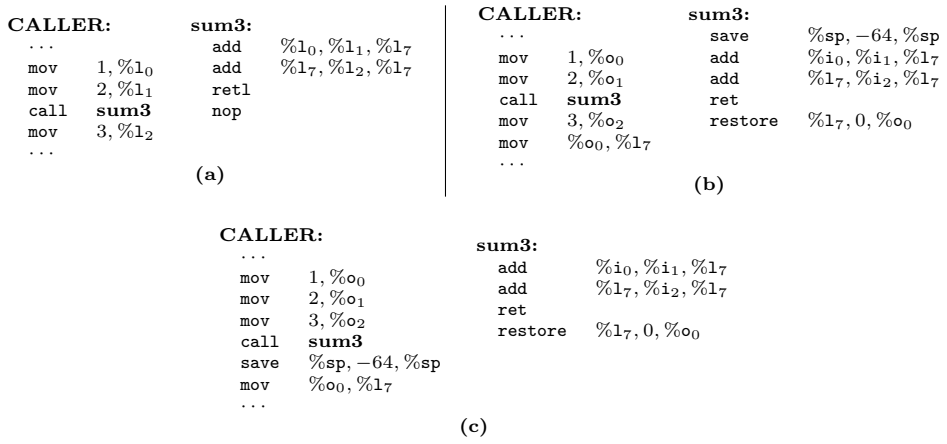

Fig. 1. Three Call Conventions in SPARCv8

(2) There are some delayed control-transfer instructions in SPARCv8, which means that the executing of these instruction will not cause control transfer immediately, but delayed one cycle. (3) Writing special register operation `wr` in SPARCv8 is "delayed-write". The write operation will update the value of target special register after the execution of some instructions following `wr`. We have to consider those features in program logic designing.

– Stack based operations like call/return are difficult to verify in low-level assembly code, because of a lack of structure. A function in low-level assembly code may be composed by multiple code blocks. It proposes some challenges to judge when the verified function's postcondition will hold especially current function may call another one. It answers why most previous works on assembly code verification [**?**, **?**] view call/return as the first-class code point. In SPARCv8, the delayed control-transfer mechanism makes the call convention richer and more flexible than other assembly code, The programs in Fig. 1 are three common function call conventions in SPARCv8. And how to handle them in a uniform method provides a new challenge for us.
– Lacking structure of assembly code also present another problem for establishing soundness. Related work like CAP [**?**] and SCAP [**?**] establish the soundness following a syntactic approach of proving type soundness which says that a verified program will never abort. Giving a semantic approach to establish the soundness will be difficult because of the requirement to link multiple code blocks together.

In this paper, we propose a novel and practical Hoare-style program logic for a realistically modelled machine code, SPARCv8. We not only solve the challenges above, but also show its application in verifying a key module in OS kernel, context switch. To summarize, our work makes the following contributions :

– We present a program logic that has been designed to fit on accurate models of SPARCv8 language, so that we can verify the SPARCv8 program modularly. We does not only care about the safety property of the SPARCv8

codes in our work, but also their functional behavior. Our logic can support main features of SPARCv8 code, including delayed control-transfer, register windows, and delayed-write.
- Our logic supports the function call verification. The inference rule designed for instruction `call` is general, and require as little restrictions as possible for caller and callee. The inference rule for instruction `call` can verify these three most common call conventions shown in Fig. 1 in a uniform way.
- We establish the soundness of our logic by a semantic approach in our work and prove that our logic is sound by Coq proof assistant [**?**]. We develop a semantics for each inference rule in our logic, and make an important conclusion that a function implemented in SPARCv8 will satisfy its specification if each block code composing it can be checked by our logic.
- We design an assertion language in our work. Our assertion language does not only provide a form to describe registers, memory values in program state, but also define a new assertion [n]p which means the predicate p will hold after n cycles. We will introduce that there is a delay list in SPARCv8 code program state in order to record the delayed write operation. This assertion will let us do not have to expose the detail of the state of delay list and the operation of **exe_delay**, which will do in each cycle, in our predicate.
- Context switch is an important component in OS kernel. Most of context switch programs are implemented in assembly code because of the requirement to assess registers and stack. We prove the correctness of the context switch module in a realistic embedded OS kernel [1], and achieve good ratio (around 16:1) of Coq proof scripts to verified SPARCv8 code. This work presents that our logic is useful and can check realistic SPARCv8 code easily and conveniently.

In the rest of paper, we first introduce the background of our work and give an overview of our approach. We present the program model and operational semantics of SPARCv8 program in Sec. 3. Then we define the program logic in Sec. 4, including the inference rules and soundness proof. Sec. 5 gives an actual application of our logic to verify a realistic context switch module in an embedded OS kernel. Finally, we discuss more related work and conclusion in Sec. 6.

2 Background and Our Approach

In this section, we give an overview of SPARCv8 instruction set and some important works [**?**, **?**] on assembly code verification (Yu *et al.* 2003, Feng *et al.* 2006). Then we will show our approach to design the program logic for SPARCv8 code.

[1] We can't publish the source code verified here, because of a confidentiality agreement between OS kernel provider and us

```
CALLER :
    ...
1   mov  1, %o0
2   call  ChangeY
3   save  %sp, −64, %sp
4   mov  %o0, %l0
    ...

ChangeY :
5   rd  Y, %l0
6   wr  %i0, 0, Y
7   nop
8   nop
9   nop
10  restore  %l0, 0, %o0
11  retl
12  nop
```

Fig. 2. An Example for SPARC Code

2.1 An Overview of SPARCv8

SPARC is a RISC instruction set architecture (ISA) originally developed by Sun Microsystems. It provides exceptionally high execution rates and low power consumption and has a range of applications, including scientific/engineer, programming, real-time, and commercial [**?**]. Here list some principle features that we have to consider in our logic designing for SPARCv8 :

- *Register Windows.* SPARCv8 uses register windows and window rotation mechanism to avoid saving context in stack directly and achieve high performance and a significant reduction in context management.
- *Delayed Control-Transfer.* SPARCv8 has two program counters pc and npc. Control-transfer instructions in SPARCv8 will change npc instead of pc to target point and cause the control transfer occurring delayed one cycle.
- *Delayed-write.* The write special register instruction `wr` is delayed-write instruction. They may take until completion of some instructions following `wr` to consummate their write operation.

We use a simple example (see Fig. 2) to interpret these three features. The following function **CALLER** calls the function **ChangeY** which updates the value of the special register Y and returns its original value.

The function **ChangeY** requires one input parameter as the new value for special register Y. The **CALLER** calls function **ChangeY** at line 2, and pc and npc point to line 2 and 3 at this moment. The instruction call change the value of pc to npc and let npc points to **ChangeY** at line 5 which means the control-flow will not transfer to **ChangeY** in the current cycle but in the next cycle after the instruction following the call completion. We call it "delayed control-transfer".

SPARCv8 program won't save current context when occurring function call. It use instruction `save` (at line 3) to save the current context and `restore` (at line 10) to restore the saved context. 32 general registers have been split into four logic parts as global ($\mathtt{r}_0 \sim \mathtt{r}_7$), out ($\mathtt{r}_8 \sim \mathtt{r}_{15}$), local ($\mathtt{r}_{16} \sim \mathtt{r}_{23}$) and in ($\mathtt{r}_{24} \sim \mathtt{r}_{31}$) registers. Out, local and in registers compose the current register window and only local registers are private resource for current context. In and out registers are shared with adjacent register windows for parameters passing. To achieve more efficiency, the instruction `save` will not save the context to stack

in memory, but do a window rotation to next window instead. After window rotation caused by `save` instruction, original parts of local and in registers will not in current window, and original out registers still in current window as in registers for parameters passing between two adjacent window. The operation of `restore` is inverse.

At line 6, the instruction `wr` hope to update the value of special register Y by the result of "$\%\mathtt{i}_0$ **xor** 0 ". The write operation will not make an effect to program state until the completion of the third instruction at line 9 following the `wr` and this feature is "delayed-write".

2.2 SCAP

SCAP [**?**] (Feng *et al.* 2007) is an extension of original assembly code verification work, CAP [**?**]. In CAP framework, programs in code heap are viewed as separately basic code blocks, and each basic code block can be checked independently. However, CAP only makes sure that the program verified will not get stuck and views function call/return just as first-class code pointers.

Stack operations like function call/return and exception handlers are very hard to reason about, because the lack of structure of assembly code makes us difficult to track their restricted scope. SCAP supports stack-based controls by capturing the invariant which can be viewed as a chain of valid return pointers at each program point. The local guarantee g in SCAP can be regarded as a specification to describe the function of an assembly code function. The CALL and RET rules in SCAP are shown below :

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

$$\frac{\forall S.\ p\ S \rightarrow g\ S\ S}{\Psi \vdash \{(p, g)\}\ \mathbf{jr}\ \$\mathrm{ra}}\ \textbf{(RET)}$$

The instruction **jal** change the control-flow to function f, save the return point in register ra, and will return to point $\mathtt{f}_{ret}$ after function f execution finish. The **CALL** rule makes sure that (1) the current function can resume its execution after f return, (2) there is still a specification g'' satisfying the resume execution of current function, and (3) function f will reinstate the return code point saved in $ra when it returns. The most interesting thing is that we do not need any knowledge about the return address $ra in precondition p. Because the invariant mentioned before makes sure the return address is a valid one.

2.3 Our Approach

As in CAP and SCAP, we also split programs in code heap into separated code blocks, so that we can check each block independently, and use the inspiration

that SCAP provides to certify function call/return in SPARCv8 code. Instead of using local guarantee g, we use precondition p and postcondition q as the function specification, which is similar with traditional hoare logic. The instruction `call` is viewed as the point that control-flow transfers to the specific function and `retl` instruction is treated as the end of current function.

We do the following works to make our logic support the features of SPARCv8 code. (1) The instruction following control-transfer instructions (excluding call and conditional control-transfer) is regarded as the end of code block, in order to handle delayed control-transfer and achieve modularity in verification. (2) We define a new assertion [n]p which means the condition p will hold after n cycles to solve delayed-write of special registers writing. When meeting `wr` instruction like "`wr` $\mathtt{r}_1$, $\mathtt{r}_2$, rsp", the state of special register rsp will be the form of "[3]rsp $\mapsto$ a" (a is the value of "$\mathtt{r}_1$ `xor` $\mathtt{r}_2$"). The delay time recorded in "[-]" will reduce one after execution of each instruction following `wr`. The assertion [n]p describe a state that we are not certainly knowing currently, but will satisfy assertion p after n cycles.

Discarding the syntactic approach for soundness definition in CAP and SCAP, we present a semantic approach to establish soundness. We will elaborate the semantics of well-formed code heap, well-formed instruction sequence and well-formed instruction in Sec. 4.3. We also prove the below corollary that if each instruction sequence of a function satisfying the soundness definition, then we get that the function can execute safely from a state satisfying p, and will satisfy q at its return point :

$$\frac{\Psi \models \{(p,q)\}\,\mathrm{pc} : C[\mathrm{pc}] \qquad S \models p \qquad \Psi \subseteq \Psi' \qquad \Psi \models C : \Psi'}{\forall n.\ \mathrm{safety}^n(C, S, \mathrm{pc}, \mathrm{pc}+4, q, 0)}$$

The frame rule (with some side conditions elided) which many other assembly code verification framework like CAP and SCAP do not give, as shown below, supports local reasoning of instruction sequence. Here, the assertion p_r describes a part of state that the verification work doesn't care about.

$$\frac{\Psi \vdash \{(p,q)\}\ \mathtt{f} : \mathbb{I}}{\Psi \vdash \{(p * p_r, q * p_r)\}\ \mathtt{f} : \mathbb{I}}$$

3 Modeling

We introduce our program model in this section. We first give a whole recognition of SPARCv8 instruction set to readers by introducing the machine state, then, we describe some details about registers, frame list and delay list following.

3.1 Program Model

The machine model and syntax of SPARCv8 language are defined in Fig. 3. SPARCv8 program can be viewed as a composition of code heap, machine state,

$$
\begin{array}{rrll}
\text{(Prog)} & \mathrm{P} & ::= & (C, S, \mathrm{pc}, \mathrm{npc}) \\
\text{(State)} & S & ::= & (M, Q, D) \\
\text{(Memory)} & M & \in & \text{Address} \rightharpoonup \text{Word} \\
\text{(OpExp)} & \mathtt{o} & ::= & r \mid w \\
\text{(Value)} & w, v, w_{id} & \in & \text{Integer} \\
\text{(Command)} & c & ::= & \mathtt{i} \mid \mathtt{call\ f} \mid \mathtt{jmpl\ a\ r}_d \mid \mathtt{retl} \mid \mathtt{be\ f} \\
\text{(SimpIns)} & \mathtt{i} & ::= & \mathtt{ld\ a\ r}_d \mid \mathtt{st\ r}_s\ \mathtt{a} \mid \mathtt{nop} \mid \mathtt{save\ r}_s\ \mathtt{o\ r}_d \mid \mathtt{restore\ r}_s\ \mathtt{o\ r}_d \mid \\
 & & & \mathtt{add\ r}_s\ \mathtt{o\ r}_d \mid \mathtt{rd\ sr\ r}_d \mid \mathtt{wr\ r}_s\ \mathtt{o\ sr} \mid \ldots \\
\text{(InstrSeq)} & \mathbb{I} & ::= & \mathtt{i};\ \mathbb{I} \mid \mathtt{jmpl\ a\ r}_d;\ \mathtt{i} \mid \mathtt{call\ f};\ \mathtt{i}; \mathbb{I} \mid \mathtt{retl};\ \mathtt{i} \mid \mathtt{be\ f};\ \mathtt{i};\ \mathbb{I}
\end{array}
\qquad
\begin{array}{rrll}
\text{(CodeHeap)} & C & ::= & \{\mathtt{f} \rightsquigarrow c\}^* \\
\text{(RState)} & Q & ::= & (R, F) \\
\text{(ProgCount)} & \mathrm{pc},\ \mathrm{npc} & \in & \text{Word} \\
\text{(AddrExp)} & \mathtt{a} & ::= & \mathtt{o} \mid r + \mathtt{o}
\end{array}
$$

Fig. 3. Machine States and Language for SPARCv8 Code

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

Fig. 4. Register File, Frame List and DelayList

pc and npc. Code heap C is a partial function from labels `f` to commands c. Label is a 32-bit integer as an address where the commands are saved. Commands in SPARCv8 can be classified into two categories. The first is simple instruction `i` whose execution will not cause control-transfer. The other is the control-transfer instruction like `call`, `jmpl` and so on.

Program state S consists of three parts : memory state M, register state Q which is a pair of register file R and frame list F, and delay list D. We elaborate R, F, D in Fig. 4. Register file R is a partial map from register name (RegName) to word. RegName includes general registers r, processor state register psr and special registers `sr`. The processor state register psr contains the integer condition codes fields `n`, `z`, `v` and `c`, which can be modified by the arithmetic and logical instructions and used for the judgment of conditional control-transfer, and `cwp` recording the identity of the current window. Register windows and delayed-write are significant features in SPARCv8. We define frame list and delay list in program model, in order to describe these mechanism formally.

Frame List. We define frame list F in order to describe the feature of register windows in SPARCv8. SPARCv8 uses register windows to reduce the time of context management. When there is a requirement to save/restore context, SPARCv8 will do window rotation instead of saving/restoring context into stack directly. The register windows mechanism can be viewed as a data structure of circular finite stack as showing in Fig. 5. 32 general registers are split into four parts as global ($\mathtt{r}_0 \sim \mathtt{r}_7$), out ($\mathtt{r}_8 \sim \mathtt{r}_{15}$), local ($\mathtt{r}_{16} \sim \mathtt{r}_{23}$) and in ($\mathtt{r}_{24} \sim \mathtt{r}_{31}$) registers. Out, local and in registers compose the current register window and only local registers are private resource for current context. In and out registers of a window are shared with its adjacent windows for parameters passing. We introduce the process of window rotation by Fig. 5. In Fig. 5, the number of

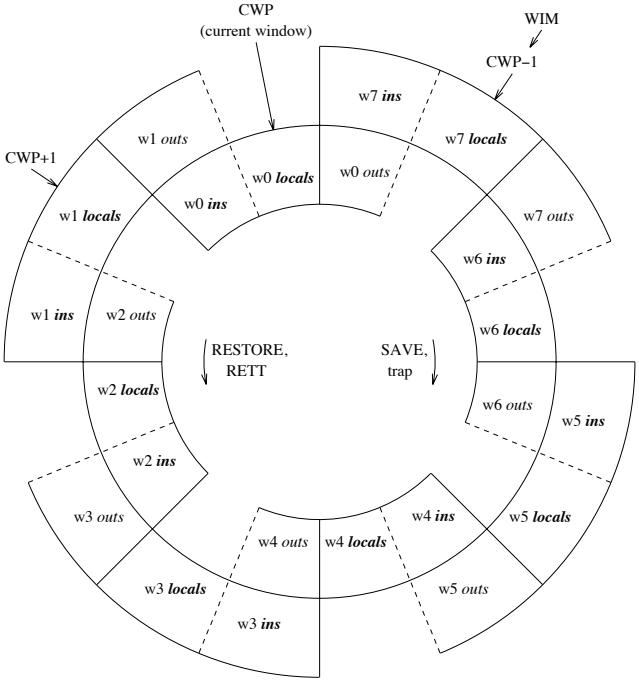

Fig. 5. Register Windows (Figure taken from [**?**])

register windows is eight, and the identities of windows are presented from w_0 to w_7. We can find that each window's in and out registers are shared with its adjacent windows. For example, the in registers of the w_0 is the out registers of the w_1, and the out registers of the w_0 is the in registers of the w_7. The value of `cwp` presents the identity of the current window. Supposing the current window is w_0, the execution of `save` will let window w_7 become the current window. The out registers of w_0 become the in registers of w_7. The local and in registers of window w_0 will not be in the current window, just as being saved.

The execution of instructions `save`, `restore` and `rett` (trap existence instruction) and trap occurring will let the window rotate as shown in Fig. 5. Instruction `save` and trap occurring will change the identity of the current window from w_{id} to $\mathbf{pre\_cwp}(w_{id})$, and the instruction `restore` and `rett` will modify the identity of the current window from w_{id} to $\mathbf{post\_cwp}(w_{id})$. The operation of $\mathbf{pre\_cwp}(w_{id})$ and $\mathbf{post\_cwp}(w_{id})$ can be formally defined as :

$$\mathbf{pre\_cwp}(w_{id}) \stackrel{\triangle}{=} (w_{id} + N - 1)\%N$$
$$\mathbf{post\_cwp}(w_{id}) \stackrel{\triangle}{=} (w_{id} + 1)\%N$$

where N is the number of register windows.

The contents of the current window are presented in general registers by out, local and in registers, and we use frame list F to present the contents of other windows, shown in Fig. 6. The definition of frame and frame list is presented in Fig. 4. A frame is an array that contains 8 words, and a frame list is a list of frames and its length is $2N - 3$ (where N is the number of register windows). Supposing the contents of out, local and in registers are fm_o, fm_l and fm_i and the frame list is F, the register windows can be presented as $\mathrm{fm}_o :: \mathrm{fm}_l :: \mathrm{fm}_i :: F$ formally.

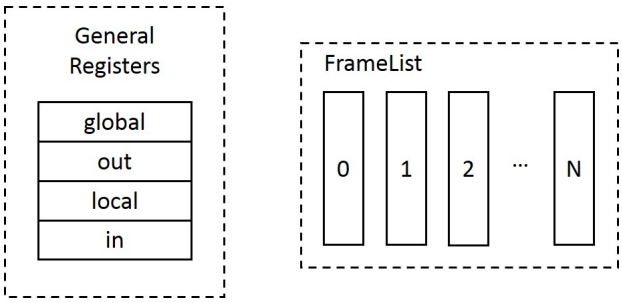

Fig. 6. Using Frame List to Present Register Windows in SPARC

Delay List. Delay list D, which is a list of delay item d, is used to save delayed-write operations caused by instruction `wr`. Because the execution of instruction `wr` will not update the target special register immediately, we put the operation of writing it as a delay item into the delay list. DelayItem d is a triple consisting of the remaining cycles c to be delayed, the target special register `sr` and the value w to be written.

Instruction sequence $\mathbb{I}$ is a sequence of commands. We restrict that a delayed control-transfer instruction must following a simple instruction `i` here, because the actual control-transfer occurring after `i` execution. And `jmpl` following a simple instruction `i`, and return command `retl` following a simple instruction `i` are viewed as the end of instruction sequence in our work. We define $C[\mathtt{f}]$ to extract an instruction sequence starting from `f` in C below.

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

The code block shown below are all satisfying our requirements for legal instruction sequences. Defining instruction sequence in this form let us achieve

modularity in our verification work, so that we can check each instruction sequence independently.

`mov`	1, %$\mathtt{o}_0$	`save`	%`sp`, −64, %`sp`
`call`	ChangeY	`add`	%$\mathtt{i}_0$, %$\mathtt{i}_1$, %$\mathtt{l}_7$
`save`	%`sp`, −64, %`sp`	`add`	%$\mathtt{l}_7$, %$\mathtt{i}_2$, %$\mathtt{l}_7$
`mov`	%$\mathtt{o}_0$, %$\mathtt{l}_0$	`restore`	%$\mathtt{g}_0$, %$\mathtt{g}_0$, %$\mathtt{g}_0$
`jmpl`	%$\mathtt{g}_3$, %$\mathtt{g}_0$	`retl`	
`nop`		`nop`	

3.2 Operational Semantics

We define the operation semantics of SPARCv8 with multiply layers. We just select a part of transition rules in Fig. 7 to give readers a whole recognition for SPARCv8 program execution because of the space limitation. As shown in Fig. 7, we define the operational semantics with four layers. As for the first layer Program Transition, we can see that a step of SPARCv8 program transition can be split into two steps. First, the operation **exe_delay** check each element in delay list, remove the delay items whose delay cycles are 0 and write the value recorded in them to specific special register. Second, the instruction that pc points executes. We define **exe_delay** simply as following.

$$\mathbf{exe\_delay}(R, D) \triangleq \begin{cases} (R, D) & D = \text{nil} \\ (R'\{\mathtt{sr} \rightsquigarrow w\}, D'') & D = (0, \mathtt{sr}, w) :: D', \\ & (R', D'') = \mathbf{exe\_delay}(R, D') \\ (R', (n-1, \mathtt{sr}, w) :: D'') & D = (n, \mathtt{sr}, w) :: D', n > 0, \\ & (R', D'') = \mathbf{exe\_delay}(R, D') \end{cases}$$

The second layer Control Transfer Instruction Transition describes the change of pc and npc for each SPARCv8 instruction execution. Transition rules for control-transfer instructions (e.g., `jmpl`, `call`, `retl`) are defined in this layer. We can see that delayed control-transfer instructions like `jmpl` will change npc instead of pc to target, set original npc to pc and cause the control transfer delayed one cycle. The instruction `retl` and `ret` are two common instructions in SPARCv8, which are used for function return. The instruction `retl` returns to the address of `f` + 8 (where $R(\mathtt{r}_{15}) = \mathtt{f}$). However, the address used for return for instruction `ret` is saved in $\mathtt{r}_{31}$. The reason can be achieved obviously by the three common call conventions we enumerate in Fig. 1. We can see that `retl` and `ret` are used for different call conventions. The instruction `ret` is used for the situation that executes a instruction `save` when occurring function call. The instruction `call` stores its label in register $\mathtt{r}_{15}$, and instruction `save` will let the original $\mathtt{r}_{15}$ become $\mathtt{r}_{31}$ of the new window by window rotation.

Instruction `save`, `restore` and `wr` are different with the other simple instructions because they may change the state of frame list and delay list. And we present their transition in the third layer. The rule for instruction `wr` tells us that the SPARCv8 set the write operation in delay list instead of updating the value of target special register `sr` immediately. The operation of **set_delay**(sr, w, D)

$$\frac{\mathbf{exe\_delay}(R, D) = (R', D') \quad C \vdash ((M, (R', F), D'), \mathrm{pc}, \mathrm{npc}) \circ\!\!\longrightarrow ((M', (R'', F'), D''), \mathrm{pc}', \mathrm{npc}')}{C \vdash ((M, (R, F), D), \mathrm{pc}, \mathrm{npc}) \longmapsto ((M', (R'', F'), D''), \mathrm{pc}', \mathrm{npc}')}$$

(a) Program Transistion

$$\frac{C(\mathrm{pc}) = \texttt{jmpl a } \mathtt{r}_d \quad [\![\mathtt{a}]\!]_R = \mathtt{f} \quad \mathbf{word\_align}(\mathtt{f}) \quad \mathtt{r}_d \in \mathrm{dom}(R)}{C \vdash ((M, (R, F), D), \mathrm{pc}, \mathrm{npc}) \circ\!\!\longrightarrow ((M, (R\{\mathtt{r}_d \rightsquigarrow \mathrm{pc}\}, F), D), \mathrm{npc}, \mathtt{f})}$$

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

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

$$\frac{C(\mathrm{pc}) = \texttt{ret} \quad R(\mathtt{r}_{31}) = \mathtt{f}}{C \vdash ((M, (R, F), D), \mathrm{pc}, \mathrm{npc}) \circ\!\!\longrightarrow ((M, (R, F), D), \mathrm{npc}, \mathtt{f} + 8)}$$

(b) Control Transfer Instruction Transition

$$\frac{(M, R) \xrightarrow{\mathtt{i}} (M', R')}{(M, (R, F), D) \bullet \xrightarrow{\mathtt{i}} (M', (R', F), D)} \qquad \frac{R(\mathtt{r}_s) = v_1 \quad [\![\mathtt{o}]\!]_R = v_2 \quad w = v_1 \,\mathbf{xor}\, v_2 \quad \mathtt{sr} \in \mathrm{dom}(R) \quad D' = \mathbf{set\_delay}(\mathtt{sr}, w, D)}{(M, (R, F), D) \bullet \xrightarrow{\texttt{wr } \mathtt{r}_s \texttt{ o sr}} (M, (R, F), D')}$$

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

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

(c) Save, Restore and Wr instruction Transition

$$\frac{R(\mathtt{sr}) = v \quad \mathtt{r}_d \in \mathrm{dom}(R)}{(M, R) \xrightarrow{\texttt{rd sr } \mathtt{r}_d} (M, R\{\mathtt{r}_d \rightsquigarrow v\})} \qquad \frac{R(\mathtt{r}_s) = v_1 \quad [\![\mathtt{o}]\!]_R = v_2 \quad \mathtt{r}_d \in \mathrm{dom}(R)}{(M, R) \xrightarrow{\texttt{add } \mathtt{r}_s \texttt{ o } \mathtt{r}_d} (M, R\{\mathtt{r}_d \rightsquigarrow v_1 + v_2\})}$$

$$\frac{[\![\mathtt{a}]\!]_R = w \quad \mathbf{word\_align}(w) \quad M(w) = v \quad \mathtt{r}_d \in \mathrm{dom}(R)}{(M, R) \xrightarrow{\texttt{ld a } \mathtt{r}_d} (M, R\{\mathtt{r}_d \rightsquigarrow v\})}$$

(d) Simple Instruction Transition

Fig. 7. Selected Operational Semantics

is defined as following, where X $(0 \leq X \leq 3)$ is the delay cycle whose specific value is implementation dependently :

$$\mathbf{set\_delay}(\mathtt{sr}, w, D) \stackrel{\triangle}{=} (X, \mathtt{sr}, w) :: D$$

The execution of instruction `save` and `restore` will cause the window rotation, so that will change the state of Frame List F. The operation $\mathbf{dec\_win}(R, F)$ describe that, if previous window is valid, we will do a right rotation for register windows by $\mathbf{right\_win}(R, F)$. As for instruction `restore`, it will change the reg-

$$
\begin{array}{ll}
\textbf{dec\_win}(Q) & \triangleq \begin{cases} \textbf{right\_win}(Q) & \textbf{if win\_valid}(\textbf{pre\_cwp}(R(\texttt{cwp})), R) \\ \bot & \textbf{otherwise} \end{cases} \\
& \quad \textbf{where } Q = (R, F) \\
\textbf{right\_win}(Q) & \triangleq \textbf{let } (F', l) := \textbf{right}(F, \textbf{fetch}(R)) \textbf{ in} \\
& \quad \textbf{let } R' = \textbf{replace}(l, R) \textbf{ in } (R'\{\texttt{cwp} \rightsquigarrow \textbf{pre\_cwp}(R(\texttt{cwp}))\}, F') \\
& \quad \textbf{where } Q = (R, F) \\
\textbf{inc\_win}(Q) & \triangleq \begin{cases} \textbf{left\_win}(Q) & \textbf{if win\_valid}(\textbf{post\_cwp}(R(\texttt{cwp})), R) \\ \bot & \textbf{otherwise} \end{cases} \\
& \quad \textbf{where } Q = (R, F) \\
\textbf{left\_win}Q & \triangleq \textbf{let } (F', l) := \textbf{left}(F, \textbf{fetch}(R)) \textbf{ in} \\
& \quad \textbf{let } R' = \textbf{replace}(l, R) \textbf{ in } (R'\{\texttt{cwp} \rightsquigarrow \textbf{post\_cwp}(R(\texttt{cwp}))\}, F') \\
& \quad \textbf{where } Q = (R, F) \\
\textbf{fetch}(R) & \triangleq (R[\texttt{r}_8, \ldots, \texttt{r}_{15}], R[\texttt{r}_{16}, \ldots, \texttt{r}_{23}], R[\texttt{r}_{24}, \ldots, \texttt{r}_{31}]) \\
\textbf{right}(F, lw) & \triangleq (\mathrm{fm}_l :: \mathrm{fm}_i :: F', (\mathrm{fm}_1, \mathrm{fm}_2, \mathrm{fm}_o)) \\
& \quad \textbf{where } F = F' \circ \mathrm{fm}_1 \circ \mathrm{fm}_2,\ lw = (\mathrm{fm}_o, \mathrm{fm}_l, \mathrm{fm}_i) \\
\textbf{left}(F, lw) & \triangleq (F' \circ \mathrm{fm}_o \circ \mathrm{fm}_l, (\mathrm{fm}_i, \mathrm{fm}_1, \mathrm{fm}_2)) \\
& \quad \textbf{where } F = \mathrm{fm}_1 :: \mathrm{fm}_2 :: F',\ lw = (\mathrm{fm}_o, \mathrm{fm}_l, \mathrm{fm}_i) \\
\textbf{replace}(lw, R) & \triangleq R\{[\texttt{r}_8, \ldots, \texttt{r}_{15}] \rightsquigarrow \mathrm{fm}_o\}\{[\texttt{r}_{16}, \ldots, \texttt{r}_{23}] \rightsquigarrow \mathrm{fm}_l\}\{[\texttt{r}_{24}, \ldots, \texttt{r}_{31}] \rightsquigarrow \mathrm{fm}_i\} \\
& \quad \textbf{where } lw = (\mathrm{fm}_o, \mathrm{fm}_l, \mathrm{fm}_i) \\
R[\texttt{r}_i, \ldots, \texttt{r}_{i+7}] & \triangleq [R(\texttt{r}_i), \ldots, R(\texttt{r}_{i+7})] \\
R\{[\texttt{r}_i, \ldots, \texttt{r}_{i+7}] \rightsquigarrow \mathrm{fm}\} & \triangleq R\{\texttt{r}_i \rightsquigarrow w_0\} \ldots \{\texttt{r}_{i+7} \rightsquigarrow w_7\} \\
& \quad \textbf{where } \mathrm{fm} = [w_0, \ldots, w_7] \\
\textbf{win\_valid}(w_{id}, R) & \triangleq 2^{w_{id}} \& R(\texttt{wim}) = 0
\end{array}
$$

Fig. 8. Auxiliary Definitions for Instruction `save` and `restore`

ister window by $\textbf{inc\_win}(R, F)$, which will do a left rotation by $\textbf{left\_win}(R, F)$ for register window. The auxiliary definitions for operational semantics of `save` and `restore` are shown in Fig. 8.

The last layer is designed for some simple instructions, whose executions only touch memory and register file.

4 Program Logic

In this section, we introduce the assertion language and program logic designed for SPARCv8 program. We will elaborate how our logic handles the features of SPARCv8 in detail. Finally, we will present our semantic approach for establishing soundness.

$$(Asrt)\ p, q \triangleq pa \mid p \wedge q \mid p \vee\ q \mid p * q \mid \forall x.\, p \mid \exists x.\, p \mid p\!\downarrow$$

$$(AtomAsrt)\ \ pa \triangleq \mathrm{emp} \mid l \mapsto w \mid \mathtt{a} =_a w \mid \mathtt{o} = w \mid rn \mapsto w \mid [n]\mathtt{sr} \mapsto w \mid \langle p \rangle \mid \{\mathtt{cwp} = w_{id}, F\} \mid \mathrm{true} \mid \mathrm{false} \mid \ldots$$

Fig. 9. Syntax of Assertions

$$\begin{array}{ll}
S \models \mathrm{emp} & \triangleq S.M = \emptyset \wedge S.Q.R = \emptyset \\
S \models l \mapsto v & \triangleq S.M = \{l \rightsquigarrow v\} \wedge S.Q.R = \emptyset \\
S \models rn \mapsto v & \triangleq S.Q.R = \{rn \rightsquigarrow v\} \wedge rn \notin \{S.D\}_R \wedge S.M = \emptyset \\
S \models [n]\mathtt{sr} \mapsto v & \triangleq S \models \{n\}\mathtt{sr} \mapsto v \vee \ldots \vee S \models \{0\}\mathtt{sr} \mapsto v \vee S \models \mathtt{sr} \mapsto v \\
S \models \{n\}\mathtt{sr} \mapsto v & \triangleq (n, \mathtt{sr}, v) \in S.D \wedge \mathrm{dom}(R) = \{\mathtt{sr}\} \wedge \mathbf{noDup}\,(\mathtt{sr}, \{S.D\}) \wedge S.M = \emptyset \\
S \models \{\mathtt{cwp} = w_{id}, F\} & \triangleq S \models \mathtt{cwp} \mapsto w_{id} \wedge S.Q.F = F \\
S \models \mathtt{a} =_a w & \triangleq [\![\mathtt{a}]\!]_{S.Q.R} = w \wedge \mathbf{word_align}(w) \\
S \models \mathtt{o} = w & \triangleq [\![\mathtt{o}]\!]_{S.Q.R} = w \\
S \models p\!\downarrow & \triangleq \exists R', D'.\ \mathbf{exe_delay}(R', D') = (S.Q.R, S.D) \wedge S|_{R',D'} \models p \\
S \models p_1 * p_2 & \triangleq \exists S_1, S_2.\ S_1 \models p_1 \wedge S_2 \models p_2 \wedge S = S_1 \uplus S_2 \\
S \models \langle p \rangle & \triangleq p \wedge S \models \mathrm{emp}
\end{array}$$

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

$$\{D\}_R \triangleq \begin{cases} \{\mathtt{sr}\} \cup \{D'\}_R & \textbf{if } D = (n, \mathtt{sr}, w) :: D' \\ \emptyset & \textbf{if } D = \emptyset \end{cases} \qquad \mathbf{noDup}(\mathtt{sr}, lr) \triangleq \mathtt{sr} \notin lr \backslash \{\mathtt{sr}\}$$

$$S|_{R',D'} \triangleq (S.M, (R', S.Q.F), D')$$

$$[\![\mathtt{o}]\!]_R \triangleq \begin{cases} R(r) & \textbf{if } \mathtt{o} = r \\ w & \textbf{if } \mathtt{o} = w, \\ & \quad -4096 \leq w \leq 4095 \\ \perp & \textbf{otherwise} \end{cases} \qquad [\![\mathtt{a}]\!]_R \triangleq \begin{cases} [\![\mathtt{o}]\!]_R & \textbf{if } \mathtt{a} = \mathtt{o} \\ v_1 + v_2 & \textbf{if } \mathtt{a} = r + \mathtt{o} \textbf{ and } R(r) = v_1 \\ & \quad \textbf{and } [\![\mathtt{o}]\!]_R = v_2 \\ \perp & \textbf{otherwise} \end{cases}$$

Fig. 10. Semantics of Assertions

4.1 Assertion Language

Our assertion language is shown in Fig. 9. And the semantics of some of them are presented in Fig. 10.

Assertion emp says that the memory and register file are both empty. $l \mapsto v$ specifies a singleton memory cell with value v stored in address l. $rn \mapsto v$ says that there is only one cell in register file that maps the register name rn to value v and rn is not in delay list. We use assertion $\{\mathtt{cwp} = w_{id}, F\}$ to describe the state of window registers, which means the frame list in current state is F and the identity of current window is w_{id}. The assertion $\mathtt{a} =_a w$ holds under a state S if the evaluation of the address expression $\mathtt{a}$ whose definition is shown in Fig. 4 is integer w, and w is four bytes alignment by $\mathbf{word\_align}(w)$. A state

S satisfies $\mathtt{o} = w$ if the evaluation of expression $\mathtt{o}$, which is either a register r or an integer, is w under the state S.

The most novel assertion here is $[n]\mathtt{sr} \mapsto v$. It describes a cell for special register $\mathtt{sr}$ in register file. However, the specific value stored in $\mathtt{sr}$ is not known. We can just infer that the value of $\mathtt{sr}$ will certainly be v after n cycles. So, the current state of special register $\mathtt{sr}$ may be either a delay item $(t, \mathtt{sr}, w)$, where $0 \leq t \leq n$, in delay list D, or not in delay list and $\mathtt{sr} \mapsto v$ holding. Here, we just permit at most one delay item for a specific special register $\mathtt{sr}$ in delay list at each moment. Because the concrete delayed time is implementation dependently, and programmer usually do not touch the target special register after instruction $\mathtt{wr}$, only if they can make sure that the write operation has already effected on the state of the target special register. By using assertion $[n]\mathtt{sr} \mapsto v$ to describe the state of special register which may be recorded in delay list, we do not have to expose the detail of delay list in our assertion. For example, if there is a delay item $(t, \mathtt{sr}, v)$ in delay list D, we can describe the state of special register as $[n]\mathtt{sr} \mapsto v$ (where $n \geq t$) or $\{t\}\mathtt{sr} \mapsto v$.

The assertion $p\downarrow$ describes a state satisfying assertion p before doing operation **exe_delay**. We require that $p\downarrow$ holds under a state S if there exists a state satisfying assertion p and can change to state S by the execution of **exe_delay**. This assertion plays an important role in our inference rules designing, because the SPARCv8 have to do **exe_delay** in each cycle. If predicate p holds before doing **exe_delay**, the state after executing **exe_delay** can be described simply just as the predicate $p\downarrow$.

Separation conjunction is the most important assertion in Separation Logic [**?**]. Previous works [**?**, **?**, **?**, **?**] for assembly code verification just split memory into disjoint parts and register file in their work is a total function. In our work, we define that both memory and register file are partial function. The operation separation conjunction not only splits memory into two disjoint parts, but also register file. We think that this change will make our specification more concise, and let us do not have to consider any side condition about aliases.

4.2 Inference Rules

We define a set of inference rules so that we can check SPARCv8 program mechanized. We start our introduction by presenting the program specification first. The code specification θ and code heap specification Ψ are defined as blow :

$$
\begin{array}{lrcl}
\text{(LogicVL)} & \iota & \in & \text{list lvar} \\
\text{(SpecPre)} & \text{fp} & \in & \text{LogicVL} \to \mathit{Asrt} \\
\text{(SpecPost)} & \text{fq} & \in & \text{LogicVL} \to \mathit{Asrt} \\
\text{(CdSpec)} & \theta & ::= & (\text{fp}, \text{fq}) \\
\text{(CdHpSpec)} & \Psi & ::= & \{\mathtt{f} \rightsquigarrow \theta\}^*
\end{array}
$$

Instead of using a pair of assertions as specification for code block, we define a list of logical variables LogicVL as a parameter of code block's pre/postcondition to get connection between initial state and final state. A predicate fp specifies the initial state, and the other predicate fq describe the state at the return point

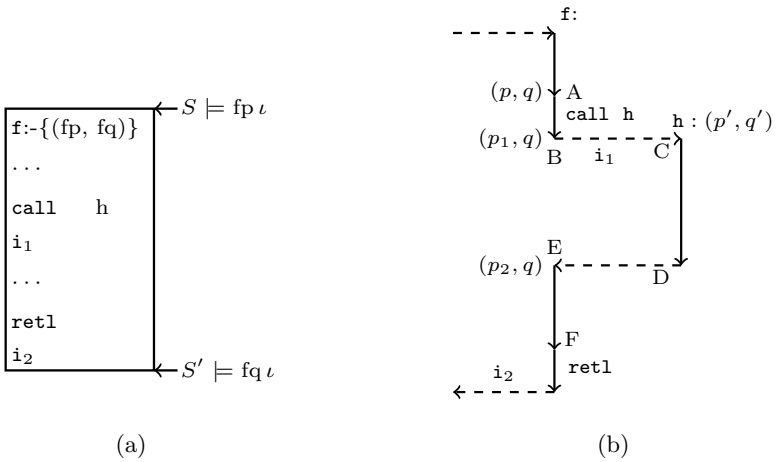

Fig. 11. Function Call/Return and Delayed-control-transfer Handling

$$
\begin{array}{l}
- \ \{(\mathrm{fp}, \mathrm{fq})\} \\
\quad \texttt{add} \quad \%\texttt{i}_0, \%\texttt{i}_1, \%\texttt{l}_7 \\
\quad \texttt{add} \quad \%\texttt{l}_7, \%\texttt{i}_2, \%\texttt{l}_7 \\
\quad \texttt{retl} \\
\quad \texttt{nop}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{fp} \stackrel{\triangle}{=} \lambda\, lv.\ \exists\, x, y, z, \texttt{f}_{ret}. \\
\qquad \langle lv = x :: y :: z :: \texttt{f}_{ret} :: \mathrm{nil} \rangle \\
\qquad * \%\texttt{i}_0 \mapsto x * \%\texttt{i}_1 \mapsto y * \%\texttt{i}_2 \mapsto z \\
\qquad * \%\texttt{l}_7 \mapsto \_ * \%\texttt{l}_{15} \mapsto \texttt{f}_{ret} \\
\mathrm{fq} \stackrel{\triangle}{=} \lambda\, lv.\ \exists\, x, y, z, \texttt{f}_{ret}. \\
\qquad \langle lv = x :: y :: z :: \texttt{f}_{ret} :: \mathrm{nil} \rangle \\
\qquad * \%\texttt{i}_0 \mapsto x * \%\texttt{i}_1 \mapsto y * \%\texttt{i}_2 \mapsto z \\
\qquad * \%\texttt{i}_7 \mapsto x + y + z * \texttt{r}_{15} \mapsto \texttt{f}_{ret}
\end{array}
$$

Fig. 12. Example for Function Specification

of the current function. Fig. 11(a) shows the meaning of our specification (fp, fq) for function f. The code specification Ψ is defined as a finite mapping from code labels to code specifications. Under the definition of our specification, we give a simple example to introduce how to write a specification for a function implemented by SPARCv8 in Fig. 12.

Example of Function Specification. The function shown in Fig. 12 sums the values of registers $\%\texttt{i}_0$, $\%\texttt{i}_1$ and $\%\texttt{i}_2$ together and writes the result into register $\%\texttt{l}_7$. The specification of the function is presented by $(\mathrm{fp}, \mathrm{fq})$. Here, we need to describe the state of register $\texttt{r}_{15}$, which saves the label of occurring function call, in pre/postcondition. However, we don't have to know the concrete value of $\texttt{r}_{15}$ with the inspiration provided by SCAP. Both of pre/postcondition parameter a list of logic variables for getting connection between them. We will find that using a list of logic variables is essential in some situation for writing specification. For example, we can't describe the call point stored in register $\texttt{r}_{15}$

equally between initial and final state, without the helping of the list of logic variables. The main role of list of logic variables is to describe the same logic variables used both in pre/postcondition.

Fig. 13 shows a part of inference rules in our logic. The top rule **CDHP** is used to check whether a code heap is well-formed. We consider that a code heap is well-formed if all of the code blocks given specifications can be verified by well-formed sequence.

The well-formed instruction sequence is designed for code block checking. The **SEQ** rule will be applied when meeting an instruction sequence starting with a simple instruction `i`. It tells us that the simple instruction `i` can be checked by inference rules presented in well-formed instruction and the rest instruction sequence also satisfies well-formed instruction sequence. The precondition for instruction `i` is not p, but $p\downarrow$, because the **exe_delay** mentioned in Sec. 3.2 will reduce the delay time of delay items in delay list and change the state of register file if a delay item's delay time is zero. The following Lemma. 4.1 makes sure that if predicate p satisfies the current state, then $p\downarrow$ will hold after the execution of **exe_delay**. The proof are straight forward by the semantics of $p\downarrow$.

Lemma 4.1. If $(M,(R,F),D) \models p$, $\mathbf{exe\_delay}(R,D) = (R',D')$, then we get $(M,(R',F),D') \models (p\downarrow)$.

Delayed-control-transfer handling. We present the **CALL** rule and **RET** rule for certifying function call/return. They also show our approach to handle delayed control-transfer feature in SPARCv8, because instruction `call` and `retl` are all delayed control-transfer instructions. Fig 11(b) presents how our logic supports function call/return verification with delayed control-transfer. In Fig. 11(b), we meet the instruction `call` at point A, and our **CALL** rule verifies the instruction `call` and its following instruction $\mathtt{i}_1$ together, because the actual function call in SPARCv8 occurs after the execution of $\mathtt{i}_1$ at point C. The **CALL** rule makes sure that precondition p' of function h will be satisfied after instruction $\mathtt{i}_1$ and the current function can resume executing safely after h returning from point E. The SPARCv8 will save the label `f` of instruction `call f'` in general register $\mathtt{r}_{15}$. So, we also enforces that the call point saved in $\mathtt{r}_{15}$ register will be restored when function `h` returns, in need of specifying a valid address to return to. The **RET** rule in our logic does not need to know any specific information about $\mathtt{r}_{15}$ by the inspiration of the previous work SCAP. The instruction `retl` is also a delayed control-transfer instruction in SPARCv8, and the control transfer returns to the caller occurring after the execution of instruction $\mathtt{i}_2$ following `retl`. So, we still verify the instruction `retl` and its following instruction together. The postcondition of function `f` will be hold after the execution of instruction $\mathtt{i}_2$. Here, we do not have to care about the return address. We just have to constrain that the instruction following `retl` does not update the value of $\mathtt{r}_{15}$ by $\mathbf{fretSta}(p\downarrow\downarrow, q)$ (defined in Fig. 14), in order to keep the address saved in $\mathtt{r}_{15}$ are equal at the call and return points.

Having two program counters pc and npc makes the SPARCv8 programming more flexible. And the call conventions are richer than other assembly code. Fig.

$\boxed{\Psi \vdash C : \Psi'}$ **(Well-formed Code Heap)**

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

$\boxed{\Psi \vdash \{(p,q)\}\ \mathtt{f} : \mathbb{I}}$ **(Well-formed Sequence)**

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

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

$$\frac{\vdash \{p\downarrow\downarrow\}\ \mathtt{i}\ \{p'\} \qquad p' \Rightarrow q \qquad \mathbf{fretSta}\,(p\downarrow\downarrow, p')}{\Psi \vdash \{(p,q)\}\ \mathtt{f} : \mathtt{retl};\ \mathtt{i}}\ \textbf{(RET)}$$

$$\frac{\begin{array}{l}\mathtt{f}' \in \mathrm{dom}(\Psi) \quad \Psi(\mathtt{f}') = (\mathrm{fp}, \mathrm{fq}) \quad \Psi \vdash \{((p \wedge \mathtt{z} = 0)\downarrow, q)\}\ \mathtt{f}+4 : \mathtt{i};\ \mathbb{I} \\ \vdash \{(p \wedge \mathtt{z} \neq 0)\downarrow\downarrow\}\ \mathtt{i}\ \{p'\} \quad \exists\ \iota.\ (p' \Rightarrow \mathrm{fp}\ \iota * p_r)\ \wedge\ (\mathrm{fq}\ \iota * p_r \Rightarrow q)\end{array}}{\Psi \vdash \{(p,q)\}\ \mathtt{f} : \mathtt{be}\ \mathtt{f}';\ \mathtt{i};\ \mathbb{I}}\ \textbf{(BE)}$$

$$\frac{\Psi \vdash \{(p,q)\}\ \mathtt{f} : \mathbb{I}}{\Psi \vdash \{(p * p_r, q * p_r)\}\ \mathtt{f} : \mathbb{I}}\ \textbf{(seq\_frame)}$$

$\boxed{\vdash \{p\}\ \mathtt{i}\ \{q\}}$ **(Well-formed Instruction)**

$$\frac{p \Rightarrow \mathtt{a} =_a l \quad p \Rightarrow l \mapsto v * \mathtt{r}_s \mapsto \_ * p_1}{\vdash \{p\}\ \mathtt{ld}\ \mathtt{a}\ \mathtt{r}_d\ \{l \mapsto v * \mathtt{r}_s \mapsto v * p_1\}}\ \textbf{(LD)} \qquad \frac{p \Rightarrow (\mathtt{r}_s = v_1 \wedge \mathtt{o} = v_2) \quad p \Rightarrow \mathtt{r}_d \mapsto \_ * p_1}{\vdash \{p\}\ \mathtt{add}\ \mathtt{r}_s\ \mathtt{o}\ \mathtt{r}_d\ \{(\mathtt{r}_d \mapsto v_1 + v_2) * p_1\}}\ \textbf{(ADD)}$$

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

$$\frac{\mathtt{sr} \mapsto \_ * p \Rightarrow (\mathtt{r}_s = v_1\ \wedge\ \mathtt{o} = v_2)}{\vdash \{\mathtt{sr} \mapsto \_ * p\}\ \mathtt{wr}\ \mathtt{r}_s\ \mathtt{o}\ \mathtt{sr}\ \{([3]\mathtt{sr} \mapsto v_1\ \mathbf{xor}\ v_2) * p\}}\ \textbf{(WR)}$$

$$\frac{\begin{array}{l}p \Rightarrow (\mathtt{r}_s = v_1 \wedge \mathtt{o} = v_2) \qquad w'_{id} = \mathbf{pre\_cwp}\,(w_{id}) \qquad v\,\&\,2^{w'_{id}} = 0 \\ p \Rightarrow \{\mathtt{cwp} = w_{id}, F \circ \mathrm{fm}_1 \circ \mathrm{fm}_2\} * \mathrm{R}_o = \mathrm{fm}_o * \mathrm{R}_l = \mathrm{fm}_l * \mathrm{R}_i = \mathrm{fm}_i * p_1 \\ \{\mathtt{cwp} = w'_{id}, \mathrm{fm}_l :: \mathrm{fm}_i :: F\} * \mathrm{R}_o = \mathrm{fm}_1 * \mathrm{R}_l = \mathrm{fm}_2 * \mathrm{R}_i = \mathrm{fm}_o * p_1 \Rightarrow \mathtt{r}_d \mapsto \_ * p_2\end{array}}{\vdash \{(\mathtt{wim} \mapsto v) * p\}\ \mathtt{save}\ \mathtt{r}_s\ \mathtt{o}\ \mathtt{r}_d\ \{(\mathtt{wim} \mapsto v) * (\mathtt{r}_d \mapsto v_1 + v_2) * p_2\}}\ \textbf{(SAVE)}$$

$$\frac{\begin{array}{l}p \Rightarrow (\mathtt{r}_s = v_1 \wedge \mathtt{o} = v_2) \qquad w'_{id} = \mathbf{post\_cwp}\,(w_{id}) \qquad v\,\&\,2^{w'_{id}} = 0 \\ p \Rightarrow \{\mathtt{cwp} = w_{id}, \mathrm{fm}_1 :: \mathrm{fm}_2 :: F\} * \mathrm{R}_o = \mathrm{fm}_o * \mathrm{R}_l = \mathrm{fm}_l * \mathrm{R}_i = \mathrm{fm}_i * p_1 \\ \{\mathtt{cwp} = w'_{id}, F \circ \mathrm{fm}_o \circ \mathrm{fm}_l\} * \mathrm{R}_o = \mathrm{fm}_i * \mathrm{R}_l = \mathrm{fm}_1 * \mathrm{R}_i = \mathrm{fm}_2 * p_1 \Rightarrow \mathtt{r}_d \mapsto \_ * p_2\end{array}}{\vdash \{(\mathtt{wim} \mapsto v) * p\}\ \mathtt{restore}\ \mathtt{r}_s\ \mathtt{o}\ \mathtt{r}_d\ \{(\mathtt{wim} \mapsto v) * (\mathtt{r}_d \mapsto v_1 + v_2) * p_2\}}\ \textbf{(RESTORE)}$$

Fig. 13. Seleted Inference Rules

1 shows three common methods for doing function call and implementations of function **sum3**. Although their implementations have a few differences, they

$$
\begin{aligned}
\mathrm{R}_o = [w_0, \ldots, w_7] &\triangleq \mathtt{r}_8 \mapsto w_0 * \cdots * \mathtt{r}_{15} \mapsto w_7 \\
\mathrm{R}_l = [w_0, \ldots, w_7] &\triangleq \mathtt{r}_{16} \mapsto w_0 * \cdots * \mathtt{r}_{23} \mapsto w_7 \\
\mathrm{R}_i = [w_0, \ldots, w_7] &\triangleq \mathtt{r}_{24} \mapsto w_0 * \cdots * \mathtt{r}_{31} \mapsto w_7 \\
\textbf{fretSta}\ (p_1, p_2) &\triangleq \forall S, S'.\ S \models p_1 \to S' \models p_2 \to \\
&\qquad (\exists v.\ S.Q.R(\mathtt{r}_{15}) = v \wedge S.Q.R(\mathtt{r}_{15}) = v) \\
p \Rightarrow q &\triangleq \forall S, S'.\ S \models p \to S' \models q
\end{aligned}
$$

Fig. 14. Auxiliary Definition for Inference Rules

have the same point that the execution of instruction `call` will save its label in $\mathtt{r}_{15}$ and this label will be restored in $\mathtt{r}_{15}$ when function returns. Our logic is designed for general SPARCv8 code verification. The **CALL** rule for instruction `call` in our logic makes as little restrictions as possible for caller and callee, and just requires that the label stored in $\mathtt{r}_{15}$ when function returns is the same as the label of instruction `call`, such that (fq $\iota \Rightarrow \mathtt{r}_{15} = \mathtt{f}$). And this is the same point that these three common function call convention following. So, the delayed control-transfer processing in our logic can support these three function call methods verification well.

We also define some practical useful rules such as **seq_frame** rule, in order to make our logic be used more convenient. The assertion p_r describes a part of state that the verification work doesn't care about. The **seq_frame** rule provides a local way for code block reasoning. We don't have to consider the program state that the current instruction sequence execution doesn't touch.

The bottom layer of our logic is well-formed instruction for single instructions verification. Most of them are intuitive and easy to understand, like **LD** rule and **ADD** rule. The operations of instruction `save` and `restore` play the similar role with stack operation **push** and **pop** to store and restore the context.

Register Windows Handling. We introduce the inference rule for instruction `save` here. The instruction `save` will operate the frame list to store current context by window rotation and update the value of register $\mathtt{r}_d$ of the new window. We first extract the values of register $\mathtt{r}_s$ and expression `o` from the current window, and then save the local and in registers in frame list. Fig. 15 shows the process of window rotation caused by instruction `save`. We use the frames fm_o, fm_l and fm_i to present the contents of out, local and in registers, and the state of current window which is composed by out, local and in registers, can be described by assertion $\mathrm{R}_o = \mathrm{fm}_o * \mathrm{R}_l = \mathrm{fm}_l * \mathrm{R}_i = \mathrm{fm}_i$. We use assertion $\{\mathtt{cwp} = w_{id}, F \circ \mathrm{fm}_1 \circ \mathrm{fm}_2\}$ to present the state of frame list and the current window pointer `cwp`. It tells us that the identity of the current window is w_{id} and the state of the frame list is $F \circ \mathrm{fm}_1 \circ \mathrm{fm}_2$. The operation "$\circ$" means appending a frame to the tail of a frame list, just as showing in Fig. 15 that fm_1 and fm_2 are at the of the frame list. So, the current state of the register windows can be presented as $\{\mathtt{cwp} = w_{id}, F \circ \mathrm{fm}_1 \circ \mathrm{fm}_2\} * \mathrm{R}_o = \mathrm{fm}_o * \mathrm{R}_l = \mathrm{fm}_l * \mathrm{R}_i = \mathrm{fm}_i$. The execution of instruction `save` will check whether the previous window is

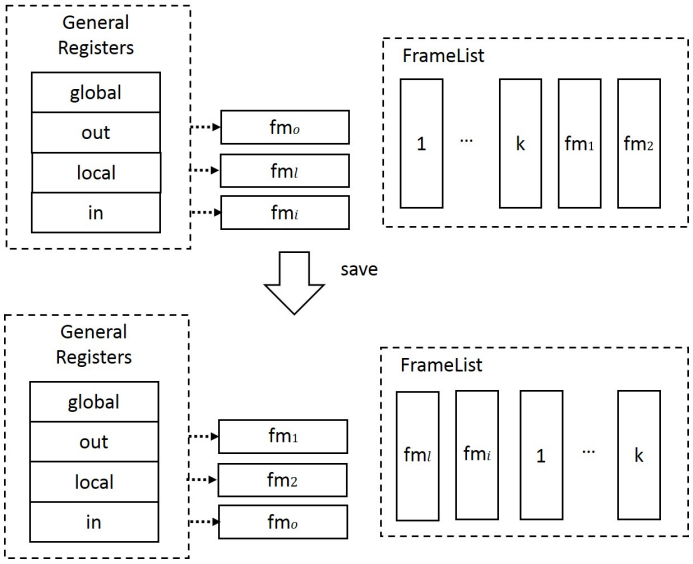

Fig. 15. Window Rotation for Instruction Save

a valid one. Each bit of special register `wim` marks whether the corresponding window is valid. In SPARCv8 programming, we usually set a window invalid, in order to avoid window overflow/underflow. Supposing the value of register `wim` is v, we can justify that the previous window is valid by $v \,\&\, 2^{w'_{id}} = 0$ (where the symbol & means the operation "and"). For example, if the value of `wim` is 8 (can also be presented as 2^3), it means the window, whose identity is 3, is invalid. If the instruction `save` want to rotate to this window, we can find that this execution will not be permitted, because the result of $8\&2^3$ is not equal to 0. When doing window rotation caused by instruction `save`, The contents of local register fm_l and in registers fm_i of the current window whose identity is w_{id} will be saved to the frame list, and 2 frames from the tail of frame list will be the new window's out and local registers. So, the new state of register windows is $\{\texttt{cwp} = w'_{id}, \mathrm{fm}_l :: \mathrm{fm}_i :: F\} * \mathrm{R}_o = \mathrm{fm}_1 * \mathrm{R}_l = \mathrm{fm}_2 * \mathrm{R}_i = \mathrm{fm}_o$. Finally, the values extracting from $\texttt{r}_s$ and `o` of original window are summed together and the result is stored in new window's $\texttt{r}_d$ register. The new window's identity w'_{id} can be achieved by **pre_cwp**(w_{id}).

Delay-Write handling. The **WR** rule is designed for verifying instruction `wr` which is a delayed-write instruction. It first extracts the value v_1 of register $\texttt{r}_s$ and v_2 of operational expression `o` from precondition. Then, the state of special register `sr` is updated with $([3]\texttt{sr} \mapsto v_1 \,\mathbf{xor}\, v_2)$, which means we do not certainly know the state of the special register `sr` now but can make sure that $(\texttt{sr} \mapsto v_1 \,\mathbf{xor}\, v_2)$ holds after 3 cycles. Using assertion $[n]p$ to handle delayed-write can help us hide the information about delay list in the specification. SPARCv8 will do **exe_delay** in each cycle. We can find that if assertion $[n]\texttt{sr} \mapsto w$ holds

$\ldots$
$-\ \{(\%\mathtt{i}_0 \mapsto v * \mathtt{Y} \mapsto \_,\ q)\}$
$\mathtt{wr}\quad \%\mathtt{i}_0,\ 0,\ \mathtt{Y}$
$-\ \{(\%\mathtt{i}_0 \mapsto v * [3]\mathtt{Y} \mapsto (v\ \mathbf{xor}\ 0))\ ,\ q)\}$
$-\ \{(\%\mathtt{i}_0 \mapsto v * [3]\mathtt{Y} \mapsto v\ ,\ q)\}$
$\mathtt{nop}$
$-\ \{(\%\mathtt{i}_0 \mapsto v * [2]\mathtt{Y} \mapsto v\ ,\ q)\}$
$\mathtt{nop}$
$-\ \{(\%\mathtt{i}_0 \mapsto v * [1]\mathtt{Y} \mapsto v\ ,\ q)\}$
$\mathtt{nop}$
$-\ \{(\%\mathtt{i}_0 \mapsto v * [0]\mathtt{Y} \mapsto v\ ,\ q)\}$
$\ldots$

Fig. 16. Example for Delayed-write Handling

in the initial state, then $[n-1]\mathtt{sr} \mapsto w$ will hold after doing **exe_delay** if $n > 0$, or $\mathtt{sr} \mapsto w$ will hold if $n = 0$.

$$([n]\mathtt{sr} \mapsto w)\downarrow \Longrightarrow \begin{cases} [n-1]\mathtt{sr} \mapsto w & \textbf{if } n > 0 \\ \mathtt{sr} \mapsto w & \textbf{if } n = 0 \end{cases}$$

The **exe_delay** operation is complicated. It traverse the delay list, reduce the delay time of delay item, and do the write operation if its delay item is zero. The method that we restrict the state of the special register, which may be recorded in delay list, described in a form of "$[n]\mathtt{sr} \mapsto v$" will not debase our logic's expression, because the only way to touch special register in SPARCv8 is by instruction `rd` and `wr`. We can find that this abstraction will achieve more convenient in our verification work. The state of special register described as $[n]\mathtt{sr} \mapsto v$ will simply be change as $[n-1]\mathtt{sr} \mapsto v$ (if $n > 0$) or $\mathtt{sr} \mapsto v$ (if $n = 0$), and other assertions that are irrelevant with delay list will keep stable after the execution of **exe_delay**.

Program in Fig. 16 is a segment of function **ChangeY** in Fig. 2 and we use it as an example to elaborate how our logic handles delayed-write feature in SPARCv8. The state of special register `Y` is $\mathtt{Y} \mapsto \_$ before the program execution, which means that there is no delay item in delay list about `Y`. The state of `Y` is changed to an uncertain state $[3]\mathtt{Y} \mapsto (v\ \mathbf{xor}\ 0)$ after executing `wr`, where the symbol **xor** is the xor operation. The assertion $[3]\mathtt{Y} \mapsto v$ means either the value of the special register `Y` is v and `Y` is not in delay list, or the value of special register `Y` is uncertain and a delay item $(X, \mathtt{Y}, v)$ is recorded in delay list (where $0 \leq X \leq 3$). The concrete value of initial delay time X is implementation dependently, and its maximum is 3. Although the value of special register `Y` may have not been updated at this time, the old value is not necessary. So, just describing the new value v in assertion is reasonable. Then, we apply the **SEQ** rule three times for the following verification. The delay time recorded in assertion is reduced one at each step. After executing the following three `nop`, the delay time reduces to zero, and $\mathtt{Y} \mapsto v$ holds when we verify the next instruction. The instruction `rd` is responsible for getting the value of target special register. The inference rule for instruction `rd` requires that the state of target special

CALLER :
...
$-$ $\{(p\,,\,q)\}$
`mov` 1, $\%\mathtt{o}_0$
$-$ $\{(p_1\,,\,q)\}$
`call` ChangeY
`save` $\%\mathtt{sp}$, -64, $\%\mathtt{sp}$
$-$ $\{(p_2\,,\,q)\}$
`mov` $\%\mathtt{o}_0$, $\%\mathtt{l}_0$
$-$ $\{(p_3\,,\,q)\}$
...

$$p \triangleq \%\mathtt{o}_0, \%\mathtt{sp}, \%\mathtt{l}_0, \mathtt{r}_{15}, \mathtt{Y} \mapsto \_, v_{sp}, \_, \_, v_y * \mathbf{Rs}(\{\mathtt{r}_0, \ldots \mathtt{r}_{31}\} \backslash \{\%\mathtt{o}_0, \%\mathtt{sp}, \%\mathtt{l}_0, \mathtt{r}_{15}\}) * \mathbf{WinSt}(w_{id}, v_{id}, F \circ \mathrm{fm}_1 \circ \mathrm{fm}_2, \{w_{id}, \mathbf{pre\_cwp}(w_{id})\})$$

$$p_1 \triangleq \%\mathtt{o}_0, \%\mathtt{sp}, \%\mathtt{l}_0, \mathtt{r}_{15}, \mathtt{Y} \mapsto 1, v_{sp}, \_, \_, v_y * \mathbf{Rs}(\{\mathtt{r}_0, \ldots \mathtt{r}_{31}\} \backslash \{\%\mathtt{o}_0, \%\mathtt{sp}, \%\mathtt{l}_0, \mathtt{r}_{15}\}) * \mathbf{WinSt}(w_{id}, v_{id}, F \circ \mathrm{fm}_1 \circ \mathrm{fm}_2, \{w_{id}, \mathbf{pre\_cwp}(w_{id})\})$$

$$p_2 \triangleq \%\mathtt{o}_0, \%\mathtt{sp}, \%\mathtt{l}_0, \mathtt{r}_{15}, \mathtt{Y} \mapsto v_y, v_{sp}, \_, \mathtt{f}, 1 * \mathbf{Rs}(\{\mathtt{r}_0, \ldots \mathtt{r}_{31}\} \backslash \{\%\mathtt{o}_0, \%\mathtt{sp}, \%\mathtt{l}_0, \mathtt{r}_{15}\}) * \mathbf{WinSt}(w_{id}, v_{id}, F \circ \_ \circ \_, \{w_{id}, \mathbf{pre\_cwp}(w_{id})\})$$

$$p_3 \triangleq \%\mathtt{o}_0, \%\mathtt{sp}, \%\mathtt{l}_0, \mathtt{r}_{15}, \mathtt{Y} \mapsto v_y, v_{sp}, v_y, \mathtt{f}, 1 * \mathbf{Rs}(\{\mathtt{r}_0, \ldots \mathtt{r}_{31}\} \backslash \{\%\mathtt{o}_0, \%\mathtt{sp}, \%\mathtt{l}_0, \mathtt{r}_{15}\}) * \mathbf{WinSt}(w_{id}, v_{id}, F \circ \_ \circ \_, \{w_{id}, \mathbf{pre\_cwp}(w_{id})\})$$

$$\mathrm{fp}_{cy} \triangleq \lambda\, lv.\ \exists \mathrm{fm}_o, \mathrm{fm}_l, \mathrm{fm}_i, \mathtt{f}_{ret}, x, v_y, v_i, w_{id}, \mathrm{fm}_1, \mathrm{fm}_2, F.\ \langle lv = \mathrm{fm}_i :: \mathrm{fm}_1 :: \mathrm{fm}_2 :: x :: \mathtt{f}_{ret} :: v_y :: v_i :: w_{id} :: \mathrm{nil}\rangle * \mathrm{R}_o = \mathrm{fm}_o * \mathrm{R}_l = \mathrm{fm}_l * \mathrm{R}_i = \mathrm{fm}_i * \mathtt{Y} \mapsto v_y * \mathtt{wim} \mapsto v_i * \{\mathtt{cwp} = w_{id}, \mathrm{fm}_1 :: \mathrm{fm}_2 :: F\} * \langle [\![\mathrm{fm}_i]\!]^{\mathrm{in}}_{\mathtt{r}_{31}} = \mathtt{f}_{ret} \wedge [\![\mathrm{fm}_i]\!]^{\mathrm{in}}_{\%\mathtt{i}_0} = x \wedge \mathbf{WinValid}(v_i, \mathbf{post\_cwp}(w_{id}))\rangle$$

$$\mathrm{fq}_{cy} \triangleq \lambda\, lv.\ \exists \mathrm{fm}_i, \mathtt{f}_{ret}, x, v_y, v_i, w_{id}, \mathrm{fm}_1, \mathrm{fm}_2, F.\ \langle lv = \mathrm{fm}_i :: \mathrm{fm}_1 :: \mathrm{fm}_2 :: x :: \mathtt{f}_{ret} :: v_y :: v_i :: w_{id} :: \mathrm{nil} \wedge [\![\mathrm{fm}_i]\!]^{\mathrm{out}}_{\mathtt{r}_{15}} = \mathtt{f}_{ret}\rangle * \mathrm{R}_o = \mathrm{fm}_i\{o_0 \rightsquigarrow v_y\}^{\mathrm{out}} * \mathrm{R}_l = \mathrm{fm}_1 * \mathrm{R}_i = \mathrm{fm}_2 * \mathtt{Y} \mapsto x * \mathtt{wim} = v_i * \{\mathtt{cwp} = \mathbf{post\_cwp}(w_{id}), F \circ \_ \circ \_\}$$

$$\mathbf{WinSt}(w_{id}, v_{id}, F, \{v_1, \ldots, v_n\}) \triangleq \mathtt{wim} \mapsto v_{id} * \{\mathtt{cwp} = w_{id}, F\} * \langle \mathbf{WinValid}(v_{id}, v_1) \wedge \cdots \wedge \mathbf{WinValid}(v_{id}, v_n)\rangle$$

$$rn_0, \ldots, rn_t \mapsto w_0, \ldots, w_t \triangleq rn_0 \mapsto w_0 * \ldots rn_t \mapsto w_t$$

$$\mathbf{Rs}(\{rn_1, \ldots, rn_t\}) \triangleq rn_0 \mapsto \_ * \cdots * rn_t \mapsto \_ \qquad \mathbf{WinValid}(v_i, w_{id}) \triangleq v_i \,\&\, 2^{w_{id}} = 0$$

$$[\![\mathrm{fm}]\!]^{\mathrm{in}}_{\mathtt{r}_i} \triangleq \begin{cases} \mathrm{fm}[i-24] & 24 \le i \le 31 \\ \textbf{undefined} & \text{otherwise} \end{cases} \qquad [\![\mathrm{fm}]\!]^{\mathrm{out}}_{\mathtt{r}_i} \triangleq \begin{cases} \mathrm{fm}[i-8] & 8 \le i \le 15 \\ \textbf{undefined} & \text{otherwise} \end{cases}$$

$$\mathrm{fm}\{\mathtt{r}_i \rightsquigarrow w\}^{\mathrm{out}} \triangleq \begin{cases} \mathrm{fm}\{(i-8) \rightsquigarrow w\} & 8 \le i \le 15 \\ \textbf{undefined} & \text{otherwise} \end{cases}$$

Fig. 17. Proof Outline for the Example

register in precondition is not in delay list. So, getting a value of special register whose state is described as "$[n]\mathtt{sr} \mapsto v$" is prohibited in our logic.

Example for SPARCv8 Code Verification. Now, we use our logic to verify the program presented in Fig. 2. The specification of function ChangeY is composed by precondition fp_{cy} and postcondition fq_{cy}. The definition of fp_{cy}, fq_{cy}, and the outline of verifying the code segment of CALLER is shown in Fig. 17. Here, the registers $\%\mathtt{o}_0, \%\mathtt{l}_0, \%\mathtt{sp}, \%\mathtt{i}_0$ are just aliases of general registers $\mathtt{r}_8, \mathtt{r}_{16}, \mathtt{r}_{14}, \mathtt{r}_{24}$. The precondition fp_{cy} of function **ChangeY** tells us that (1) the call address is stored in $\mathtt{r}_{31}$ in initial state, (2) The value to update special

register Y is stored in general register $\mathtt{r}_0$, and (3) the post window is valid, so that instruction **restore** can execution safely. As for postcondition fq_{cy} describing the state at the return point, it requires that (1) the return address will be restored in $\mathtt{r}_{15}$, (2) The value Y will be updated to a new one, and (3) the value register $\%\mathtt{o}_0$ will be equal to the Y's original value. We use fm[i] (where $0 \leq i \leq 7$) as the i^{th} number of frame fm, and fm$\{i \rightsquigarrow w\}$ (where $0 \leq i \leq 7$) meaning assigning value w to the i^{th} number of frame fm.

With the specification of function **ChangeY**, we can verify the code segment presented in **CALLER**. We let assertion q as the postcondition of **CALLER** and assertion p describing the initial state. We define $\mathbf{Rs}(\{rn_1, \ldots, rn_t\})$ to present the state of a set of registers whose specific values we do not care about. Its parameter $\{rn_1, \ldots, rn_t\}$ is a set of registers. And its instantiation $\{\mathtt{r}_0, \ldots, \mathtt{r}_{31}\} \backslash \{\%\mathtt{o}_0, \%\mathtt{sp}, \%\mathtt{l}_0, \mathtt{r}_{15}\}$ in our example means the set of registers of $\mathtt{r}_0 \sim \mathtt{r}_{31}$, excluding $\%\mathtt{o}_0$, $\%\mathtt{sp}$, $\%\mathtt{l}_0$ and $\mathtt{r}_{15}$. We have to describe them in our predicate in order to pass the verification of instruction `save` and `restore`. We require that the current and previous windows are valid in assertion p, so that instruction `save` and `restore` can execute safely. The head instruction `mov` sets the value of register $\%\mathtt{o}_0$ as integer one, such as assertion p_1 showing. Then, we check instruction `call` and its following instruction together and change the precondition from p_1 to p_2. We suppose that the label of instruction `call` is `f`, and `f` will be saved in $\mathtt{r}_{15}$ after executing instruction `call`.

The program shown in Fig. 18 is another common call convention in SPARCv8 we have mentioned. The **CALLER** uses $\%\mathtt{o}_0$, $\%\mathtt{o}_1$, $\%\mathtt{o}_2$ to pass parameters for calling function **sum3**. The operation of setting the third parameter 3 to $\%\mathtt{o}_3$ is following the instruction "`call` **sum3**". This operation will be finished before control transfer occurring because of the delayed control transfer. Our logic is designed for general SPARCv8 code, so it can work well for verifying the most common function call conventions in SPARCv8 by a uniform way. The specification of function **sum3** is presented in Fig. 18. Here, $\mathrm{fp}_{\mathrm{sum3}}$ is the precondition, and $\mathrm{fq}_{\mathrm{sum3}}$ is the postcondition. The specification of function **sum3** tells us that if the values of $\%\mathtt{o}_0$, $\%\mathtt{o}_1$, $\%\mathtt{o}_2$ are x, y and z before the program's execution, then the value of $\%\mathtt{o}_0$ is equal to $x + y + z$ after the executing of the function **sum3**.

4.3 Soundness

We elaborate the semantics of our logic and establish its soundness now.

Definition 4.2 (Semantics of Well-formed Instruction).

$$\models \{p\}\, \mathtt{i}\, \{q\} \triangleq \forall S.\ S \models p \rightarrow (\exists S'.\ (S \xrightarrow{\mathtt{i}} S' \wedge S' \models q)$$

The soundness definition of well-formed instruction is straight-forward. It tells that if there is a state S satisfying the precondition p, then the execution of instruction `i` from state S will reach a state S' satisfying the postcondition q. The soundness proof of well-formed instruction can be completed by discussing each case of instruction `i`.

CALLER :

```
...
mov     1, %o0
mov     2, %o1
call    sum3
mov     3, %o3
mov     %o0, %l7
...
```

sum3 :

```
save    %sp, -64, %sp
add     %i0, %i1, %l7
add     %l7, %i2, %l7
ret
restore %l7, 0, %o0
```

$$\mathrm{fp}_{\mathrm{sum3}} \triangleq \lambda\, lv.\ \exists x, y, z, \mathtt{f}_{ret}, \mathrm{fm}_o, \mathrm{fm}_l, \mathrm{fm}_i, w_{id}, v_i, F, \mathrm{fm}_1, \mathrm{fm}_2.$$
$$\langle lv = x :: y :: z :: \mathtt{f}_{ret} :: \mathrm{nil}\rangle * \mathrm{R}_o = \mathrm{fm}_o * \mathrm{R}_l = \mathrm{fm}_l * \mathrm{R}_i = \mathrm{fm}_i$$
$$*\langle [\![\mathrm{fm}_o]\!]^{\mathrm{out}}_{\%\mathtt{o}_0} = x \wedge [\![\mathrm{fm}_o]\!]^{\mathrm{out}}_{\%\mathtt{o}_1} = y \wedge [\![\mathrm{fm}_o]\!]^{\mathrm{out}}_{\%\mathtt{o}_2} = z \wedge [\![\mathrm{fm}_o]\!]^{\mathrm{out}}_{\mathtt{r}_{15}} = \mathtt{f}_{ret}\rangle$$
$$*\mathbf{WinSt}(w_{id}, v_{id}, F \circ \mathrm{fm}_1 \circ \mathrm{fm}_2, \{w_{id}, \mathbf{pre\_cwp}(w_{id})\})$$

$$\mathrm{fq}_{\mathrm{sum3}} \triangleq \lambda\, lv.\ \exists x, y, z, \mathtt{f}_{ret}, \mathrm{fm}_o, \mathrm{fm}_l, \mathrm{fm}_i, w_{id}, v_i, F, \mathrm{fm}_1, \mathrm{fm}_2.$$
$$\langle lv = x :: y :: z :: \mathtt{f}_{ret} :: \mathrm{nil} \wedge [\![\mathrm{fm}_o]\!]^{\mathrm{out}}_{\%\mathtt{o}_0} = x + y + z \wedge [\![\mathrm{fm}_o]\!]^{\mathrm{out}}_{\mathtt{r}_{15}} = \mathtt{f}_{ret}\rangle$$
$$*\mathrm{R}_o = \mathrm{fm}_o * \mathrm{R}_l = \mathrm{fm}_l * \mathrm{R}_i = \mathrm{fm}_i$$
$$*\mathbf{WinSt}(w_{id}, v_{id}, F \circ \mathrm{fm}_1 \circ \mathrm{fm}_2, \{w_{id}, \mathbf{pre\_cwp}(w_{id})\})$$

```
...
```
$-\ \{(p, q)\}$
```
mov     1, %o0
```
$-\ \{(p_1, q)\}$
```
mov     2, %o1
```
$-\ \{(p_2, q)\}$
```
call    sum3
mov     3, %o2
```
$-\ \{(p_3, q)\}$
```
mov     %o0, %l7
...
```

$$p \triangleq \%\mathtt{o}_0 \mapsto \_ * \%\mathtt{o}_1 \mapsto \_ * \%\mathtt{o}_2 \mapsto \_ * \%\mathtt{l}_7 \mapsto \_$$
$$*\mathbf{Rs}(\{\mathtt{r}_0, \ldots \mathtt{r}_{31}\} \backslash \{\%\mathtt{o}_0, \%\mathtt{o}_1, \%\mathtt{o}_2, \%\mathtt{l}_7\})$$
$$*\mathbf{WinSt}(w_{id}, v_{id}, F \circ \mathrm{fm}_1 \circ \mathrm{fm}_2, \{w_{id}, \mathbf{pre\_cwp}(w_{id})\})$$

$$p_1 \triangleq \%\mathtt{o}_0 \mapsto 1 * \%\mathtt{o}_1 \mapsto \_ * \%\mathtt{o}_2 \mapsto \_ * \%\mathtt{l}_7 \mapsto \_$$
$$*\mathbf{Rs}(\{\mathtt{r}_0, \ldots \mathtt{r}_{31}\} \backslash \{\%\mathtt{o}_0, \%\mathtt{o}_1, \%\mathtt{o}_2, \%\mathtt{l}_7\})$$
$$*\mathbf{WinSt}(w_{id}, v_{id}, F \circ \mathrm{fm}_1 \circ \mathrm{fm}_2, \{w_{id}, \mathbf{pre\_cwp}(w_{id})\})$$

$$p_2 \triangleq \%\mathtt{o}_0 \mapsto 1 * \%\mathtt{o}_1 \mapsto 2 * \%\mathtt{o}_2 \mapsto \_ * \%\mathtt{l}_7 \mapsto \_$$
$$*\mathbf{Rs}(\{\mathtt{r}_0, \ldots \mathtt{r}_{31}\} \backslash \{\%\mathtt{o}_0, \%\mathtt{o}_1, \%\mathtt{o}_2, \%\mathtt{l}_7\})$$
$$*\mathbf{WinSt}(w_{id}, v_{id}, F \circ \mathrm{fm}_1 \circ \mathrm{fm}_2, \{w_{id}, \mathbf{pre\_cwp}(w_{id})\})$$

$$p_3 \triangleq \%\mathtt{o}_0 \mapsto 6 * \%\mathtt{o}_1 \mapsto \_ * \%\mathtt{o}_2 \mapsto \_ * \%\mathtt{l}_7 \mapsto \_$$
$$*\mathbf{Rs}(\{\mathtt{r}_0, \ldots \mathtt{r}_{31}\} \backslash \{\%\mathtt{o}_0, \%\mathtt{o}_1, \%\mathtt{o}_2, \%\mathtt{l}_7\})$$
$$*\mathbf{WinSt}(w_{id}, v_{id}, F \circ \_ \circ \_, \{w_{id}, \mathbf{pre\_cwp}(w_{id})\})$$

Fig. 18. Example for Another Call Convention in SPARCv8

Theorem 4.3 (Soundness of Well-formed Instruction).

$$\vdash \{p\}\, \mathtt{i}\, \{q\} \Longrightarrow \models \{p\}\, \mathtt{i}\, \{q\}$$

The soundness definition of well-formed instruction sequence is a bit complex. It means that for any code heap C, if we can extract instruction sequence $\mathbb{I}$ from C beginning with label $\mathtt{f}$, and there is a state S satisfying the precondition p, then the safety instruction sequence judgment $\mathsf{safety\_insSeq}\,(C, S, \mathtt{f}, \mathtt{f} + 4, q, \Psi)$ holds.

Definition 4.4 (Semantics of Well-formed Instruction Sequence).

$$\Psi \models \{(p, q)\}\ \mathtt{f} : \mathbb{I} \triangleq \forall C, S.$$
$$C[\mathtt{f}] = \mathbb{I} \rightarrow S \models p \rightarrow \mathsf{safety\_insSeq}\,(C, S, \mathtt{f}, \mathtt{f} + 4, q, \Psi)$$

The meaning of $\mathsf{safety\_insSeq}\,(C, S, \mathrm{pc}, \mathrm{npc}, q, \Psi)$ can be understand approximately as a instruction sequence $\mathbb{I}$ extracted from code heap C starting with pc and npc can execute safely from a initial state S and the predicate q will hold in final state if it ends with a instruction `retl`. The safety instruction sequence judgment[2] shown in Def. 4.5 makes sure the progress and preservation of program. Here, "$\longmapsto^2$" means executing two steps.

Definition 4.5 (Instruction Sequence Execution Safety). The relation $\mathsf{safety\_insSeq}\,(C, S, \mathrm{pc}, \mathrm{npc}, q, \Psi)$ has six parameters : the code heap C, current state S, pc, npc, postcondition q, and code specification Ψ. It tells us the following things :

- if $C(\mathrm{pc}) = \mathtt{i}$ then :
  - there exists $S', \mathrm{pc}, \mathrm{npc}'$, such that $C \vdash (S, \mathrm{pc}, \mathrm{npc}) \longmapsto (S', \mathrm{pc}', \mathrm{npc}')$,
  - for any $S', \mathrm{pc}', \mathrm{npc}'$, if $C \vdash (S, \mathrm{pc}, \mathrm{npc}) \longmapsto (S', \mathrm{pc}', \mathrm{npc}')$, then $\mathsf{safety\_insSeq}\ (C, S', \mathrm{pc}', \mathrm{npc}', q, \Psi)$
- if $C(\mathrm{pc}) = \mathtt{jmpl\ a\ r}_d$ then :
  - there exists $S', \mathrm{pc}', \mathrm{npc}'$, such that $C \vdash (S, \mathrm{pc}, \mathrm{npc}) \longmapsto^2 (S', \mathrm{pc}', \mathrm{npc}')$
  - for any $S', \mathrm{pc}', \mathrm{npc}'$, if $C \vdash (S, \mathrm{pc}, \mathrm{npc}) \longmapsto^2 (S', \mathrm{pc}', \mathrm{npc}')$, then there exists $\mathrm{fp}, \mathrm{fq}, \iota$ and p_r, such that the following holds :
    - $\mathrm{pc}_2 \in \mathrm{dom}(\Psi),\ \Psi(\mathrm{pc}_2) = (\mathrm{fp}, \mathrm{fq}),\ \mathrm{npc}_2 = \mathrm{pc}_2 + 4$
    - $S_2 \models (\mathrm{fp}\ \iota) * p_r,\ (\mathrm{fq}\ \iota) * p_r \Rightarrow q.$
- if $C(\mathrm{pc}) = \mathtt{call\ f}$ then :
  - there exists $S', \mathrm{pc}', \mathrm{npc}'$, such that $C \vdash (S, \mathrm{pc}, \mathrm{npc}) \longmapsto^2 (S', \mathrm{pc}', \mathrm{npc}')$.
  - for any S', pc' and npc', if $C \vdash (S, \mathrm{pc}, \mathrm{npc}) \longmapsto^2 (S', \mathrm{pc}', \mathrm{npc}')$, then there exists $\mathrm{fp}, \mathrm{fq}, \iota$ and p_r, such that the following holds :
    - $\mathrm{pc}' = \mathtt{f}, \mathrm{npc}' = \mathtt{f} + 4,$
    - $\mathtt{f} \in \mathrm{dom}(\Psi),\ \Psi(\mathtt{f}) = (\mathrm{fp}, \mathrm{fq}),$
    - $S' \models (\mathrm{fp}\ \iota) * p_r,$
    - for any S', if $S' \models (\mathrm{fq}\ \iota) * p_r$, then $\mathsf{safety\_insSeq}\ (C, S', \mathrm{pc}{+}8, \mathrm{pc}{+}12, q, \Psi),$
    - for any S', if $S' \models (\mathrm{fq}\ \iota)$, then $S'.Q.R(\mathtt{r}_{15}) = \mathrm{pc}.$

  .....
- if $C(\mathrm{pc}) = \mathtt{retl}$ then :
  - there exists $S', \mathrm{pc}', \mathrm{npc}'$, such that $C \vdash (S, \mathrm{pc}, \mathrm{npc}) \longmapsto^2 (S', \mathrm{pc}', \mathrm{npc}')$
  - for any S', pc' and npc', if $C \vdash (S, \mathrm{pc}, \mathrm{npc}) \longmapsto^2 (S', \mathrm{pc}', \mathrm{npc}')$, then there exists $\mathrm{fp}, \mathrm{fq}, \iota$ and p_r,
    - $S_2 \models q$
    - $\mathrm{pc}_2 = S_2.Q.R(\mathtt{r}_{15}) + 8, \mathrm{npc}_2 = S_2.Q.R(\mathtt{r}_{15}) + 12$

Now, we can present our soundness theorem of instruction sequence just as following :

Theorem 4.6 (Soundness of Well-formed Instruction Sequence).

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

[2] We just list some selected cases of safety_insSeq because of space limitation. More details are presented in Coq implementation [**?**]

The well-formed code heap judgment tells us that we call a code heap well-formed only if each code block in code heap given specification satisfies well-formed instruction sequence. So the soundness theorem of well-formed code heap can be presented simply as :

Definition 4.7 (Well-formed Code Heap Judgment).

$$\begin{aligned}\Psi \models C : \Psi' \triangleq \forall\, \mathtt{f}, \mathrm{fp}, \mathrm{fq}, \iota, S.\\ (\mathrm{f} \in \mathrm{dom}(\Psi') \wedge \Psi'(\mathtt{f}) = (\mathrm{fp}, \mathrm{fq}) \wedge S \models (\mathrm{fp}\ \iota) \wedge \Psi \subseteq \Psi') \rightarrow\\ \Psi \models \{(\mathrm{fp}\ \iota, \mathrm{fq}\ \iota)\}\ \mathtt{f} : C[\mathtt{f}]\end{aligned}$$

Theorem 4.8 (Soundness of Well-formed Code Heap).

$$\Psi \vdash C : \Psi' \Longrightarrow \Psi \models C : \Psi'$$

Besides the above works, there is still a important problem remaining to solve. The specification (p, q) of a instruction sequence $\mathbb{I}$ contains a precondition p specifying the starting state of the execution of $\mathbb{I}$, and postcondition q describing the state at the return point of the current function. We all know that assembly programs are lack of structure, and a function in assembly code usually has multiply segments. So, for a given function f and its specification (p, q), how can we ensure that its execution will reach a state satisfying postcondition q from a state that precondition p holds, if each segment composing it is verified?

In order to solve this problem, we first define a relation $\mathsf{safe}^n(C, S, \mathrm{pc}, \mathrm{npc}, q, k)$ to describe the restrictions to obey in a process of a program execution below.

Definition 4.9 (Program Execution Safety). The $\mathsf{safe}^n(C, S, \mathrm{pc}, \mathrm{npc}, q, k)$ relation tells us that the program can execute n steps safely from pc, npc and state S. It has six parameters that index n recording the steps that the program can execute safely, code heap C, current state S, postcondition q, and the depth of function call k.

a) $\mathsf{safe}^0\,(C, S, \mathrm{pc}, \mathrm{npc}, q, k)$ always holds.
b) $\mathsf{safe}^{n+1}\,(C, S, \mathrm{pc}, \mathrm{npc}, q, k)$ holds if and only if :
   1. if $C(\mathrm{pc}) = \{\mathtt{i}, \mathtt{jmpl\ a\ r}_d, \mathtt{be\ a}\}$ then :
      - there exists $S', \mathrm{pc}', \mathrm{npc}'$, such that $C \vdash (S, \mathrm{pc}, \mathrm{npc}) \longmapsto (S', \mathrm{pc}', \mathrm{npc}')$
      - $\forall S', \mathrm{pc}', \mathrm{npc}'.$
        $C \vdash (S, \mathrm{pc}, \mathrm{npc}) \longmapsto (S', \mathrm{pc}', \mathrm{npc}') \Rightarrow \mathsf{safe}^n\,(C, S', \mathrm{pc}', \mathrm{npc}', q, k)$
   2. if $C(\mathrm{pc}) = \mathtt{call\ f}$ then :
      - there exists $S', \mathrm{pc}', \mathrm{npc}'$, such that $C \vdash (S, \mathrm{pc}, \mathrm{npc}) \longmapsto^2 (S', \mathrm{pc}', \mathrm{npc}')$
      - for any $S', \mathrm{pc}', \mathrm{npc}'$, if $C \vdash (S, \mathrm{pc}, \mathrm{npc}) \longmapsto^2 (S', \mathrm{pc}', \mathrm{npc}')$,
        then $\mathsf{safe}^n\,(C, S', \mathrm{pc}', \mathrm{npc}', q, k+1)$
   3. if $C(\mathrm{pc}) = \mathtt{retl}$ then :
      - there exists $S', \mathrm{pc}', \mathrm{npc}'$, such that $C \vdash (S, \mathrm{pc}, \mathrm{npc}) \longmapsto^2 (S', \mathrm{pc}', \mathrm{npc}')$
      - for any $S', \mathrm{pc}', \mathrm{npc}'$, if $C \vdash (S, \mathrm{pc}, \mathrm{npc}) \longmapsto^2 (S', \mathrm{pc}', \mathrm{npc}')$, then
        if $k = 0$ then
        $S' \models q$
        else
        $\mathsf{safe}^n\,(C, S', \mathrm{pc}', \mathrm{npc}', q, k-1)$

Informally, the relation $\mathsf{safe}^n\,(C, S, \mathrm{pc}, \mathrm{npc}, q, k)$ requires the following hold for each step of program execution :

- $\mathsf{safe}^0\,(C, S, \mathrm{pc}, \mathrm{npc}, q, k)$ always holds because the program does not need to execute.
- If $\mathsf{safe}^{n+1}\,(C, S, \mathrm{pc}, \mathrm{npc}, q, k)$ holds, we can get the following information:
  - If the current instruction is a simple instruction `i`, or control transfer instruction (excluding `call` and `retl`), then the program can execute one step. And if the program executes one step and reach a state $(C, S', \mathrm{pc}', \mathrm{npc}')$, then we get $\mathsf{safe}^n\,(C, S, \mathrm{pc}', \mathrm{npc}', q, k)$ holds for the following execution.
  - If the current instruction is `call`, then the program can execute two steps. Supposing the program state will become $(C, S', \mathrm{pc}', \mathrm{npc}')$ after two steps, $\mathsf{safe}^n(C, S', \mathrm{pc}', \mathrm{npc}')$ holds for remaining execution. The value of k which records the level of function call increases one.
  - If the current instruction is `retl`, the program can execute two steps, and postcondition q will hold after two steps if the depth of function call is zero. Otherwise, the program can execute continuously with the level of function decreasing one.

With the safety judgment relation, we present an important corollary below. We suppose there is a function f. So, this corollary tells us that if the state S of starting point satisfies the precondition p of function f, and each code blocks composing the function f verified, then we can know that the postcondition q holds at the return point of function f.

Corollary 4.10 (Well-formed Function). If $\Psi \models \{(p, q)\}\,\mathrm{pc} : C[\mathrm{pc}]$, $S \models p$, $\Psi \subseteq \Psi'$ and $\Psi \models C : \Psi'$, then $\forall n.\ \mathsf{safe}^n(C, S, \mathrm{pc}, \mathrm{pc} + 4, q, 0)$.

The proof of Corollary 4.10 can be achieved by induction on index n and discussing each case of current instruction. Most cases are intuitive, except `call` instruction. The instruction `call` will change the depth of function call levels from zero to one. So we can't use the induction hypothesis directly, because the function call level presented in induction hypothesis is zero. So we need to prove the following properties first.

Lemma 4.11. If $\mathsf{safety\_insSeq}\,(C, S, \mathrm{pc}, \mathrm{npc}, q, \Psi)$, $\Psi \models C : \Psi'$, $q \Rightarrow \mathtt{r}_{15} = \mathtt{f}$, $(\forall S'.\ S' \models q \rightarrow \mathsf{safe}^n(C, S', \mathtt{f}+8, \mathtt{f}+12, q', k))$, then $\mathsf{safe}^n(C, S, \mathrm{pc}, \mathrm{npc}, q', k+1)$ holds.

Lemma 4.11 plays a similar role with well-formed stack presented in SCAP [**?**]. Supposing a current function f, it says that if the current function f can execute safely from the initial state S and the program can resume executing safely after f returns. So, we can make a conclusion that the program can execute safely from state S.

5 Verifying a Realistic Context Switch Module

We apply our program logic to verify a realistic context switch module implemented in SPARCv8. Fig. 19 shows the structure of the program we verified. And we describe the function of a part of them informally.

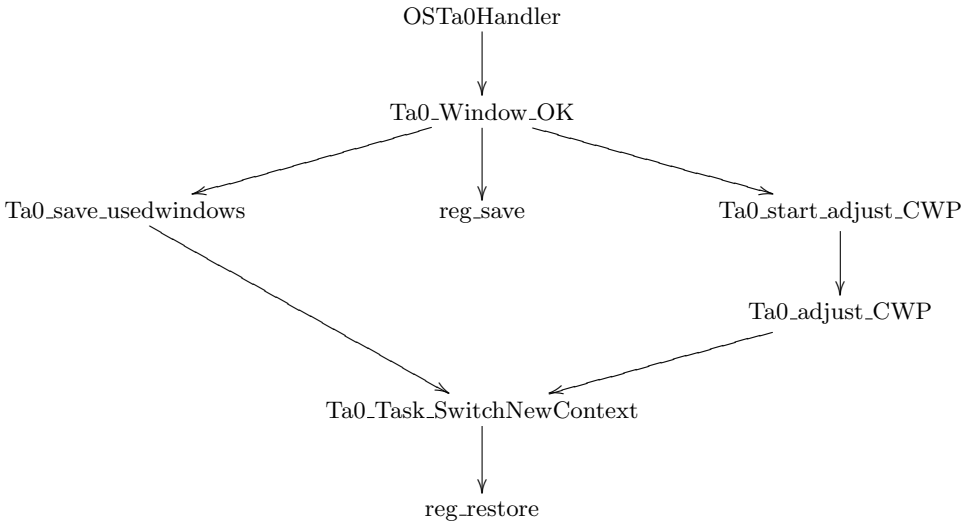

Fig. 19. The Structure of Context Switch Module

- *OSTa0Handler* is the entry of context switch module. It checks whether there is a requirement to do a context switch by SwitchFlag. If there is no need to do a context switch (SwitchFlag = 0), the program exits directly. If a context switch is required, it entries code block Ta0_Window_OK for further executions.
- *Ta0_Window_OK* is the entry to save current task contexts. It first judges whether the current task is null. It jumps to code block Ta0_start_adjust_CWP if the the current task is null. Otherwise, it calls function reg_save to store the current window into current task's TCB. Just saving the current context is not enough. The program will entries code block Ta0_save_usedwindows for register windows saving.
- *Ta0_save_usedwindows* is used to save the context stored in register windows temporarily into current task's stack.
- *Ta0_Task_Switch_NewContext* is responsible for restoring the context stored in new task's TCB and the other context saved in the top of the new task's stack. Then it changes the new task as the current.
- *reg_save* and *reg_restore* are functions that save the current context to old task's TCB and restore the context stored in new task's TCB.

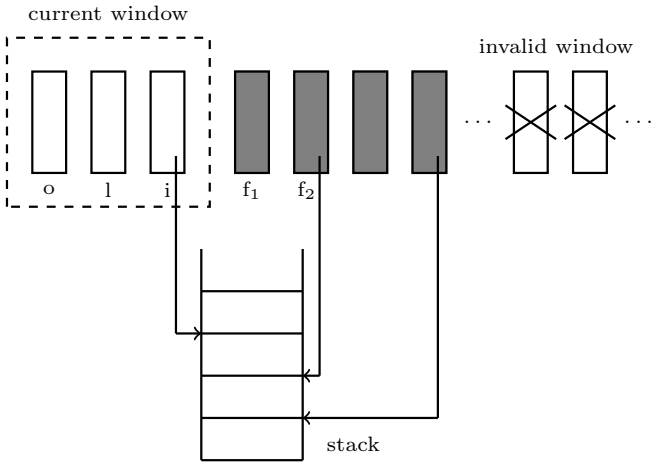

Fig. 20. Stack Windows Constraint

Challenge. Most of works for the context switch module we verified are similar with traditional ones, like saving the current task's context and restoring the new task's context. However, the register windows mechanism in SPARCv8 brings much more difficulties than verifying a context switch module implemented by other assembly code. We have introduced that the SPARCv8 uses register windows to achieve more efficient for context management. Register windows play a role as a temporary stack. SPARCv8 choses to save context by window rotation instead of put current context into stack directly, but the limitation of the numbers of windows causes that the SPARCv8 can't save context by window rotation all the time, so the stack is still required here. For example, if there is a requirement to save the current context, and a window overflow trap occurs. The trap handler will save the oldest elements of the window into the stack and makes the window available for context storing.

Because of the existence of register windows, context switch module not only has to save the current context, but also have to push the contexts temporarily stored in register windows into stack. Fig. 20 shows a relationship between register windows and stack. In Fig. 20, every two shaded frames present a context which have been saved temporarily in frame list. Each context stored temporarily in frame list records a pointer that points to a stack frame, which is reserved for contexts saving, in stack where it will be stored. The size of stack frame is 64 bytes.

The code block Ta0_save_usedwindows (shown in Fig. 22) is responsible for storing the contexts saved in register windows temporarily into stack. It checks whether the next window is invalid (line 1 $\sim$ line 4). The value of special register `wim` which marks whether a window is valid is saved in $\%\mathtt{g}_7$. As for $\%\mathtt{g}_4$, supposing the identity of the current window is w_{id}, the 0 to (OS_WINS - 1) bits of general register $\%\mathtt{g}_4$ is always equal to $2^{w_{id}}$ before program shown in Fig. 22

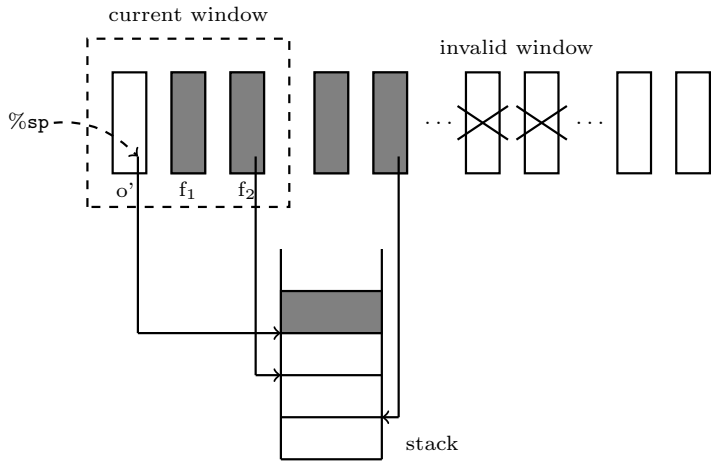

Fig. 21. Saving Context Stored in Frame List by Window Rotation

execution. In SPARCv8 program, we usually set a window invalid and view it as a stack bottom or a top of stack to avoid overflow or underflow as shown in Fig. 20. We can see that marking a window invalid is necessary, because the register windows is a cycle (as shown in Fig. 5) and we will not judge whether a window overflow/underflow exception will occur without the help of the invalid window. Here, in context switch module, the invalid window plays as a role of stack bottom. If the next window is invalid, judged by logical instruction `andcc` at line 4, it means that all the contexts saved in register windows temporarily have been stored into stack and we can jump to code block Ta0_Task_Switch_NewContext to restore the new task's context (at line 5). Otherwise, we use instruction `restore` to rotate to next window (at line 7) and store the window's local and in registers into stack (line 8 $\sim$ line 23). For example, in Fig. 20, the execution of instruction `restore` will let frame f_1 and f_2 become the local and in registers of current window as shown in Fig. 21. Then, they will be stored into a stack frame where the %`sp` (an alias of $\mathtt{r}_{14}$) points (the shaded part of stack in Fig. 21). Then, we jump to the head of Ta0_save_usedwindows to repeat the work. We can see that the precondition of code block Ta0_save_usedwindows is presented as a loop invariant.

Invariant for Register Windows Saving. We have to capture the invariant in the execution of Ta0_save_usedwindows whose implementation has been shown in Fig. 22, in order to do verification. The constant OS_NWINS present the number of register windows. The value of special register `wim`, whose bits mark whether a window is valid, is saved in general register $\%\mathtt{g}_7$ and will not be updated in execution, so the value of $\%\mathtt{g}_7$ is a constant. When we enter Ta0_save_usedwindows at the first time, the value of general register $\%\mathtt{g}_4$ is equal to the identity of current window. We show the invariant $I(\mathrm{ow})$ below. We need

```
Ta0_save_usedwindows :
1     sll      %g4, 1, %g5
2     srl      %g4, OS_WINS − 1, %g4
3     or       %g4, %g5, %g4
4     andcc    %g5, %g7, %g0
5     bnz      Ta0_Task_Switch_NewContext
6     nop
7     restore
8     st       %l0, [%sp + 0]
      ...
23    st       %i7, [%sp + 60]
24    jmpl     Ta0_save_usedwindows, %g0
25    nop
```

Fig. 22. Ta0_save_usedwindows Code

to describe the state of general register, frame list and current task's stack in invariant. The parameter ow presents the initial state of register windows when entering the code block Ta0_save_usedwindows at the first time. It records the contents of general registers $(\mathrm{fm}_g, \mathrm{fm}_0, \mathrm{fm}_l, \mathrm{fm}_i)$ which is composed by global, out, local and in registers, and the state of register windows (w_{oid}, v_i, F) which includes the current window's identity w_{oid}, identity of invalid window v_i, and frame list F. Similarly, we use cw to present the current state of register windows. The definitions used in the invariant are shown in Fig. 23.

$$
\begin{aligned}
I(\mathrm{ow}) \triangleq\ & \exists\ \mathrm{fm}'_g, \mathrm{fm}'_o, \mathrm{fm}'_l, \mathrm{fm}'_i, w_{id}, F', p, l_1, l_2. \\
& \mathrm{R}_g = \mathrm{fm}'_g * \mathrm{R}_o = \mathrm{fm}'_o * \mathrm{R}_l = \mathrm{fm}'_l * \mathrm{R}_i = \mathrm{fm}'_i * \mathbf{Fs}(w_{id}, v_i, F') * \mathrm{stack}(cstk) \\
& * \langle \mathbf{SFInv}(\mathrm{ow}, \mathrm{cw}, cstk) \wedge \mathbf{CWPInv}(\mathrm{ow}, \mathrm{cw}) \rangle \\
\text{where } \mathrm{ow} =\ & ((\mathrm{fm}_g, \mathrm{fm}_o, \mathrm{fm}_l, \mathrm{fm}_i),\ (w_{oid}, v_i, F)), \\
\mathrm{cw} =\ & ((\mathrm{fm}'_g, \mathrm{fm}'_o, \mathrm{fm}'_l, \mathrm{fm}'_i),\ (w_{id}, v_i, F')),\ cstk = (p, l_1 \cdot l_2).
\end{aligned}
$$

In the invariant, we use assertion $\mathrm{stack}(cstk)$ to present the state of stack, where $cstk$ is equal to $(p, l_1 \cdot l_2)$. Here, p presents the address of stack top and $l_1 \cdot l_2$ is the contents of stack. The part of stack used to save the register windows can be split in two parts l_1 and l_2. The list l_1 records the part of spaces in stack whose corresponding register windows have already been stored. And list l_2 present the spaces waiting for saving register windows. The l_1 and l_2 are both list of frame pairs. We use a pair of frames to present a stack frame, because each windows local and in registers compose a context, and the program saves thm into a corresponding stack frame at each time.

SFInv and **CWPInv** are pure properties in loop invariant. $\mathbf{SFInv}(\mathrm{ow}, \mathrm{cw}, cstk)$ tells us the following things :

- $\mathbf{sf\_match}(w_{oid}, l_1, F \circ \mathrm{fm}_o, w_{id})$ presents that the list l_1 are equal to contents of register windows that have been saved. The identities of saved windows are from w_{oid} to w_{id}.
- $\mathbf{sf\_pt}(p - 64 \times |l_1|, \mathrm{fm}'_l :: \mathrm{fm}'_i :: F' \circ \mathrm{fm}'_o, l_2, w_{id}, v_i)$ describes that the rest register windows unsaved still satisfy the restriction shown in Fig. 20. The iden-

$$\mathrm{R}_g = [w_0, \ldots, w_7] \triangleq \mathtt{r}_0 \mapsto w_0 * \cdots * \mathtt{r}_7 \mapsto w_7$$

$$\begin{aligned}\mathbf{Fs}(w_{id}, v_i, F) \triangleq\ & \{\mathtt{cwp} = w_{id}, F\} * \mathtt{wim} \mapsto 2^{v_i} \\ & * \langle |F| = 2 * \mathrm{OS\_NWINS} - 3 \wedge 0 \leq w_{id}, v_i \leq 7 \rangle\end{aligned}$$

$$\mathrm{stack}(p, l) \triangleq \begin{cases} \mathrm{emp} & l = \mathrm{nil} \\ \{|\mathrm{fm}_1, \mathrm{fm}_2|\}_p * \mathrm{stack}(p - 64, l') & l = (\mathrm{fm}_1, \mathrm{fm}_2) :: l' \end{cases}$$

$$\{|\mathrm{fm}_1, \mathrm{fm}_2|\}_p \triangleq \{|\mathrm{fm}_1|\}_p * \{|\mathrm{fm}_2|\}_{p-32} \qquad \{|\,[w_0, \ldots, w_7]\,|\}_p \triangleq p \mapsto w_0 * \ldots (p + 28) \mapsto w_7$$

$$\begin{aligned}\mathbf{SFInv}(\mathrm{ow}, \mathrm{cw}, \mathit{cstk}) \triangleq\ & \mathbf{sf\_match}(w_{oid}, l_1, F \circ \mathrm{fm}_o, w_{id}) \\ & \wedge \mathbf{sf\_pt}(p - 64 \times |l_1|, l_2, \mathrm{fm}'_l :: \mathrm{fm}'_i :: F' \circ \mathrm{fm}'_o, w_{id}, v_i) \\ & \wedge \mathbf{frestore}(w_{oid}, \mathrm{fm}_o :: \mathrm{fm}_l :: \mathrm{fm}_i :: F', w_{id}, \mathrm{fm}'_o :: \mathrm{fm}'_l :: \mathrm{fm}'_i :: F') \\ \text{where } \mathrm{ow} = & ((\mathrm{fm}_g, \mathrm{fm}_o, \mathrm{fm}_l, \mathrm{fm}_i), (w_{oid}, v_i, F)), \\ \mathrm{cw} = & ((\mathrm{fm}'_g, \mathrm{fm}'_o, \mathrm{fm}'_l, \mathrm{fm}'_i), (w_{id}, v_i, F')),\ \mathit{cstk} = (p, l_1 \cdot l_2)\end{aligned}$$

$$\begin{aligned}\mathbf{CWPInv}(\mathrm{ow}, \mathrm{cw}) \triangleq\ & \exists i.\ [\![\mathrm{fm}_g]\!]^{\mathrm{glb}}_{g_4} = i \wedge [\![\mathrm{fm}_g]\!]^{\mathrm{glb}}_{g_7} = 2^{v_i} \wedge i = \mathbf{rotate}(w_{oid}, w_{id}, v_i) \\ & \wedge ((i = 2^{w_{id}}) \vee (i = 2^{w_{id} + \mathrm{OS\_NWINS}} + 2^{w_{id}})) \\ \text{where } \mathrm{ow} = & ((\mathrm{fm}_g, \mathrm{fm}_o, \mathrm{fm}_l, \mathrm{fm}_i), (w_{oid}, v_i, F)), \\ \mathrm{cw} = & ((\mathrm{fm}'_g, \mathrm{fm}'_o, \mathrm{fm}'_l, \mathrm{fm}'_i), (w_{id}, v_i, F')).\end{aligned}$$

$$\mathbf{sf\_match}(w_{id}, \emptyset, F, v_i) \triangleq w_{id} = v_i$$

$$\begin{aligned}\mathbf{sf\_match}(w_{id}, \mathrm{fp} :: l, F, v_i) \triangleq\ & \exists F', \mathrm{fm}_1, \mathrm{fm}_2.\ w_{id} \neq v_i \wedge \mathrm{fp} = (\mathrm{fm}_1, \mathrm{fm}_2) \\ & \wedge F = \mathrm{fm}_1 :: \mathrm{fm}_2 :: F' \wedge \mathbf{sf\_match}(\mathbf{post\_cwp}(w_{id}), l, F', v_i)\end{aligned}$$

$$\mathbf{sf\_pt}(p, w_{id}, F, l, v_i) \triangleq \mathbf{post\_cwp}(w_{id}) = v_i$$

$$\begin{aligned}\mathbf{sf\_pt}(p, w_{id}, F, \mathrm{fp} :: l, v_i) \triangleq\ & \exists \mathrm{fm}_1, \mathrm{fm}_2, F.\ \mathbf{post\_cwp}(w_{id}) \neq v_i \wedge \mathrm{fm}_i[6] = p \\ & \wedge F = \mathrm{fm}_1 :: \mathrm{fm}_2 :: F' \wedge \mathbf{sf\_pt}(p - 64, \mathbf{post\_cwp}(w_{id}), F', l, v_i)\end{aligned}$$

$$\mathbf{frestore}(w_{id}, F, w_{id}, F') \triangleq F = F'$$

$$\begin{aligned}\mathbf{frestore}(w_{oid}, F, w_{id}, F') \triangleq\ & \exists F'',\ \mathrm{fm}_1, \mathrm{fm}_2.\ w_{oid} \neq w_{id} \wedge F' = F'' \circ \mathrm{fm}_1 \circ \mathrm{fm}_2 \\ & \wedge \mathbf{frestore}(w_{oid}, F, \mathbf{pre\_cwp}(w_{id}), \mathrm{fm}_1 :: \mathrm{fm}_2 :: F'')\end{aligned}$$

$$\mathbf{rotate}(w_{oid}, w_{id}, v_i) \triangleq \begin{cases} 2^{w_{id}} & \mathbf{if}\ w_{oid} = w_{id} \\ (i >> (\mathrm{OS\_WINS} - 1)) \\ \quad |\,(i << 1) & \mathbf{if}\ w_{oid} \neq w_{id}, \\ & \quad i = \mathbf{rotate}(w_{oid}, \mathbf{pre\_cwp}(w_{id}), v_i) \end{cases}$$

Fig. 23. Definition Used in Invariant

tities of unsaved windows are from w_{id} to v_i. The addresses of stack frames are decreasing in stack from top to bottom. So, the addresses of places preserved for unsaved register windows in stack are starting from $p - 64 \times |l_1|$. Here, the size of a stack frame is 64 bytes, and $|l_1|$ presents the length of list l_1.

– $\mathbf{frestore}(w_{oid}, \mathrm{fm}_o :: \mathrm{fm}_l :: \mathrm{fm}_i :: F, w_{id}, \mathrm{fm}'_o :: \mathrm{fm}'_l :: \mathrm{fm}'_i :: F')$ makes sure that the current state of register windows can be got by window rotation from the initial one. Here the contents of initial register windows can be presented as $\mathrm{fm}_o :: \mathrm{fm}_l :: \mathrm{fm}_i :: F$ and the current one is $\mathrm{fm}'_o :: \mathrm{fm}'_l :: \mathrm{fm}_i :: F'$.

CWPInv describes the invariant property of the value of $\%\mathtt{g}_4$. Supposing the current value of $\%\mathtt{g}_4$ is i and the identity of current window is w_{id} before this time's execution, $\mathbf{rotate}(w_{oid}, w_{id}, v_i)$ tells us how to get i from the initial one. By the program shown in Fig.22, we know that the value of $\%\mathtt{g}_4$ will be updated as $(i >> (\text{OS\_WINS} - 1) \,|\, (i << 1))$, where the $|$ is the or operation. And we can also find that i is equal to either $2^{w_{id}}$ or $2^{w_{id}+\text{OS\_NWINS}}$.

We omit the operations about interrupt and float registers in verification work, because our current work doesn't consider them. The context switch module we verified is about 253 lines, and we prove it by 4096 lines Coq proof scripts. The ratio of Coq proof scripts and Sparc code is around 16:1.

6 Related Work and Conclusion

There are many impressive pioneering works that focus on machine code verification. Projects on proof-carrying code (PCC) [**?**, **?**] and typed assembly language (TAL) [**?**, **?**] evoke interests in low-level code verification. They aim to find a method to address safety properties of assembly code by automatic type-checking. Generating proofs automatically is difficult in many times. So, the work [**?**] of Yu *et al.* provides the CAP framework to support a Hoare-logic style reasoning for assembly code. However, works on PCC, TAL and CAP only focus on the safety properties of assembly code.

Certifying function call/return in assembly code is difficult, because assembly codes are usually lacking structure. Feng *et al.* develop SCAP framework [**?**] for verification of assembly code with all kinds of stack-based control abstraction, including function call/return. Our method to handle function call/return is inspired by SCAP. However, most works on CAP, including SCAP, are based on a simply abstract machine model and establish soundness following the syntactic approach of proving type sound. And they also do not give the frame rule for local verification. Myreen's work [**?**] presents a framework for ARM verification based on a realistic model and can state the functional behavior of ARM program, but doesn't support function call/return verification.

Following the CAP framework, Ni *et al.* present XCAP [**?**] to solve embedded code pointers (ECPs) problem in assembly code level, and Feng *et al.* extend the basic CAP to handler concurrent multi-threaded assembly code verification. Our current work is not support ECPs and concurrency verifications [**?**, **?**, **?**]. So, these work can be the complementaries in our future work.

The existent works about SPARC are less. Wang *et al.* [**?**] formalize the SPARCv8 ISA in Coq. Their work does not develop a program logic and uses the operational semantics for code reasoning directly. Tan and Apple's work [**?**] which is a part of Foundational Proof-Carrying Code (FPCC) [**?**] presents a program logic about reasoning unstructured control flow in machine-language program. Although their logic is implemented on the top of SPARC machine language and provides fine-grained composition rules to compose program segment, the main target of their work is not to design a program logic specially for SPARC code. They do not consider the main features of SPARC in their logic and still treat

function call and return as the first-class function.

We are not the first one to verify a context switch module that Ni *et al.* have used the XCAP framework to certify a machine context management in x86 [**?**]. However, the programs they proved are implemented by them and context switch module is only 19 lines in their work. We prove a realistic context switch module which is more complex and considers more details and cases for actual execution. Although we does not consider ECPs problem which XCAP can handler well, it does not influence the context switch module that we verified to link with other modules, because it's a function and our logic supports function call/return verification.

Conclusion. We present a program logic for SPARC code verification. Our logic is based on realistic and accurate models of SPARC program, and supports module, local, function call/return and main features of SPARC verification. We also give a semantic approach for soundness establishing and prove the soundness of our logic. We finally apply it to verify a context switch module which has been used in an actual embedded OS kernel. The current work can only handler sequential SPARC program verification for partial correctness, and we will extend our work for concurrency and refinement verification in the future.