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 kernels are at the most foundational layer of computer software systems. To interact directly with hardware, many important components in OS kernels are implemented in assembly, such as the context switch code or the code that manages interrupts. Their correctness is crucial to ensure the safety and security of the whole system. However, assembly code verification remains a challenging task in existing work on OS kernel verification (*e.g.* [18, 8, 7]), where the assembly code is either unverified or verified based on operational semantics without a general program logic.

SPARC (Scalable Processor ARChitecture) is a CPU instruction set architecture (ISA) with high-performance and great flexibility [2]. It has been widely used in various processors for workstations and embedded systems. The SPARCv8 ISA has some interesting features, which makes it a non-trivial task to design a Hoare-style program logic for assembly code.

- *Delayed Control-Transfer.* SPARCv8 has two program counters `pc` and `npc`. The `npc` register points to the next instruction. Control-transfer instructions in SPARCv8 change `npc` instead of `pc` to the target program point and cause the control transfer to happen one cycle after executing the control transfer instruction.

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  retl
11  restore  %l0, 0, %o0
```

Fig. 1. An Example for SPARC Code

- *Delayed-writes.* The `wr` instruction that writes special registers is not executed immediately in the current cycle when pointed to `pc`. Instead the write operation is buffered and executed N cycles later, where N is a global parameter which usually ranges from 0 to 3.
- *Register Windows.* SPARCv8 uses register windows and window rotation mechanism to avoid saving contexts in the stack directly and achieves high performance and a significant reduction in context management.

We use a simple example (see Fig. 1) to show 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 an input parameter as the new value for the special register `Y`. The **CALLER** calls function **ChangeY** at line 2, and `pc` and `npc` point to line 2 and 3 respectively at this moment. The call instruction changes 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 next cycle but in the cycle after the execution of the `save` instruction following the call. Similarly, when **ChangeY** returns (at line 10), the control is transferred back to the caller after executing the `restore` instruction at line 11. We call this feature "delayed control-transfer".

SPARCv8 uses the `save` instruction (at line 3 in the example) to save the current context and `restore` (at line 10) to restore it. A total of 32 general registers are split into four logic groups 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. Local registers are for private use in the current context. In and out registers are shared with adjacent register windows for parameters passing. The `save` instruction rotates the register window from the current one to the next. Then the *local* and *in* registers in the original window are no longer accessible, and the original *out* registers becomes the *in* registers in the current window. The `restore` instruction does the inverse. The arguments taken by the `save` and `restore` instructions are irrelevant here and can be ignored.

At line 6, the `wr` instruction tries to update the special register `Y` with the value of $\%\mathtt{i}_0 \oplus 0$. However, the write operation is delayed for N cycles, where N is some predefined parameter that ranges from 0 to 3. Suppose $N = 3$, then it does not take effect until the completion of the third `nop` instruction at line

9. Reading of `Y` earlier than line 9 can only give us the old value. This feature is called "delayed-write". For compatibility, programmers usually do not rely on the exact value of N and assume it always takes the maximum value (3).

These unique features make the semantics of the SPARCv8 code context-dependent. For instance, a read of a special register (*e.g.* the register `Y` in the above example) needs to make sure there are enough instructions executed since the most recent *delayed* write. As another example, the instruction following the `call` can be any instruction in general, but it is not supposed to update the register $\mathtt{r}_{15}$, which contains the return address saved by the `call` instruction. In addition, the delayed control transfer and the register windows also allows highly flexible calling conventions. Together, they make it a challenging task to have a Hoare-style program logic for local and modular reasoning of SPARCv8 assembly code.

Working towards a fully certified OS kernel for aerospace crafts whose inline assembly is written in SPARCv8, we try to address these challenges and propose a practical program logic for realistically modelled SPARCv8 code. We have applied our logic to verify the task context switch module in the kernel. Our work is based on earlier work on assembly code verification but makes the following contributions:

- Our logic supports all the above features of SPARCv8. We redefine basic blocks to include the instruction following the jump or return as the tail of a basic block, which models the delayed control transfer. To reason about delayed writes, we introduce a modal assertion $\triangleright_n p$, saying that p becomes true in up to n cycles. In particular, $\triangleright_n \mathtt{sr} \mapsto w$ means the special register `sr` will hold the value w in up to n cycles. We also give logic rules for `save` and `restore` instructions that do register window rotation.
- Following SCAP [6], our logic supports modular reasoning of function calls in a direct-style. We use the standard pre- and post-conditions as function specifications, instead of the binary assertion g used in SCAP. This allows us to reuse existing techniques (*e.g.* Coq tactics) developed to simplify the program verification process. The logic rules for function call and return is general and independent of any specific calling convention.
- We give semantic-based direct-style interpretation for the logic judgments, based on which we establish the soundness of our logic. This is different from previous work, which either does syntactic-based soundness proof (*e.g.* SCAP [6]) or treats return code pointers as first-class code pointers and gives CPS-style semantics. Those approaches for soundness make it difficult to verify the interaction between the inline assembly and the C code in the kernel, the latter being verified following a direct-style program logic.
- Context switch of concurrent tasks is an important component in OS kernels. It is usually implemented as inline assembly because of the need to access registers and the stack. We verify the context switch module in a realistic embedded OS kernel for aerospace crafts [1], which consists of around 250 lines of SPARCv8 code.

[1] The source code cannot be published due to copyright issues.

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

Fig. 2. Machine States and Language for SPARCv8 Code

The program logic, its soundness proof and the verification of the context switch module have been mechanized in Coq.

In the rest of paper, we present the program model and operational semantics of SPARCv8 in Sec. 2. Then we propose the program logic in Sec. 3, including the inference rules and the soundness proof. We show the verification of the context switch module in Sec. 4. Finally we discuss more on related work and conclude in Sec. 5.

2 The SPARCv8 Assembly Language

We introduce the key SPARCv8 instructions, the model of machine states, and the operational semantics in this section.

2.1 Language syntax and states

The machine model and syntax of SPARCv8 assembly language are defined in Fig. 2. The whole program configuration P consists of the code heap C, the machine state S, and the program counters `pc` and `npc`. The code heap C is a partial function from labels `f` to commands c. Labels are 32-bit integers (called *words*), which can be viewed as memory addresses where the commands are saved. Commands in SPARCv8 can be classified into two categories, the simple instructions `i` and the control-transfer instructions like `call` and `jmp`.

The machine state S consists of three parts: the memory M, the register state Q which is a pair of register file R and frame list F, and the delay buffer D. As defined in Fig. 3, R is a partial map from register names to words. Registers include the general registers `r`, the processor state register `psr` and the special registers `sr`. The processor state register `psr` contains the integer condition code fields `n`, `z`, `v` and `c`, which can be modified by the arithmetic and logical instructions and used for conditional control-transfer, and `cwp` recording the id of the current register window. We explain the frame list F and the delay buffer D below.

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

Fig. 3. Register File, Frame List and DelayList

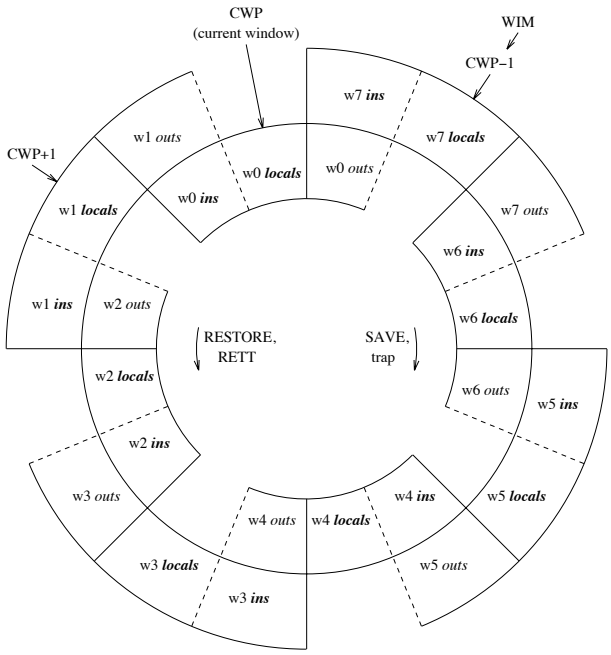


Fig. 4. Register Windows (figure taken from [2])

Register windows and frame List. SPARCv8 provides 32 general registers, which are split into four groups 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. The latter three groups (*out*, *local* and *in*) form the current *register window*.

At the entry and exit of functions and traps, one needs to save and restore some of the general registers as execution contexts. Instead of saving them into stacks in memory, SPARCv8 uses multiple register windows to form a circular stack, and does window rotation for efficient context save and restore. As shown in Fig. 4, there are N register windows ($N = 8$ here) consisting of $2 \times N$ groups of registers (each group containing 8 registers). The `cwp` register (part of `psr`) records the id number of the current window ($\mathtt{cwp} = 0$ in this example).

The *in* and *out* registers of each window are shared with its adjacent windows for parameter passing. 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. This explains why we need only $2 \times N$ groups of registers for N windows, while each window consisting of three groups (*out*, *local* and *in*).

$$\mathsf{out} \triangleq [\mathtt{r}_8, \ldots, \mathtt{r}_{15}] \qquad \mathsf{local} \triangleq [\mathtt{r}_{16}, \ldots, \mathtt{r}_{23}] \qquad \mathsf{in} \triangleq [\mathtt{r}_{24}, \ldots, \mathtt{r}_{31}]$$

$$R([\mathtt{r}_i, \ldots, \mathtt{r}_{i+k}]) \triangleq [R(\mathtt{r}_i), \ldots, R(\mathtt{r}_{i+k})]$$

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

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

$$\mathbf{next\_cwp}(w_{id}) \triangleq (w_{id} + N - 1)\%N \qquad \mathbf{prev\_cwp}(w_{id}) \triangleq (w_{id} + 1)\%N$$

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

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

Fig. 5. Auxiliary Definitions for Instruction `save` and `restore`

To save the context, the `save` instruction rotates the window by decrements the `cwp` pointer (modular N). So w_7 becomes the current window. The *out* registers of the w_0 becomes the *in* registers of the w_7. The *in* and *local* registers of w_0 become inaccessible. This is like pushing them onto the circular stack. The `restore` instruction does the inverse, which is like a stack pop.

We use the register `wim` as a bit vector to record the end of the stack. Each bit in `wim` corresponds to a register window. The bit corresponds to the last available window is set to 1, which means *invalid*. All other bits are 0 (*i.e. valid*). When executing `save` (and `restore`), we need to ensure the next window is valid. We use the assertion $\mathbf{win\_valid}(w_{id}, R)$ defined in Fig. 5 to say the window pointed to by w_{id} is valid, given the value of `wim` in R.

We use the frame list F to model the circular stack consisting of register windows. As defined in Fig. 3, a frame is an array of 8 words, modeling a group of 8 registers. F consists of a sequence of frames corresponding to all the register windows except the *out*, *local* and *in* registers in the current window. Then `save` saves the *local* and *in* registers onto the head of F and loads the two groups of register at the *tail* of F to the *local* and *out* registers (and the original *out* registers becomes the *in* group). The `restore` instruction does the inverse. The operations are defined formally in Fig. 5.

The delay buffer. The delay buffer D is a sequence of delayed writes. Because the `wr` instruction does not update the target register immediately, we put the write operation onto the delay buffer. A delayed write is recorded as a triple consisting of the remaining cycles t to be delayed, the target special register `sr` and the value w to be written.

Instruction sequences. We use an instruction sequence $\mathbb{I}$ to model a basic block, *i.e.* a sequence of commands ending with a control transfer. As defined in Fig. 2, we require that a delayed control-transfer instruction must be followed by a simple instruction `i`, because the actual control-transfer occurs after the execution of `i`. The end of each instruction sequence can only be `jmp` or `retl` followed by a simple instruction `i`. Note that we do not view the `call` instruction as the end of a basic block, since the callee is expected to return, following our direct-style semantics for function calls. We define $C[\mathtt{f}]$ to extract an instruction sequence starting from `f` in C below.

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

2.2 Operational Semantics

We define the operation semantics of SPARCv8 with multiply layers. We just select a part of transition rules in Fig. 6 to give readers a whole recognition for SPARCv8 program execution because of the space limitation. As shown in Fig. 6, 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. The state transition " $\rightrightarrows$ " checks 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 the state transition caused by delay list simply as following. We can find that if a special register `sr` recorded in D isn't in the domain of R. It will not make any changes for R.

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

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

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., `jmp`, `call`, `retl`) are defined in this layer. We can see that delayed control-transfer instructions like `call` will change `npc`

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

(a) Program Transistion

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

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

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

$$\frac{C(\mathtt{pc}) = \mathtt{retl} \quad R(\mathtt{r}_{15}) = \mathtt{f}}{C \vdash ((M, (R, F), D), \mathtt{pc}, \mathtt{npc}) \circ\!\!\longrightarrow ((M, (R, F), D), \mathrm{npc}, \mathtt{f} + 8)}$$
$$\frac{C(\mathtt{pc}) = \mathtt{ret} \quad R(\mathtt{r}_{31}) = \mathtt{f}}{C \vdash ((M, (R, F), D), \mathtt{pc}, \mathtt{npc}) \circ\!\!\longrightarrow ((M, (R, F), D), \mathrm{npc}, \mathtt{f} + 8)}$$

(b) Control Transfer Instruction Transition

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

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

(c) Save, Restore and Wr instruction Transition

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

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

(d) Simple Instruction Transition

Fig. 6. Selected Operational Semantics

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 $\mathtt{f}+8$ (where $R(\mathtt{r}_{15}) = \mathtt{f}$). However, the address used for return for instruction `ret` is saved in $\mathtt{r}_{31}$. The instruction `ret` is used for the situation

that the procedure will execute an instruction `save` and allocate a new window for itself. 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 $\mathbf{set\_delay}(\mathtt{sr}, w, D)$ is defined as following, where X $(0 \leq X \leq 3)$ is the delay cycle whose specific value is implementation dependently :

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

The execution of instruction `save` and `restore` will cause the window rotation, so that will change the state of Frame List F. The operation $\mathbf{save}(R, F)$ 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 register window by $\mathbf{restore}(R, F)$, which will do a left rotation by $\mathbf{left\_win}(R, F)$ for register window. The auxiliary definitions for operational semantics of `save` and `restore` are shown in Fig. 5.

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

3 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.

3.1 Assertion Language

$$\begin{array}{c}\textit{(Asrt)}\ p, q \triangleq \mathrm{emp} \mid l \mapsto w \mid \mathtt{a} =_a w \mid \mathtt{o} = w \mid \mathtt{rn} \mapsto w \mid \rhd_n \mathtt{sr} \mapsto v \mid \mathtt{cwp} \mapsto (\!| w_{id}, F |\!) \mid \\ \langle \mathsf{P} \rangle \mid p \wedge q \mid p \vee q \mid p * q \mid \forall x.\, p \mid \exists x.\, p \mid p{\downarrow} \mid \ldots\end{array}$$

Fig. 7. Syntax of Assertions

Our assertion language is shown in Fig. 7. And the semantics of some of them are presented in Fig. 8.

Assertion emp says that the memory and register file are both empty. $l \mapsto w$ specifies a singleton memory cell with value w stored in address l. $\mathtt{rn} \mapsto w$ says that there is only one cell in register file that maps the register name `rn` to value w and `rn` is not in delay list. We use assertion $\mathtt{cwp} \mapsto (\!| w_{id}, F |\!)$ to describe the state of window registers. Here, the identity of the current window is w_{id} and F

$$
\begin{array}{ll}
S \models \mathrm{emp} & \triangleq S.M = \emptyset \wedge S.Q.R = \emptyset \\
S \models l \mapsto w & \triangleq S.M = \{l \rightsquigarrow w\} \wedge S.Q.R = \emptyset \\
S \models \mathtt{rn} \mapsto w & \triangleq S.Q.R = \{\mathtt{rn} \rightsquigarrow w\} \wedge \mathtt{rn} \notin \mathrm{dom}(S.D) \wedge S.M = \emptyset \\
S \models \rhd_n \mathtt{sr} \mapsto v & \triangleq \exists k, R', D'.\, 0 \le k \le n{+}1 \wedge (R, D) \rightrightarrows^k (R', D') \wedge \\
 & \qquad ((M, (R', F), D') \models \mathtt{sr} \mapsto v) \wedge \mathbf{noDup}(D, \mathtt{sr}) \\
 & \qquad \text{where } S = (M, (R, F), D) \\
S \models \mathtt{cwp} \mapsto (\!| w_{id}, F |\!) & \triangleq (S \models \mathtt{cwp} \mapsto w_{id}) \wedge \exists F'.\, F \cdot F' = S.Q.F \\
S \models \mathtt{a} =_a w & \triangleq [\![\mathtt{a}]\!]_{S.Q.R} = w \wedge \mathbf{word\_align}(w) \\
S \models \mathtt{o} = w & \triangleq [\![\mathtt{o}]\!]_{S.Q.R} = w \\
S \models p\downarrow & \triangleq \exists R', D'.\, (R', D') \rightrightarrows (S.Q.R, S.D) \wedge S[R', D'] \models p \\
S \models p_1 * p_2 & \triangleq \exists S_1, S_2.\, S_1 \models p_1 \wedge S_2 \models p_2 \wedge S = S_1 \uplus S_2 \\
S \models \langle \mathsf{P} \rangle & \triangleq \mathsf{P} \wedge S \models \mathrm{emp}
\end{array}
$$

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

$$
\mathrm{dom}(D) \triangleq \begin{cases} \{\mathtt{sr}\} \cup \mathrm{dom}(D') & \textbf{if } D = (n, \mathtt{sr}, w) :: D' \\ \emptyset & \textbf{if } D = \mathrm{nil} \end{cases}
$$

$$
\mathbf{noDup}(D, \mathtt{sr}) \triangleq \begin{cases} \mathtt{sr} \notin \mathrm{dom}(D') & \textbf{if } D = (n, \mathtt{sr}, w) :: D' \\ \mathtt{sr} \neq \mathtt{sr}' \wedge \mathbf{noDup}(D', \mathtt{sr}) & \textbf{if } D = (n, \mathtt{sr}', w) :: D' \\ \textbf{True} & \textbf{if } D = \mathrm{nil} \end{cases}
$$

$$
S[R', D'] \triangleq (M, (R', F), D'), \quad \text{where } S = (M, (R, F), D)
$$

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

Fig. 8. Semantics of Assertions

is a prefix of the frame list. We have introduced that the frame list can be viewed as a circularly finite stack. and describing its whole content is unnecessary. Just specifying a part of it that saves the information we care about is enough. 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. 3 is integer w, and w is aligned on a 4-byte boundary by $\mathbf{word\_align}(w)$. Assertion $\mathtt{o} = w$ holds if the evaluation of expression $\mathtt{o}$, which is either a general register or a word, is w under the current state. And $\langle \mathsf{P} \rangle$ requires that the current state is emp and proposition P holds.

The most novel assertion here is $\rhd_n \mathtt{sr} \mapsto v$. It describes a cell for special register $\mathtt{sr}$ in register file, but the specific value stored in it is not known currently. 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 $(k, \mathtt{sr}, v)$, where $0 \le k \le n$, 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 by $\mathbf{noDup}(\mathtt{sr}, D)$ in each moment. Because the concrete delayed time is implementation dependently, and programmers 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. Using assertion $\rhd_n \mathtt{sr} \mapsto v$ let us do not have to expose the detail of delay list in our assertion. For example, if there is a delay item $(k, \mathtt{sr}, v)$ in delay list D, we can describe the state of special register as $\rhd_n \mathtt{sr} \mapsto v$ (where $n \geq k$).

We say that the assertion $p\downarrow$ holds on a state S, if there exists a state that can transfer to S by delay list step satisfying p. This assertion plays an important role in our inference rules designing, because of delay list step in each cycle. If predicate p holds before delay list step, the state after transition "$\_ \rightrightarrows \_$" can be described simply just as the predicate $p\downarrow$.

Separation conjunction is the most important assertion in Separation Logic [15]. We define that both memory and register file are partial function in our work. The operation separation conjunction not only splits memory into two disjoint parts, but also register file. It will make our specification more concise, and let us do not have to consider any side condition about aliases. However, readers may confuse that whether there will be some problems, because the separation conjunction doesn't split the delay list and the SPARCv8 has to execute delay list step in each cycle. The following lemma tells that splitting R and executing delay list step on them separately will not influence the result of executing it on a whole one. The reason is that the delay list step will do nothing on R, if the target special register is not in the domain of R.

Lemma 3.1. If $(R, D) \rightrightarrows (R', D')$ and $R = R_1 \uplus R_2$, then there exists R'_1 and R'_2, such that $(R_1, D) \rightrightarrows (R'_1, D')$, $(R_2, D) \rightrightarrows (R'_2, D')$ and $R' = R'_1 \uplus R'_2$ hold.

We can achieve Corollary 3.2 based on the Lemma 3.1. It tells us that we can consider $p_1\downarrow$ and $p_2\downarrow$ separately, if the $(p_1 * p_2)\downarrow$ holds.

Corollary 3.2.

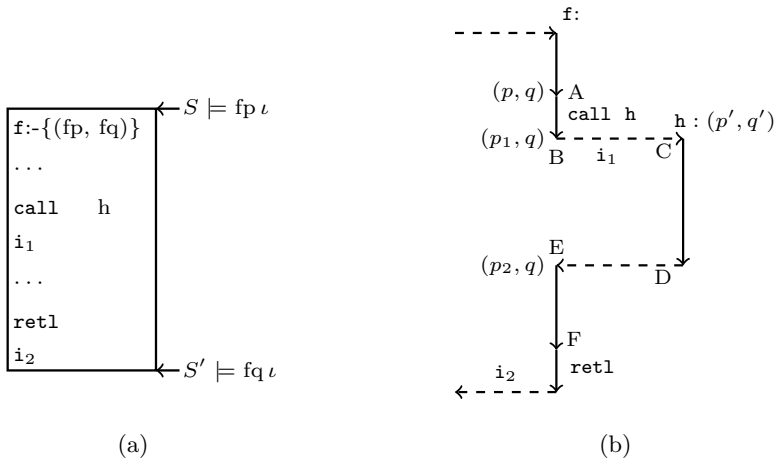
$$(p_1 * p_2)\downarrow \Longrightarrow (p_1)\downarrow * (p_2)\downarrow$$

3.2 Inference Rules

We define a set of inference rules so that we can check SPARCv8 code mechanized. The code specification θ and code heap specification Ψ are defined as blow :

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

In order to get connection between intial state and final state, we define a list of logical variables LogicVL as a parameter of code block's pre/postcondition. A predicate fp specifies the initial state, and the other predicate fq describe the



(a) (b)

Fig. 9. Function Call/Return and Delayed-control-transfer Handling

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

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

Fig. 10. Example for Function Specification

state at the return point of the current function. Fig. 9(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. we give a simple example to introduce how to write a specification for a function implemented by SPARCv8 in Fig. 10.

Example of Function Specification. The function shown in Fig. 10 sums the values of registers $\%\mathtt{i}_0$, $\%\mathtt{i}_1$ and $\%\mathtt{i}_2$ together and writes the result into register $\%\mathtt{l}_7$. The specification of the function is presented by (fp, fq). Here, we need to describe the state of register $\mathtt{r}_{15}$, which saves the label of occurring function call, in pre/postcondition. However, we don't have to know the concrete value of $\mathtt{r}_{15}$ 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 $\mathtt{r}_{15}$

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. 11 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 of the state transition caused by delay list step. The following Lemma 3.3 makes sure that if predicate p satisfies the current state, then $p{\downarrow}$ will hold after delay list step. The proof are straight forward by the semantics of $p{\downarrow}$.

Lemma 3.3. If $(M,(R,F),D) \models p$, $(R,D) \rightrightarrows (R',D')$, then $(M,(R',F),D') \models (p{\downarrow})$ holds.

Delayed-control-transfer handling. We present the **CALL** rule and **RETL** 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. Here, we treat the instruction `call` and `jmp` differently, since the callee is expected to return and we need to make sure that caller's remaining code can resume executing safely after return. The inference rule designed for `jmp` is **J** rule. Fig 9(b) presents how our logic supports function call/return verification with delayed control-transfer. In Fig. 9(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 by $(\mathrm{fq}\ \iota \Rightarrow \mathtt{r}_{15} = \mathtt{f})$ when function `h` returns, in need of specifying a valid address to return to. The **RETL** rule in our logic still verify the instruction `retl` and its following instruction together because of the delayed control transfer. The postcondition of function `f` will be hold after the execution of instruction $\mathtt{i}_2$ following `retl`, and we do not have to know any specific information about $\mathtt{r}_{15}$ by the inspiration of the previous work SCAP. We just have to constrain that the instruction following `retl` does not update the value of $\mathtt{r}_{15}$ by $\mathbf{fretlSta}\,(p{\downarrow}{\downarrow}, q)$ (defined in Fig. 12), in order to keep the address saved in $\mathtt{r}_{15}$ are equal at the call and return points.

The **seq_frame** rule provides a local way for code block reasoning. The assertion p_r describes a part of state that the verification work doesn't care about.

$\boxed{\Psi \vdash C : \Psi'}$ **(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)**

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

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

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

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

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

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

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

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

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

$$\frac{\mathtt{sr} \mapsto \_ * p \Rightarrow (\mathtt{r}_s = v_1\ \wedge\ \mathtt{o} = v_2)}{\vdash \{\mathtt{sr} \mapsto \_ * p\}\ \mathtt{wr}\ \mathtt{r}_s\ \mathtt{o}\ \mathtt{sr}\ \{(\triangleright_3 \mathtt{sr} \mapsto (v_1 \oplus v_2)) * p\}}\ \textbf{(WR)}$$

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

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

Fig. 11. Seleted Inference Rules

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

Fig. 12. Auxiliary Definition for Inference Rules

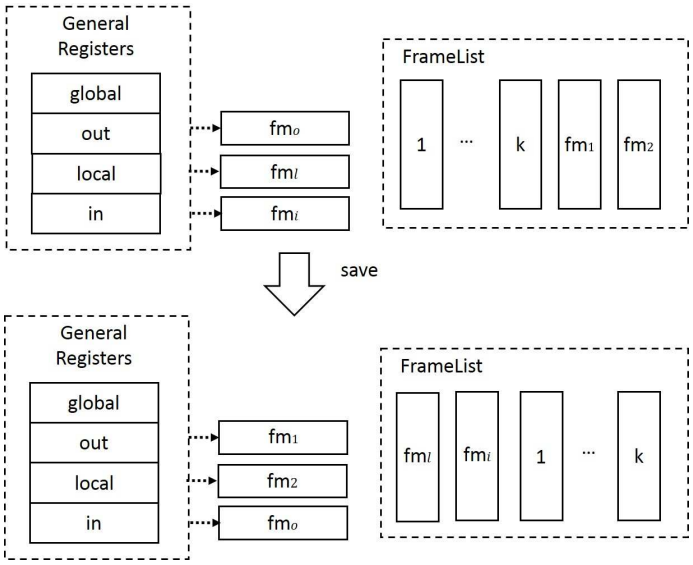


Fig. 13. Window Rotation for Instruction Save

The bottom layer of our logic is well-formed instruction for single instructions verification. The inference rules for instruction `save` and `restore` are presented in this layer.

Register Windows Handling. We introduce the inference rule for instruction `save` here. The instruction `save` will operate the frame list to store the contents of current window's local and in registers into frame list by window rotation and update the value of register $\mathtt{r}_d$ of the new window. Fig. 13 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 can be described as $(\mathrm{R}_o \mapsto \mathrm{fm}_o) * (\mathrm{R}_l \mapsto \mathrm{fm}_l) * (\mathrm{R}_i \mapsto \mathrm{fm}_i)$. The assertion $\mathtt{cwp} \mapsto (\!| w_{id}, F_{\_\cdot\_\cdot\_} |\!)$ tells us that the identity of the current window is w_{id} and the $F_{\_\cdot\_\cdot\_}$ is a prefix of the frame list in current state. Two reserved "$\_$" are used to make sure that there still exists a segment F in frame list after window rotation, because the window rotation will loads the two groups of register at the tail

of frame list to the local and out registers. So, the current state of the register windows can be presented as $\texttt{cwp} \mapsto (\!| w_{id}, F \cdot \_ \cdot \_ |\!) * (\mathrm{R}_o \mapsto \mathrm{fm}_o) * (\mathrm{R}_l \mapsto \mathrm{fm}_l) * (\mathrm{R}_i \mapsto \mathrm{fm}_i)$. The execution of instruction `save` will check whether the previous window is 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"). 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. We usually do not care about the contents of frames loaded when doing `save`. So, we do not have to specify them and the new state of register windows is $\texttt{cwp} \mapsto (\!| w'_{id}, \mathrm{fm}_l :: \mathrm{fm}_i :: F |\!) * (\mathrm{R}_o \mapsto \_) * (\mathrm{R}_l \mapsto \_) * (\mathrm{R}_i \mapsto \mathrm{fm}_o)$. Finally, the values extracting from $\mathtt{r}_s$ and `o` of original window are summed together and the result is stored in new window's $\mathtt{r}_d$ register. The new window's identity w'_{id} can be achieved by $\mathbf{next\_cwp}(w_{id})$.

$$
\begin{array}{l}
\quad \ldots \\
- \ \{(\%\mathtt{i}_0 \mapsto v \ * \ \mathtt{Y} \mapsto \_ \,,\, q)\} \\
\quad \mathtt{wr} \quad \%\mathtt{i}_0,\, 0,\, \mathtt{Y} \\
- \ \{(\%\mathtt{i}_0 \mapsto v \ * \ \triangleright_3 \mathtt{Y} \mapsto (v \oplus 0)) \,,\, q)\} \\
- \ \{(\%\mathtt{i}_0 \mapsto v \ * \ \triangleright_3 \mathtt{Y} \mapsto v \,,\, q)\} \\
\quad \mathtt{nop} \\
- \ \{(\%\mathtt{i}_0 \mapsto v \ * \ \triangleright_2 \mathtt{Y} \mapsto v \,,\, q)\} \\
\quad \mathtt{nop} \\
- \ \{(\%\mathtt{i}_0 \mapsto v \ * \ \triangleright_1 \mathtt{Y} \mapsto v \,,\, q)\} \\
\quad \mathtt{nop} \\
- \ \{(\%\mathtt{i}_0 \mapsto v \ * \ \triangleright_0 \mathtt{Y} \mapsto v \,,\, q)\} \\
\quad \ldots
\end{array}
$$

Fig. 14. Example for Delayed-write Handling

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 $\mathtt{r}_s$ and v_2 of operational expression `o` from precondition. Then, the state of special register `sr` is updated with $(\triangleright_3 \mathtt{sr} \mapsto v_1 \oplus v_2)$, which means we do not certainly know the state of the special register `sr` now but can make sure that $(\mathtt{sr} \mapsto v_1 \oplus v_2)$ holds after 3 cycles. Using assertion $\triangleright_n \mathtt{sr} \mapsto w$ to handle delayed-write can help us hide the information about delay list in the specification. We can find that if assertion $\triangleright_n \mathtt{sr} \mapsto w$ holds in the initial state, then $\triangleright_{n-1} \mathtt{sr} \mapsto w$ will hold after doing delay list transition if $n > 0$, or $\mathtt{sr} \mapsto w$ will hold if $n = 0$.

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

The method that we restrict the state of the special register, which may be recorded in delay list, described in a form of "$\triangleright_n \mathtt{sr} \mapsto w$" 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.

Program in Fig. 14 is a segment of function **ChangeY** in Fig. 1 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 $\triangleright_3 \mathtt{Y} \mapsto (v \oplus 0)$ after executing `wr`, where the symbol $\oplus$ is the xor operation. Although the value of special register `Y` may have not been updated at this time, its 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 register in precondition is not in delay list. So, getting a value of special register from an uncertain state like "$\triangleright_n \mathtt{sr} \mapsto v$" is prohibited in our logic.

3.3 Soundness

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

Definition 3.4 (Semantics of Well-formed Instruction).

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

Well-formed instruction's semantics says that the postcondition q will hold after the execution of `i` if the initial state S satisfying p. The soundness proof can be completed by discussing each case of instruction `i`.

Theorem 3.5 (Soundness of Well-formed Instruction).

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

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

Definition 3.6 (Semantics of Well-formed Instruction Sequence).

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

The $\mathsf{safety\_insSeq}\,(C, S, \mathtt{pc}, \mathtt{npc}, q, \Psi)$ [2] describes that the program can execute safely from state S, pc and npc to the first control transfer instruction (excluding call) it meets, and is formally defined in Def. 3.7. Here, "$\longmapsto^2$" means executing two steps.

Definition 3.7 (Instruction Sequence Execution Safety). The relation $\mathsf{safety\_insSeq}\,(C, S, \mathtt{pc}, \mathtt{npc}, q, \Psi)$ 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 $C(\mathtt{pc}) = \mathtt{i}$ then :
  - there exists $S', \mathtt{pc}, \mathtt{npc}'$, such that $C \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto (S', \mathtt{pc}', \mathtt{npc}')$,
  - for any $S', \mathtt{pc}', \mathtt{npc}'$, if $C \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto (S', \mathtt{pc}', \mathtt{npc}')$, then $\mathsf{safety\_insSeq}\ (C, S', \mathtt{pc}', \mathtt{npc}', q, \Psi)$
- if $C(\mathtt{pc}) = \mathtt{jmp\ a}$ then :
  - there exists $S', \mathtt{pc}', \mathtt{npc}'$, such that $C \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$
  - for any $S', \mathtt{pc}', \mathtt{npc}'$, if $C \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$, then there exists $\mathrm{fp}, \mathrm{fq}, \iota$ and p_r, such that the following holds :
    - $\mathtt{pc}' \in \mathrm{dom}(\Psi)$, $\Psi(\mathtt{pc}') = (\mathrm{fp}, \mathrm{fq})$, $\mathtt{npc}' = \mathtt{pc}' + 4$
    - $S' \models (\mathrm{fp}\ \iota) * p_r$, $(\mathrm{fq}\ \iota) * p_r \Rightarrow q$.
- if $C(\mathtt{pc}) = \mathtt{call\ f}$ then :
  - there exists $S', \mathtt{pc}', \mathtt{npc}'$, such that $C \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$.
  - for any $S', \mathtt{pc}'$ and $\mathtt{npc}'$, if $C \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$, then there exists $\mathrm{fp}, \mathrm{fq}, \iota$ and p_r, such that the following holds :
    - $\mathtt{pc}' = \mathtt{f}, \mathtt{npc}' = \mathtt{f} + 4$,
    - $\mathtt{f} \in \mathrm{dom}(\Psi)$, $\Psi(\mathtt{f}) = (\mathrm{fp}, \mathrm{fq})$,
    - $S' \models (\mathrm{fp}\ \iota) * p_r$,
    - for any S', if $S' \models (\mathrm{fq}\ \iota) * p_r$, then $\mathsf{safety\_insSeq}\ (C, S', \mathtt{pc}{+}8, \mathtt{pc}{+}12, q, \Psi)$,
    - for any S', if $S' \models (\mathrm{fq}\ \iota)$, then $S'.Q.R(\mathtt{r}_{15}) = \mathtt{pc}$.

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

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

Theorem 3.8 (Soundness of Well-formed Instruction Sequence).

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

The semantics of well-formed code heap 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 :

[2] We just list some selected cases of safety_insSeq because of space limitation. More details are presented in Coq implementation [1]

Definition 3.9 (Semantics of Well-formed Code Heap).

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

Theorem 3.10 (Soundness of Well-formed Code Heap).

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

Besides the above works, there is still an 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 $\mathsf{safe}^n(C, S, \mathtt{pc}, \mathtt{npc}, q, k)$ to describe the restrictions to obey in a process of a program execution below.

Definition 3.11 (Program Execution Safety). The $\mathsf{safe}^n(C, S, \mathtt{pc}, \mathtt{npc}, q, k)$ tells us that the program can execute n steps safely from $\mathtt{pc}$, $\mathtt{npc}$ 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) $\mathsf{safe}^0\,(C, S, \mathtt{pc}, \mathtt{npc}, q, k)$ always holds.
b) $\mathsf{safe}^{n+1}\,(C, S, \mathtt{pc}, \mathtt{npc}, q, k)$ holds if and only if :
   1. if $C(\mathtt{pc}) = \{\mathtt{i}, \mathtt{jmpl\ a}, \mathtt{be\ f}\}$ then :
      - there exists $S', \mathtt{pc}', \mathtt{npc}'$, such that $C \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto (S', \mathtt{pc}', \mathtt{npc}')$
      - for any $S', \mathtt{pc}', \mathtt{npc}'$, if $C \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto (S', \mathtt{pc}', \mathtt{npc}')$ then $\mathsf{safe}^n\,(C, S', \mathtt{pc}', \mathtt{npc}', q, k)$
   2. if $C(\mathtt{pc}) = \mathtt{call\ f}$ then :
      - there exists $S', \mathtt{pc}', \mathtt{npc}'$, such that $C \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$
      - for any $S', \mathtt{pc}', \mathtt{npc}'$, if $C \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$, then $\mathsf{safe}^n\,(C, S', \mathtt{pc}', \mathtt{npc}', q, k+1)$
   3. if $C(\mathtt{pc}) = \mathtt{retl}$ then :
      - there exists $S', \mathtt{pc}', \mathtt{npc}'$, such that $C \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$
      - for any $S', \mathtt{pc}', \mathtt{npc}'$, if $C \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$, then
        if $k = 0$ then
        $S' \models q$
        else
        $\mathsf{safe}^n\,(C, S', \mathtt{pc}', \mathtt{npc}', q, k-1)$

The parameter k of $\mathsf{safe}^n(C, S, \mathtt{pc}, \mathtt{npc}, q, k)$ is used to record the depth of function call, in order to judge which `retl` is the end point of the function we care about. The instruction `call` increases the depth of function call, but `retl`

decreases it and judges whether q holds if the depth of function call is zero ($k = 0$).

Now, we present an important corollary below. Supposing there is a function f, the corollary tells us that if a state S of start 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 3.12 (Well-formed Function). If $\Psi \models \{(p, q)\}\, \mathtt{pc} : C[\mathtt{pc}]$, $S \models p$, $\Psi \subseteq \Psi'$ and $\Psi \models C : \Psi'$, then $\forall n.\ \mathsf{safe}^n(C, S, \mathtt{pc}, \mathtt{pc} + 4, q, 0)$.

The proof of Corollary 3.12 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 3.13. If $\mathsf{safety\_insSeq}\,(C, S, \mathtt{pc}, \mathtt{npc}, q, \Psi)$, $\Psi \models C : \Psi'$, $q \Rightarrow \mathtt{r}_{15} = \mathtt{f}$, $(\forall S'.\ S' \models q \rightarrow \mathsf{safe}^n(C, S', \mathtt{f}+8, \mathtt{f}+12, q', k))$, then $\mathsf{safe}^n(C, S, \mathtt{pc}, \mathtt{npc}, q', k+1)$ holds.

Lemma 3.13 plays a similar role with well-formed stack presented in SCAP [6]. 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.

4 Verifying a Realistic Context Switch Module

We apply our program logic to verify a realistic context switch module implemented in SPARCv8. Fig. 15 shows the structure of the program we verified. And we describe the function of a part of them informally.

- *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.

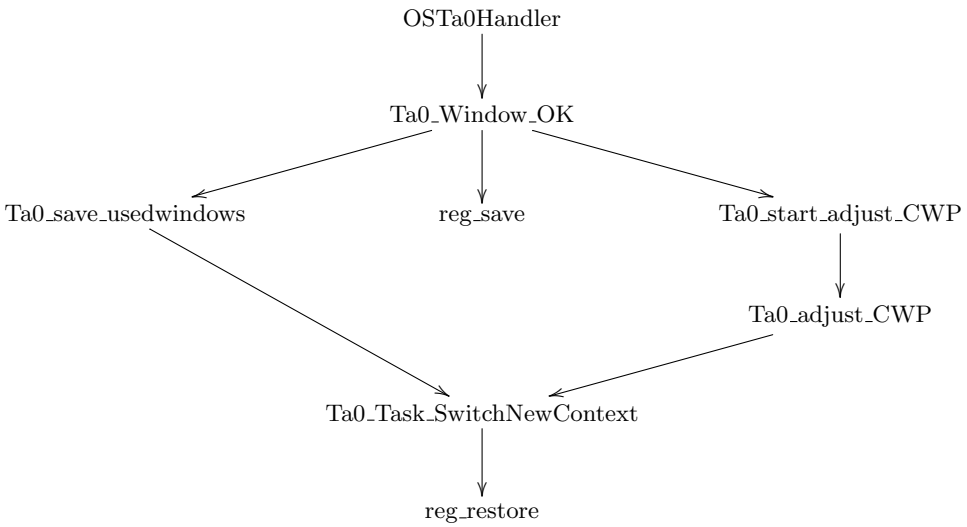


Fig. 15. The Structure of Context Switch Module

- *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.

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. 16 shows a relationship between register windows and stack. In Fig. 16, 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.

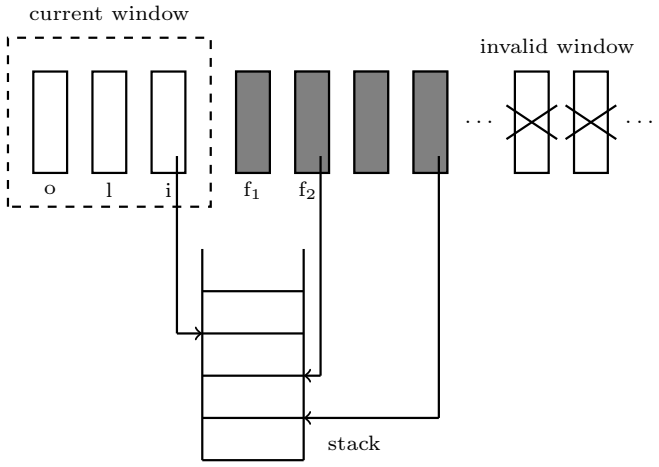


Fig. 16. Stack Windows Constraint

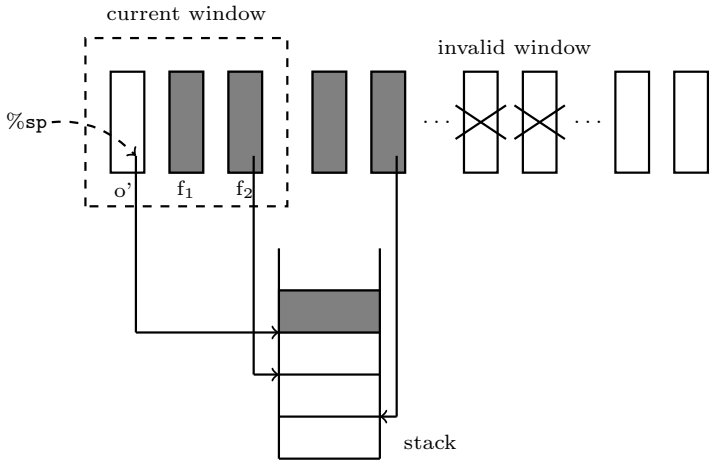


Fig. 17. Saving Context Stored in Frame List by Window Rotation

The code block Ta0_save_usedwindows (shown in Fig. 18) 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. 18 execution. Here, the constant OS_NWINS present the number of register windows. 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.

16. We can see that marking a window invalid is necessary, because the register windows is a cycle (as shown in Fig. 4) 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. 16, 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. 17. 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. 17). 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.

```
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    jmp       Ta0_save_usedwindows
25    nop
```

Fig. 18. Ta0_save_usedwindows Code

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. 18, in order to do verification. 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 $I(ow)$ below. We need 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

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

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

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

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

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

$$\begin{aligned}\mathbf{CWPInv}(ow, cw) \triangleq\ & \exists i.\ \mathrm{fm}_g[4] = i \wedge \mathrm{fm}_g[7] = 2^v \wedge i = \mathbf{rotate}(w_{oid}, w_{id}, v) \\ & \wedge ((i = 2^{w_{id}}) \vee (i = 2^{w_{id} + \mathrm{OS\_NWINS}} + 2^{w_{id}})) \\ \text{where } ow =\ & ((\mathrm{fm}_g, \mathrm{fm}_o, \mathrm{fm}_l, \mathrm{fm}_i), (w_{oid}, v, F)), \\ cw =\ & ((\mathrm{fm}'_g, \mathrm{fm}'_o, \mathrm{fm}'_l, \mathrm{fm}'_i), (w_{id}, v, F')).\end{aligned}$$

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

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

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

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

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

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

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

Fig. 19. Definition Used in Invariant

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. 19.

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

In the invariant, we use assertion 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. **SFInv**($ow, cw, cstk$) tells us the following things :

- **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}.
- **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. 16. The identities 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.
- **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, **rotate**(w_{oid}, w_{id}, v_i) tells us how to get i from the initial one. By the program shown in Fig.18, we know that the value of $\%\mathtt{g}_4$ will be updated as $(i >> (\mathrm{OS\_WINS} - 1) \,|\, (i << 1))$, where the $|$ is the or operation. And we can also find that i is equal to either $2^{w_{id}}$ or $2^{w_{id}+\mathrm{OS\_NWINS}}$.

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.

5 Related Work and Conclusion

There are many impressive pioneering works that focus on machine code verification. Projects on proof-carrying code (PCC) [12, 3] and typed assembly language (TAL) [9, 10] 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 [19] 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 [6] 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 [11] 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 [13] 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 [4, 5, 20]. So, these work can be the complementaries in our future work.

The existent works about SPARC are less. Wang *et al.* [17] 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 [16] which is a part of Foundational Proof-Carrying Code (FPCC) [3] 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 [14]. 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.

References

[1] Program logic for sparc implementation in coq (project code). `https://github.com/jpzha/VeriSparc`.

[2] Sparc. `https://gaisler.com/doc/sparcv8.pdf`.

[3] A. W. Appel. Foundational proof-carrying code. In *Proc. 16th Annual IEEE Symposium on Logic in Computer Science*, pages 85–97, Jan 1998.

[4] X. Feng and Z. Shao. Modular verification of concurrent assembly code with dynamic thread creation and termination. In *Proc. 2005 ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, pages 254–267, Sept 2005.

[5] X. Feng, Z. Shao, Y. Guo, and Y. Dong. Certifying low-level programs with hardware interrupts and preemptive threads. In *Proc. 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, pages 170–182, 2008.

[6] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular verification of assembly code with stack-based control abstractions. In *Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*, June 2006.

[7] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S.-C. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *Proc. 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15)*, pages 595–608, Jan 2015.

[8] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*, pages 207–220, Oct 2009.

[9] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. Talx86:a realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, May 1996.

[10] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system f to typed assembly language. In *Proc. 25th ACM Symp. on Principles of Prog. Lang.*, pages 85–97, Jan 1998.

[11] M. O. Myreen and M. J. Gordon. Hoare logic for realistically modelled machine code. In *Proc. 13th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2007.

[12] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proc.2nd USENIX Symp. on Operating System Design and Impl*, pages 229–243, 1996.

[13] Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *Proc. 33rd ACM Symposium on Principles of Programming Languages (POPL'06)*, pages 320–333, 2006.

[14] Z. Ni, D. Yu, and Z. Shao. Using xcap to certify realistic systems code: Machine context management. In *Proc. 20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'07)*, Sept 2007.

[15] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proc. 17th Annual IEEE Symposium on Logic in Computer Science*, 2002.

[16] G. Tan and A. W. Appel. A compositional logic for control flow. In *Proc. 7th Verification, Model Checking and Abstract Interpretation (VMCAI'06)*, Jan 2006.

[17] J. Wang, M. Fu, L. Qiao, and X. Feng. Formalizing sparcv8 instruction set architecture in coq. In *Symposium on Dependable Software Engineering Theories, Tools and Applications*, Oct 2007.

[18] F. Xu, M. Fu, X. Feng, X. Zhang, H. Zhang, and Z. Li. A practical verification framework for preemptive os kernels. In *Proc. 28th International Conference on Computer Aided (CAV'16)*, pages 59–79, July 2016.

[19] D. Yu, A. H. Nadeem, and Z. Shao. Building certified libraries for pcc : Dynamic storage allocation. *Science of Computer Programming*, 50(1-3):101–127, Mar 2004.

[20] D. Yu and Z. Shao. Verification of safety properties for concurrent assembly code. In *Proc. 2004 International Conference on Functional Programming*, Sept 2004.