Modular Verification of SPARCv8 Code*

Junpeng Zha[1], Xinyu Feng[2,✉], and Lei Qiao[3]

[1] Nanjing University
[2] State Key Laboratory for Novel Software Technology, Nanjing University
xyfeng@nju.edu.cn
[3] Beijing Institute of Control Engineering

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 applied it to verify the main body of a context switch routine in a realistic embedded OS kernel, and extended it to support refinement verification. 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. And some other codes that are not required to be written in assembly are also implemented in assembly (*e.g.*memcpy in linux v2.6.17.10 [2]), in order to achieve high performance. 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.* [29, 15, 12]), 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 [4]. It has been widely used in various processors for workstations and embedded systems. The SPARCv8 ISA has some interesting features, which make it a non-trivial task to design a Hoare-style program logic for assembly code.

– *Delayed control transfers.* SPARCv8 has two program counters `pc` and `npc`. The `npc` register points to the next instruction to run. Control-transfer instructions in SPARCv8 change `npc` instead of `pc` to the target program point, while `pc` takes the original value of `npc`. This makes the control transfer to happen one cycle later than the execution of the control transfer instructions.

* This work is supported in part by grants from National Natural Science Foundation of China (NSFC) under Grant Nos. 61632005, 61502442 and 61502031.

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

Fig. 1. An Example for SPARC Code

- *Delayed writes.* The `wr` instruction that writes a special class of registers does not take effect immediately. Instead the write operation is buffered and then executed X cycles later, where X is a predefined system parameter which usually ranges from 0 to 3.
- *Register windows.* SPARCv8 uses register windows and the window rotation mechanism to avoid saving contexts in the stack directly and achieves high performance in context management.

We use a simple example in Fig. 1 to show these three features. The function `CALLER` calls `ChangeY`, which updates the special register `Y` and returns its original value.

`ChangeY` requires an input parameter as the new value for the special register `Y`. `CALLER` calls `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 transfers".

SPARCv8 uses the `save` instruction (at line 3 in the example) to save the current context and `restore` (at line 10) to restore it. Its 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. Correspondingly, we give aliases "$\%\mathtt{g}_0 \sim \%\mathtt{g}_7$", "$\%\mathtt{o}_0 \sim \%\mathtt{o}_7$", "$\%\mathtt{l}_0 \sim \%\mathtt{l}_7$" and "$\%\mathtt{i}_0 \sim \%\mathtt{i}_7$" for these groups respectively. The out, local and in registers form the *current register window*. The local registers are for private use in the current context. The 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` in this example tells us that, in addition to operate the register window, the execution of this instruction will also assign the value of ($\%\mathtt{sp} - 64$), where the value of the register $\%\mathtt{sp}$ is taken from the original window, to the $\%\mathtt{sp}$ register in the new register window. The $\%\mathtt{sp}$ register, which is an alias of $\mathtt{r}_{14}$, acts as the stack pointer.

So, " `save` $\%\mathtt{sp}, -64, \%\mathtt{sp}$ " in this example means that we allocate a new stack frame, whose size is 64 byte, to the callee (`ChangY`). The arguments taken by the `restore` are irrelevant here and can be ignored here.

At line 6, the `wr` instruction tries to update the special register `Y` with the value of $\%\mathtt{i}_0 \oplus 0$ (bitwise exclusive OR). However, the write is delayed for X cycles, where X is some predefined system parameter that ranges from 0 to 3. For portability, programmers usually do not rely on the exact value of X and assume it takes the maximum value 3. Therefore three `nop` instructions are inserted. Reading of `Y` earlier than line 9 may give us the old value. This feature is called "delayed writes".

These 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 allow 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 main body of the task context switch routine in the kernel. Then, considering some works, *e.g.*[29, 10], define a set of abstract assembly primitives to substitute their concrete implementations for some convenience in program verification. We extend our program logic to support refinement verification to bridge the gap. The extended program logic ensures the *contextual refinement* relation [18, 29] between the implementation of SPARCv8 function and its corresponding abstract assembly primitive. The *contextual refinement* can be represented as the following form, which means that calling a SPARCv8 function C_{as} will not produce more observable behaviors than calling its corresponding abstract assembly primitive Υ under *any context* C:

$$\forall\, C.\, C[C_{\text{as}}] \subseteq C[\Upsilon]$$

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 block, which models the delayed control transfer. To reason about delayed writes, we introduce a modal assertion $\triangleright_t \mathtt{sr} \mapsto w$, saying that the special register `sr` will hold the value w in up to t cycles. We also give logic rules for `save` and `restore` instructions that do register window rotation.
- Following SCAP [11], 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) 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 direct-style semantic interpretation for the logic judgments, based on which we establish the soundness. This is different from previous work, which either does syntactic-based soundness proof (*e.g.* SCAP [11]) or treats return code pointers as first-class code pointers and gives CPS-style (continuation-passing 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 main body of the context switch routine in a realistic embedded OS kernel for aerospace crafts, which consists of around 250 lines of SPARCv8 code.
- In order to support refinement verification, we define a Pseudo-SPARCv8 program as our high-level program, which is multithread and can call abstract assembly primitives. It also abstracts or omits some sophisticated mechanism, like register window and delayed writes, in SPARCv8, and makes the program state of Pseudo-SPARCv8 program more simpler than the physical SPARCv8 program defined in Sec. 2. So, it provides us more convenience to write the abstract assembly primitive in high-level program.
- We extend the program logic defined before to support refinement verification. The extended logic can ensure the *contextual refinement* between of SPARCv8 functions and their corresponding abstract assembly primitives. Thanks for our direct-style semantics interpretation for the logic judgments proposed before, which allows us to establish the correctness of each assembly function separatly, the extension does not meet much challenges.

The program logic, its soundness proof, the verification of the context switch module, and the extended program logic for refinememt verification have been mechanized in Coq[3].

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, and show how our logic supprts the three main features of SPARCv8. We show the verification of the main body of the context switch routine in Sec. 4. The extended program logic to support refinement verification is presented in Sec. 5, and the high-level Pseudo-SPARCv8 program is introduced in Sec. 5.1. Finally we discuss more on related work and conclude in Sec. 6.

2 The SPARCv8 Assembly Language

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

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

Fig. 2. Machine States and Language for SPARCv8 Code

2.1 Language syntax and states

The machine model and syntax of SPARCv8 assembly language are defined in Fig. 2. Here, we follow the block-based memory [17] used in CompCert, so that each procedure can have a block of its own stack frame. The memory address l is defined as a pair of its block id and the offset. The type of block is the integer in mathemantics represent as $\mathbb{Z}$, and the type of offset is a 32-bit integer, which we called *word* in our work. So, the value here is either a word w or address l. 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 addresses or locations where the commands are saved in code heap. The operand expression `o`, which is either a general register `r` or an immediate value w, and address expression `a`, which is either a operand expression or a sum of the value of register `r` and an operation expression, are auxiliary definitions used as paramters of commands. Note, the immediate value is a word, and not allowed to be an address in our work. 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 mapping 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.

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*.

$$
\begin{array}{rrcl@{\qquad}rrcl}
\text{(RegFile)} & R & \in & \text{RegName} \rightharpoonup \text{Val} & \text{(RegName)} & \texttt{rn} & ::= & \texttt{r}_0 \mid \ldots \mid \texttt{r}_{31} \mid \texttt{psr} \mid \texttt{sr} \\
\text{(PsrReg)} & \texttt{psr} & ::= & \texttt{n} \mid \texttt{z} \mid \texttt{v} \mid \texttt{c} \mid \texttt{cwp} & \text{(SpeReg)} & \texttt{sr} & ::= & \texttt{wim} \mid \texttt{Y} \mid \texttt{asr}_0 \mid \ldots \mid \texttt{asr}_{31} \\
\text{(FrameList)} & F & ::= & \text{nil} \mid \text{fm} :: F & \text{(Frame)} & \text{fm} & := & [v_0, \ldots, v_7] \\
\text{(DelayBuff)} & D & ::= & \text{nil} \mid (t, \texttt{sr}, w) :: D & \text{(DelayCycle)} & t & \in & \{0, 1, \ldots, X\}
\end{array}
$$

Fig. 3. Register File, Frame List and DelayBuffer

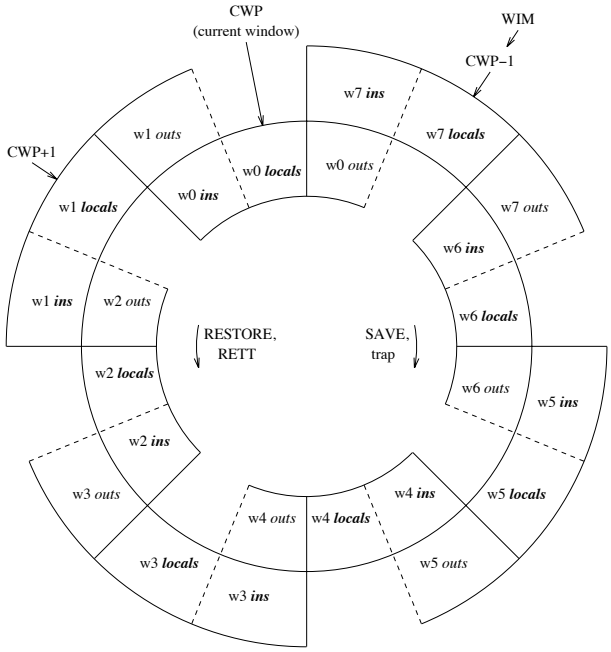

Fig. 4. Register Windows (figure taken from [4])

At the entry and exit of functions and traps, one may need 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 ($\texttt{cwp} = 0$ 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).

To save the context, the `save` instruction rotates the window by decrements the `cwp` pointer (modulo N). So w_7 becomes the current window. The out registers of w_0 becomes the in registers of w_7. The in and local registers of w_0 become

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

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

$$R\{[\mathtt{r}_i, \ldots, \mathtt{r}_{i+7}] \rightsquigarrow \mathrm{fm}\} ::= R\{\mathtt{r}_i \rightsquigarrow v_0\} \ldots \{\mathtt{r}_{i+7} \rightsquigarrow v_7\} \quad \text{where} \quad \mathrm{fm} = [v_0, \ldots, v_7]$$

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

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

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

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

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

inaccessible. This is like pushing them onto the circular stack. The `restore` instruction does the inverse, which is like a stack pop.

The `wim` register is used as a bit vector to record the end of the stack. Each bit in `wim` corresponds to a register window. The bit corresponding 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, in order to avoid the overflow of register window because of the limitation of the number of windows. 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 $\mathtt{sr}$ and the value w to be written. Note that we restrict that the value of a special register can only be a word, because the special registers are used to record the state of processor, and there is impossible to store memory addresses in them.

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 $\mathtt{i}$, because the actual control-transfer occurs after the execution of $\mathtt{i}$. The end of each instruction sequence can only be $\mathtt{jmp}$ or $\mathtt{retl}$ followed by a simple instruction $\mathtt{i}$. Note that we do not view the $\mathtt{call}$ 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 $\mathtt{f}$ 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

The operational semantics is taken from Wang *et al.* [28], but we omit features like interrupts and traps. We show the selected rules in Fig. 6. The program transition relation $C \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto (S', \mathtt{pc}', \mathtt{npc}')$ is defined in Fig. 6 (a). Before the execution of the instruction pointed by $\mathtt{pc}$, the delayed writes in D with 0 delay cycles are executed first. The execution of the delayed writes are defined in the form of $(R, D) \rightrightarrows (R', D')$, as shown below:

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

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

Note that the write of $\mathtt{sr}$ has no effect if $\mathtt{sr}$ is not in the domain of R. Since R is defined as a partial map, we can prove the following lemma.

Lemma 2.1. $(R, D) \rightrightarrows (R', D')$ and $R = R_1 \uplus R_2$, if and only if there exists R'_1 and R'_2, such that $(R_1, D) \rightrightarrows (R'_1, D')$, $(R_2, D) \rightrightarrows (R'_2, D')$, and $R' = R'_1 \uplus R'_2$.

Here the disjoint union $R_1 \uplus R_2$ represents the union of R_1 and R_2 if they have disjoint domains, and undefined otherwise. This lemma is important to give sound semantics to delay buffer related assertions, as discussed in Sec. 3.

The transition steps for individual instructions are classified into three categories: the control transfer steps ($\_ \vdash \_ \ \circ\!\!\longrightarrow \_$), the steps for $\mathtt{save}$, $\mathtt{restore}$ and

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

(a) Program Transistion

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

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

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

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

(b) Control Transfer Instruction Transition

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

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

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

(c) Save, Restore and Wr instruction Transition

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

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

(d) Simple Instruction Transition

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

(e) Expression Semantics

Fig. 6. Selected operational semantics rules

`wr` instructions ($\_ \bullet\!\!\xrightarrow{=} \_$), and the steps for other simple instructions ($\_ \xrightarrow{=} \_$). The corresponding step transition relations are defined inductively in Fig. 6 (b), (c) and (d) respectively.

Note that, after the control-transfer instructions, `pc` is set to `npc` and `npc` contains the target code pointer. This explains the one cycle delay for the control transfer. The `call` instruction saves `pc` into the register $\mathtt{r}_{15}$, while `retl` uses $\mathtt{r}_{15}+8$ as the return address (which is the address for the second instruction following the `call`). Evaluation of expressions `a` and `o` is defined as $[\![\mathtt{a}]\!]_R$ and $[\![\mathtt{o}]\!]_R$ in Fig. 6 (e). Here, we define the sum of two values v_1 and v_2 below. The result of v_1+v_2 is legel, if both of the v_1 and v_2 are words (Int32), or v_1 is an address and v_2 is an offset. This is explain why the immediate value in our work is a word, because immediate value usually acts as an offset, which is a word, in the calculation of address.

$$v_1 + v_2 \ ::= \begin{cases} w_1 + w_2 & \textbf{if } v_1 = w_1, \text{ and } v_2 = w_2 \\ (b, w_1 + w_2) & \textbf{if } v_1 = (b, w_1), \text{ and } v_2 = w_2 \\ \bot & \textbf{otherwise} \end{cases}$$

The `wr` wants to save the bitwise exclusive OR of the operands into the special register `sr`, but it puts the write into the delay buffer D instead of updating R immediately. The operation $\textbf{set\_delay}(\mathtt{sr}, w, D)$ is defined below:

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

where X ($0 \leq X \leq 3$) is a predefined system parameter for the delay cycle. Note that as we have explained before, special register is used to record the state of processors, so we do not permit saving memory address in it.

The `save` and `restore` instruction rotate the register windows and update the register file. Their operations over F and R are defined in Fig. 5.

3 Program Logic

In this section, we introduce the assertion language and program logic designed for SPARCv8 program.

3.1 Assertions

$$\begin{aligned} \textit{(Asrt)}\ p, q ::= {} & \mathsf{emp} \mid l \mapsto v \mid \mathtt{rn} \mapsto v \mid \ \rhd_t \mathtt{sr} \mapsto w \mid p\downarrow \ \mid \mathtt{cwp} \mapsto (\!| w_{id}, F |\!) \\ & \mid \ p \wedge q \mid p \vee \ q \mid p * q \mid \mathtt{a} =_a v \mid \mathtt{o} = v \mid \forall x.\, p \mid \exists x.\, q \mid \ \ldots \end{aligned}$$

Fig. 7. Syntax of Assertions

We define syntax of assertions in Fig. 7, and their semantics in Fig. 8. We extend separation logic assertions with specifications of delay buffers and register windows. Registers are like variables in separation logic, but are treated as

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

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

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

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

Fig. 8. Semantics of Assertions

resources. The assertion emp says that the memory and the register file are both empty. $l \mapsto v$ specifies a singleton memory cell with value v stored in the address l. $\mathtt{rn} \mapsto v$ says that $\mathtt{rn}$ is the only register in the register file and it contains the value v. Also $\mathtt{rn}$ is *not* in the delay buffer. Separating conjunction $p * q$ has the standard semantics as in separation logic [26].

The assertion $\triangleright_t \mathtt{sr} \mapsto w$ describes a delayed write in the delay buffer D. It describes the uncertainty of $\mathtt{sr}$'s value in R, which is unknown for now but will become w in up to $t{+}1$ cycles. We use $\_ \rightrightarrows^k \_$ to represent k-step execution of the delayed writes in D. It also requires that there be at most one delayed write for a specific special register $\mathtt{sr}$ in D (*i.e.* $\mathbf{noDup}(\mathtt{sr}, D)$). This prevents more than one delayed writes to the same register within 4 instruction cycles, which practically have no restrictions on programming. By the semantics we have

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

The assertion $p\downarrow$ allows us to reduce the uncertainty by executing one step of the delayed writes. It specifies states reachable after executing one step of delayed writes from those states satisfying p. Therefore we know:

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

Also it's easy to see that if p syntactically does not contain sub-terms in the form of $\triangleright_t \mathtt{sr} \mapsto w$, then $(p\downarrow) \iff p$.

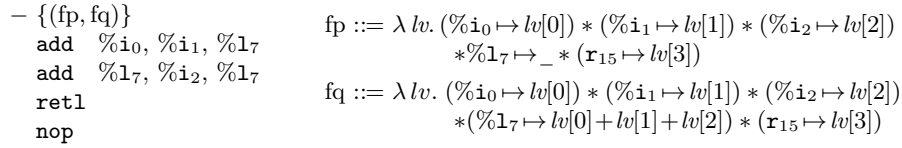

Fig. 9. Example for Function Specification

The following lemma shows $(\_)\downarrow$ is distributive over separating conjunction.

Lemma 3.1. $(p * q)\downarrow \iff (p\downarrow) * (q\downarrow)$.

The lemma can be proved following Lemma 2.1.

We use $\texttt{cwp} \mapsto (\!| w_{id}, F |\!)$ to describe the pointer `cwp` of the current register window and the frame list as a circular stack. Note that F is just a prefix of the frame list, since usually we do not need to know contents of the full list. Here we use $F{\cdot}F'$ to represent the concatenation of lists F and F'. Therefore we have $\texttt{cwp} \mapsto (\!| w_{id}, F{\cdot}F' |\!) \implies \texttt{cwp} \mapsto (\!| w_{id}, F |\!)$.

The assertions $\texttt{a} =_a v$ and $\texttt{o} = v$ describe the value of `a` and `o` respectively. They are intuitionistic assertions. Since `a` is used as an address, we also require it to be properly aligned on a 4-byte boundary. We define **word_align** to represent this restriction below. The result of the address expression `a` may be a word, if it's a pointer in code heap, or an memory address, if it's a location of memory.

$$\textbf{word\_align}(v) ::= \exists\, w.\, (v = w \,\vee\, (\exists\, b.\, v = (b, w))) \,\wedge\, w\%4 = 0$$

3.2 Inference Rules

The code specification θ and code heap specification Ψ are defined below:

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

The code heap specification Ψ maps the code labels for basic blocks to their specifications θ, which is a pair of pre- and post-conditions. Instead of using normal assertions, the pre- and post-conditions are assertions parameterized over a list of values *lgvl*. They play the role of auxiliary variables — Feeding the pre- and the post-conditions with the same *lgvl* allows us to establish relationship of states specified in the pre- and post-conditions.

Although we assign a θ to each basic block, the post-condition does not specify the states reached at the end of the block. Instead, it specifies the condition that needs to be specified in the future when the *current function* returns. This follows the idea developed in SCAP [11], but we use the standard unary state assertion instead of the binary state assertions used in SCAP, so that existing proof techniques (such as Coq tactics) for standard Hoare-triples can be applied to simplify the verification process.

We give a simple example in Fig. 9 to show a specification for a function, which simply sums the values of the registers $\%\mathtt{i}_0$, $\%\mathtt{i}_1$ and $\%\mathtt{i}_2$ and writes the result into the register $\%\mathtt{l}_7$. The specification $(\mathrm{fp}, \mathrm{fq})$ says that, when provided with the same lv as argument, the function preserves the value of $\%\mathtt{i}_0$, $\%\mathtt{i}_1$ and $\%\mathtt{i}_2$, $\%\mathtt{l}_7$ at the end contains the sum of $\%\mathtt{i}_0$, $\%\mathtt{i}_1$ and $\%\mathtt{i}_2$, and the function also preserves the value of $\mathtt{r}_{15}$, which it uses as the return address. To verify the function, we need to prove that it satisfies $(\mathrm{fp}\, lv, \mathrm{fq}\, lv)$ for all lv.

Figure 10 shows selected inference rules in our logic. The top rule **CDHP** verifies the code heap C. It requires that every basic block specified in Ψ can be verified with respect to the specification, with any argument ι used to instantiate the pre- and post-conditions.

The **SEQ** rule is applied when meeting an instruction sequence starting with a simple instruction `i`. The instruction `i` is verified by the corresponding well-formed instruction rules, with the precondition $p\downarrow$ and some post-condition p'. We use $p\downarrow$ because there is an implicit step executing delayed writes before executing every instruction. The post-condition p' for `i` is then used as the precondition to verify the remaining part of the instruction sequence.

Delayed control transfers. We distinguish the `jmp` and `call` instructions — The former makes an *intra-function* control transfer, while the latter makes function calls. The **JMP** rule requires that the target address is a valid one specified in Ψ. Starting from the precondition p, after executing the instruction `i` following **JMP** and the corresponding delayed writes, the post-condition p' of `i` should satisfy the precondition of the target instruction sequence, with some instantiation ι of the logical variables and a frame assertion p_r. Since the target instruction sequence of `jmp` is in the same function as the `jmp` instruction itself, the post-condition fq specified at the target address (with the same instantiation ι of the logical variables and the frame assertion p_r) should meet the post-condition q of the current function. As we explained before, the post-condition q does not specify the states reached at the end of the instruction sequence (which are specified by p' instead).

The **CALL** rule is similar to the **JMP** rule in that it also requires the post-condition p_2 of the instruction `i` following the `call` satisfy the precondition of the target instruction sequence, with some instantiation ι of the logical variables and a frame assertion p_r. Here we need to record that the code label `f` is saved in $\mathtt{r}_{15}$ by the `call` instruction. When the callee returns, its post-condition fq (with the same instantiation of auxiliary variables ι) needs to ensure $\mathtt{r}_{15}$ still contains `f`, so that the callee returns to the correct address. Also the fq with the frame p_r needs to satisfy the precondition p' for the remaining instruction sequences of the caller.

The **RETL** rule simply requires that the post-condition q holds at the end of the instruction `i` following `retl`. Also `i` cannot touch the register $\mathtt{r}_{15}$, therefore $\mathtt{r}_{15}$ specified in p must be the same as in q. Since at the calling point we already required that the post-condition of the callee guarantees $\mathtt{r}_{15}$ contains the correct return address, we know $\mathtt{r}_{15}$ contains the correct value before `retl`.

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

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

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

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

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

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

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

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

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

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

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

where $[\mathtt{r}_i, \ldots, \mathtt{r}_{i+7}] \mapsto [w_0, \ldots, w_7] \ ::= \mathtt{r}_i \mapsto w_0 * \cdots * \mathtt{r}_{i+7} \mapsto w_7$
and out, local and in are defined in Fig. 5.

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

Fig. 10. Seleted Inference Rules

Delayed writes and register windows. The bottom layer of our logic is for well-formed instructions. The **WR** rule requires the ownership of the target register `sr` in the precondition ($\mathtt{sr} \mapsto \_$). Also it implies there is no delayed writes to `sr` in the delay buffer (see the semantics defined in Fig. 8). At the end of the delayed write, we use $\triangleright_3 \mathtt{sr} \mapsto w_1 \oplus w_2$ to indicate the new value will be ready in up to 3 cycles. Since the maximum delay cycle X cannot be bigger than 3 and the value of X may vary in different systems, programmers usually take a conservative approach to assume $X = 3$ for portability of code. Our rule reflects this conservative view. The **RD** rule says the special register can be read only if it is not in the delay buffer. The **SAVE** and **RESTORE** rules reflect the save and recovery of the execution contexts, which is consistent with the operational semantics of the `save` and `restore` instructions given in Figs. 5 and 6.

3.3 Semantics and Soundness

We first define the safety of instruction sequences, $\mathsf{safe\_insSeq}(C, S, \mathtt{pc}, \mathtt{npc}, q, \Psi)$. It says C can execute safely from S, `pc` and `npc` until reaching the end of the current instruction sequence ($C[\mathtt{pc}]$), and q holds if $C[\mathtt{pc}]$ ends with the return instruction `retl`. It is formally defined in Def. 3.2. Here we use "$\_ \longmapsto^n \_$" to represent n-step execution. The defintion can ensure the *progress* and *preservations* of the execution of the instruction sequence. *Progress* property means the program can execute a step, when meeting simple instructions, or two steps, when meeting the delayed control transfer instructions, *e.g.*`call` and `jmp`. *Preservations* property means that if the program can execute one or two steps, the remaining part of instruction sequence can still execute safely.

Definition 3.2 (Safety of Instruction Sequences).
$\mathsf{safe\_insSeq}(C, S, \mathtt{pc}, \mathtt{npc}, q, \Psi)$ holds if and only if the following are true (we omit the case for `be` here, which is similar to `jmp`):

- if $C(\mathtt{pc}) = \mathtt{i}$ then:
  - there exist $S', \mathtt{pc}, \mathtt{npc}'$, 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 $\mathsf{safe\_insSeq}\ (C, S', \mathtt{pc}', \mathtt{npc}', q, \Psi)$.
- if $C(\mathtt{pc}) = \mathtt{jmp\ a}$ then:
  - there exist $S', \mathtt{pc}', \mathtt{npc}'$, 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 exist $\mathrm{fp}, \mathrm{fq}, \iota$ and p_r, such that the following hold:
    (1) $\mathtt{npc}' = \mathtt{pc}' + 4$, $\Psi(\mathtt{pc}') = (\mathrm{fp}, \mathrm{fq})$,
    (2) $S' \models (\mathrm{fp}\ \iota) * p_r,\ (\mathrm{fq}\ \iota) * p_r \Rightarrow q$.
- if $C(\mathtt{pc}) = \mathtt{be\ f}$ then ...
- if $C(\mathtt{pc}) = \mathtt{call\ f}$ then:
  - there exist $S', \mathtt{pc}', \mathtt{npc}'$, 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 exist $\mathrm{fp}, \mathrm{fq}, \iota$ and p_r, such that the following hold:
    (1) $\mathtt{npc}' = \mathtt{pc}' + 4$, $\Psi(\mathtt{pc}') = (\mathrm{fp}, \mathrm{fq})$,
    (2) $S' \models (\mathrm{fp}\ \iota) * p_r$,
    (3) for any S', if $S' \models (\mathrm{fq}\ \iota) * p_r$, then $\mathsf{safe\_insSeq}\ (C, S', \mathtt{pc} + 8, \mathtt{pc} + 12, q, \Psi)$,

(4) for any S', if $S' \models (\text{fq}\ \iota)$, then $S'.Q.R(\mathtt{r}_{15}) = \mathtt{pc}$.

– if $C(\mathtt{pc}) = \mathtt{retl}$ then :
  - there exist $S', \mathtt{pc}', \mathtt{npc}'$, 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 $S' \models q$, $\mathtt{pc}' = S'.Q.R(\mathtt{r}_{15})+8$, and $\mathtt{npc}' = S'.Q.R(\mathtt{r}_{15})+12$.

Then we can define the semantics for well-formed instruction sequences and well-formed code heap. The semantics of well-formed instruction sequences tells us that for any C, if the instruction sequence starting from label $\mathtt{f}$ is $\mathbb{I}$, then the instruction sequence $\mathbb{I}$ can execute safely if the initial state S satisfies the precondition p. And the semantics of well-formed code heap says that all the instruction sequences in code heap C given specifications can execute safely.

Definition 3.3 (Judgment Semantics).

– $\Psi \models \{(p, q)\}\ \mathtt{f} : \mathbb{I}$ if and only if, for all C and S such that $C[\mathtt{f}] = \mathbb{I}$ and $S \models p$, we have $\mathsf{safe\_insSeq}(C, S, \mathtt{f}, \mathtt{f}+4, q, \Psi)$.

– $\models C : \Psi$ if and only if, for all $\mathtt{f}$, fp and fq such that $\Psi(\mathtt{f}) = (\text{fp}, \text{fq})$, we have $\Psi \models \{(\text{fp}\ \iota, \text{fq}\ \iota)\}\ \mathtt{f} : C[\mathtt{f}]$ for all ι.

Next we define the safety $\mathsf{safe}^n(C, S, \mathtt{pc}, \mathtt{npc}, q, k)$ of whole program execution in Def. 3.4. It says that, starting with $\mathtt{pc}$, $\mathtt{npc}$ and the state S, and with the depth k of function calls, the code C either *halts* in less than n steps, with the final state satisfies q, or it executes at least n steps safely. Here we say C halts if it reaches the return point of the topmost function (when the depth k of the function call is 0). In the definition below, the depth k increases by the `call` instruction and decreases by `retl` (unless $k = 0$).

Definition 3.4 (Program Safety). $\mathsf{safe}^0(C, S, \mathtt{pc}, \mathtt{npc}, q, k)$ always holds. $\mathsf{safe}^{n+1}(C, S, \mathtt{pc}, \mathtt{npc}, q, k)$ holds if and only if the following are true:

1. if $C(\mathtt{pc}) \in \{\mathtt{i}, \mathtt{jmp\ a}, \mathtt{be\ f}\}$, then:
   – there exist $S', \mathtt{pc}', \mathtt{npc}'$, 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 exist $S', \mathtt{pc}', \mathtt{npc}'$ 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 exist $S', \mathtt{pc}', \mathtt{npc}'$ 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
   $\mathsf{safe}^n(C, S', \mathtt{pc}', \mathtt{npc}', q, k-1)$.

Then the following theorem and corollary show the soundness of our logic. The correctness of corollary 3.6 guarantees that we can get the correctness of the whole program, if each basic code block or internal function composing it is verified by our program logic.

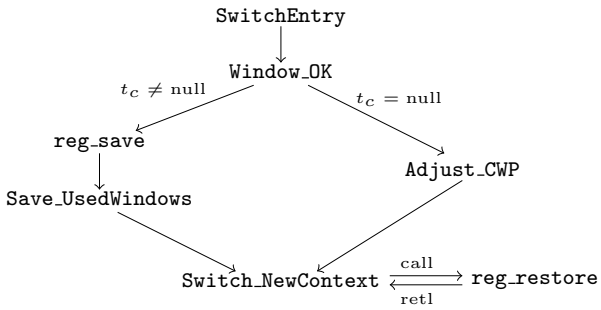

Fig. 11. The Structure of Context Switch Module

Theorem 3.5 (Soundness). $\vdash C{:}\Psi \Longrightarrow \models C{:}\Psi$

Corollary 3.6 (Function Safety). If $\Psi \models \{(p,q)\}\,\mathtt{pc} : C[\mathtt{pc}]$, $S \models p$, and $\models C{:}\Psi$, then $\forall n.\ \mathsf{safe}^n(C, S, \mathtt{pc}, \mathtt{pc}{+}4, q, 0)$.

4 Verifying a Realistic Context Switch Module

We apply our program logic to verify the main body of a context switch routine implemented in SPARCv8, which is used to save the current task's context and restore the new task's context. Figure 11 shows the structure of the code.

- `SwitchEntry` is the entry of the module. It checks `SwitchFlag` to see if a context switch is needed. If yes, it enters the `Window_OK` block.
- `Window_OK` checks if the current task is null (which may happen if the switch follows the delete of the current task). If yes, it jumps to `Adjust_CWP`, which resets the pointer `cwp` of the current register window so that it points to the last valid window. It essentially pops all the frames to empty the circular stack of register windows. If the current task is *not* null, it calls `reg_save` to save the general registers into the TCB, and then enter the code block `Save_UsedWindows` to save other register windows (F in our state model).
- `Save_UsedWindows` saves the register windows (except the current one) into the current task's stack in memory. It checks whether the previous window is valid. If it's valid, use the instruction `restore` to set the previous window as the current one, and save its contents into stack (in memory), then check the previous one continuously.
- `Switch_NewContext` restores the general registers and other register windows from the new task's TCB and its stack in memory, respectively. Then it sets the new task as the current one.

The main complexity of the verification lies in the code manages the register windows. To save all the register windows, `Save_UsedWindows` repetitively restores the next window into general registers (as the current window) and then saves them into memory, until all the windows are saved.

Specification. Below we give the pre- and post-conditions (a_{pre} and a_{post}) of the verified module. Each of them takes 5 arguments, the id of the current task t_c, the id of the new task t_n, the value *flag* of the SwitchFlag, the values *env* of general registers and all other register windows, and the new task's context *nst* that needs to be restored.

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

In the specification, we use $\mathsf{Env}(\mathit{env})$ to specify the values of general registers and the register windows. The variable TaskNew records the identifier of the new task. If SwitchFlag is false, we do not need any knowledge about the current and the new tasks since there is no context switch. Otherwise we describe the state of the current task (its TCB and stack in memory) using $\mathsf{CurT}(t_c, \_, \mathit{env})$, and the saved context of the new task using $\mathsf{NoCurT}(t_n, \mathit{nst})$. Due to space limitation we omit the detailed definitions here.

If we compare a_{pre} and a_{post}, we can see that t_n becomes the current task ($\mathsf{CurT}(t_n, \mathit{nst}, \mathit{env}')$), and its general registers and stack, specified by $\mathsf{Env}(\mathit{env}')$, are loaded from the saved context *nst* (*i.e.* $\mathsf{p\_env}(\mathit{env}') = \mathit{nst}$). Here $\mathsf{p\_env}(\mathit{env}')$ refers to the part of the environment that we want to save or restore as context. Correspondingly, t_c becomes non-current-thread, and part of its environment *env* at the entry of the context switch is saved, as specified by $\mathsf{NoCurT}(t_c, \mathsf{p\_env}(\mathit{env}))$.

Tactics for Automated Reasoning. Cao *et al.*[8] propose a set of practical tactics for verifiying C program in Coq.

$$
\frac{p_1 \Longrightarrow p_2}{p_1 * q' \Longrightarrow p_2 * q'}\ L_1 \qquad \frac{w = w' \qquad p_1 \Longrightarrow p_2}{(l \mapsto w) * p_1 \Longrightarrow (l \mapsto w') * p_2}\ L_2
$$

One of a important tacitc in their work is sep_cancel. It has a library, including some rules (*e.g.* L_1 and L_2 shown above), and uses the rules in library repeatedly to solve " $p \Longrightarrow q$ ". The rules L_1 and L_2 let we can use sep_cancel to eliminate the subterms, which describe the same resources, in p and q.

In our work, we add some additional rules $L_3 \sim L_9$ (defined in Fig. 12) to the library of sep_cancel. The soundness of rules $L_3 \sim L_9$ can be achieved from the properties of assertions introduced in Sec. 3.1 directly. Aftering extending the original sep_cancel tactic, we can find that the correctness proof of the following implication can be accomplished by using the extended sep_cancel directly.

$$
(\triangleright_3 \mathtt{Y} \mapsto w_1 * \triangleright_0 \mathtt{wim} \mapsto w_2 * \mathtt{r}_5 \mapsto w_3)\downarrow \Longrightarrow \triangleright_2 \mathtt{Y} \mapsto w_1 * \mathtt{wim} \mapsto w_2 * \mathtt{r}_5 \mapsto w_3
$$

Using the extended sep_cancel can greatly simplify our verification work, and improve the efficiency of the code proving.

$$\frac{w = w' \quad p_1 \Longrightarrow p_2}{(\mathtt{rn} \mapsto w) * p_1 \Longrightarrow (\mathtt{rn} \mapsto w') * p_2}\ L_3 \qquad \frac{w = w' \quad p_1 \Longrightarrow p_2}{(\triangleright_t \mathtt{st} \mapsto w) * p_1 \Longrightarrow (\triangleright_t \mathtt{st} \mapsto w) * p_2}\ L_4$$

$$\frac{w = w' \quad p_1 \Longrightarrow p_2}{(\triangleright_0 \mathtt{sr} \mapsto w)\downarrow * p_2 \Longrightarrow (\mathtt{sr} \mapsto w) * p_2}\ L_5$$

$$\frac{w = w' \quad p_1 \Longrightarrow p_2}{(\triangleright_{t+1} \mathtt{sr} \mapsto w)\downarrow * p_1 \Longrightarrow (\triangleright_t \mathtt{sr} \mapsto w')\downarrow * p_2}\ L_6$$

$$\frac{w = w' \quad p_1 \Longrightarrow p_2}{(\mathtt{rn} \mapsto w)\downarrow * p_1 \Longrightarrow \mathtt{rn} \mapsto w' * p_2}\ L_7 \qquad \frac{w = w' \quad p_1 \Longrightarrow p_2}{(l \mapsto w)\downarrow * p \Longrightarrow l \mapsto w' * (l \mapsto w') * p'}\ L_8$$

$$\frac{p_1\downarrow \Longrightarrow p_1' \quad p_2\downarrow \Longrightarrow p_2'}{(p_1 * p_2)\downarrow \Longrightarrow p_1' * p_2'}\ L_9$$

Fig. 12. Extended rules for Tactic sep_cancel

```
Theorem CtxSwitch_Correct :
    wf_cdhp spec C.
Proof.
    ... ... ...
Qed.
```

(a)

```
Lemma SwitchEntryProof :
    forall vl,
        spec |- {{ switch_entry_pre vl }}
                  f0 : switch_entry_code
                {{ switch_entry_post vl }}.
Proof.
    ... ... ...
Qed.
```

(b)

Fig. 13. Example of Context Switch Routine Proof in Coq

Proof Efforts in Coq. We omit the code that manages interrupt and float registers in the original system, which are not supported in our logic. The segment we verify has around 250 lines of assembly code, and we verify it by 6690 lines of Coq proof scripts.

We give a simple introduction to show how we verify this context switch routine in Coq using our program logic. Fig. 13 (a) is the final theorem of the correctness of context routine in Coq. The " `CtxSwitch_Correct` " is the name of this theorem. And " `wf_cdhp spec C` " is the Coq implementation of well-formed code heap " $\vdash C : \Psi$ ". Here, the code heap " `C` " is the code heap storing the context switch code, and " `spec` " is the specifications of each code block. The correctness of this theorem can be proved by the definition of well-formed code heap, which is defined as " `wf_cdhp` " in Coq. According to the definition of " $\vdash C : \Psi$ " in Fig. 10, we need to proof the correctness of each code block given specification in code heap. We use `SwitchEntry` as an example of proving the correctness of code block (instruction sequence) in Coq. Fig. 13 (b) shows the lemma describing the correctness of `SwitchEntry` in Coq. This lemma can

be viewed as the a Coq implementation of the following properties.

$$\text{for all } \iota.\ \Psi \vdash \{(\mathtt{a}_{pre}\ \iota, \mathtt{a}_{post}\ \iota)\}\ \mathtt{f}_0 : \texttt{SwitchEntry}$$

As shown in Fig. 11, the `SwitchEntry` is the entry of the context switch routine. So, it's specification is the $\mathtt{a}_{pre}$ and $\mathtt{a}_{post}$ defined before, according to the meaning of the specification of a code block. The name of this lemma is `SwitchEntryProof` in Coq. The " `switch_entry_pre` " and " `switch_entry_post` " are Coq implementations of pre- and postcondition $\mathtt{a}_{pre}$ and $\mathtt{a}_{post}$. The " `vl` " and " `f0` " are logical variables and the starting label of code block `SwitchEntry` in Coq respectively. And the code block `SwitchEntry` is represented as " `switch_entry_code` " in Coq, which is an instruction sequence extracted from code heap " `C` ".

Readers may find that the **CALL** rule in our logic, defined in Fig. 10, doesn't support the calling of the context switch routine, because the switching of return pointer isn't permitted. We don't consider to solve this problem by modifying our **CALL** rule, but address it in another way, proving the context refinement between context switch routine and its abstract assembly primitive switch. The **CALL** rule is just used for verifying the internal functions.

5 Refinement Verification for SPARCv8

Inspired by the *relational* program logic for refinement verification, *e.g.*[18, 29], we extend our program logic for SPARCv8 to support refinement verification in this section. We define the high- and low-level program in Sec. 5.1 and Sec. 5.2, and their state relation in Sec. 5.3. The refinement relation is presented in Sec. 5.4, and program logic is shown in Sec. 5.5 and Sec. 5.6. Finally, we show the logic soundness in Sec. 5.7.

5.1 High-level Pseudo-SPARCv8 Language

We choose the SPARCv8 program defined in Sec. 2 as the *low-level* program. And let's think about what the *high-level* program should look like. ***First***, the high-level program should be multithreaded, because we hope to verify the switch primitive, whose execution will set a ready thread as the current one. ***Second***, considering programmers in this level may define various abstract assembly primitives, we should just give the type of the abstract assembly primitive, which is like an interface and its implementation is remained for users to complete. ***Third***, delay buffer D is not needed in high-level program state. The special registers are usually used to record the state of the processors. So, modifying them should be carefully and hidden in the implementations of the abstract assembly primitives. ***Fourth***, the programmers in the high-level do not need to understand the full complexities of *register window* mechanism. They just need to know that we can use instruction `save` to save the contents of the current window, and use `restore` to restore the contexts of the previous procedure. According to

$$
\begin{array}{rrl}
\text{(HCode)} & \Pi & ::= (C, \Omega) \qquad\qquad \text{(CodeHeap)}\ C \in \text{Word} \rightharpoonup \text{Comm} \\
\text{(PrimSet)} & \Omega & ::= \{\mathtt{f}_1 \rightsquigarrow \Upsilon_1, \ldots, \mathtt{f}_n \rightsquigarrow \Upsilon_n\} \\
\text{(Prim)} & \Upsilon & \in \text{List Val} \to \text{HState} \to \text{HState} \to \text{Prop} \\
\text{(Comm)} & c & ::= \mathtt{i} \mid \mathtt{call\ f} \mid \mathtt{jmp\ a} \mid \mathtt{retl} \mid \mathtt{be\ f} \\
\text{(SimpIns)} & \mathtt{i} & ::= \mathtt{Psave}\ w \mid \mathtt{Prestore} \mid \mathtt{print} \mid \mathtt{ld\ a\ r}_d \mid \mathtt{st\ r}_s\ \mathtt{a} \mid \mathtt{add\ r}_s\ \mathtt{o\ r}_d \\
 & & \quad \mid \mathtt{rd\ sr\ r}_d \mid \mathtt{wr\ r}_s\ \mathtt{o\ r}_d \mid \ldots \\
\text{(HMsg)} & \alpha & ::= \tau \mid \mathsf{out}(v) \mid \mathsf{call}(\mathtt{f}, \overline{v})
\end{array}
$$

Fig. 14. Syntax of Pseudo-SPARCv8 Code

$$
\begin{array}{rrlrrl}
\text{(HProg)} & \mathbb{P} & ::= (\Pi, \mathbb{S}, \mathtt{pc}, \mathtt{npc}) & \text{(HState)} & \mathbb{S} & ::= (T, t, \mathcal{K}, M) \\
\text{(ThrdPool)} & T & ::= \{t \rightsquigarrow \mathcal{K}\}^* & \text{(ThrdLcSt)} & \mathcal{K} & ::= (\mathbb{Q}, \mathtt{pc}, \mathtt{npc}) \\
\text{(HRstate)} & \mathbb{Q} & ::= (\mathbb{R}, b, \mathbb{F}) & \text{(ProgCount)} & \mathtt{pc}, \mathtt{npc} & \in \text{Word} \\
\text{(Tid)} & t & \in \mathbb{Z} & \text{(HRegFile)} & \mathbb{R} & \in \text{HRegName} \rightharpoonup \text{Val} \\
\text{(HRegName)} & \hat{\mathtt{rn}} & ::= \mathtt{r}_0 \mid \ldots \mid \mathtt{r}_{31} \mid \mathtt{n} \mid \mathtt{z} \mid \mathtt{c} \mid \mathtt{v} & & & \\
\text{(HFrmList)} & \mathbb{F} & ::= \text{nil} \mid (b, \text{fm}_1, \text{fm}_2) :: \mathbb{F} & \text{(HFrame)} & \text{fm} & ::= [v_0, \ldots, v_7]
\end{array}
$$

Fig. 15. Machine States for Pseudo-SPARCv8 Code

these considerations, we define the Pseudo-SPARCv8 program as our high-level program in this subsection.

We define the syntax of the high-level Pseudo-SPARCv8 language in Fig. 14. The code Π in high-level has two parts : the code heap C, whose definition is similar with the one defined in Fig. 2; and a set of abstract primitives Ω, which is a partial mapping from labels to abstract assembly primitive. The code heap C in Π can be viewed as the client code to call abstract assembly primitive. The high-level abstract assembly primitive Υ is defined as a relation that takes a list of value as arguments and maps a high-level program state (defined in Fig. 15) to another. Comparing the simple instruction with the one shown in Fig. 2, we add three pesudo instructions. The " `Psave` w " is used to allocate a new stack frame of size w words for procedure. The `Prestore` is responsible for freeing the current stack frame and restore the contexts of the previous procedure. We also introduce a pseudo instruction `print`, whose execution will occur an output $\mathsf{out}(v)$, to replace the system call of function print. The high-level message α can be either an empty message τ, or an output $\mathsf{out}(v)$, or a $\mathsf{call}(\mathtt{f}, \overline{v})$ meaning to call primitive labelled `f` with arguments $\overline{v}$.

The machine states of high-level Pseudo-SPARCv8 program is defined in Fig. 15. The high-level program $\mathbb{P}$ is a a pair of high-level code Π and high-level state $\mathbb{S}$. High-level program state is a tuple including: a thread pool T, current thread id t, the thread local state $\mathcal{K}$ of the current thread, and the memory M.

Thread Local State. The thread local state $\mathcal{K}$ is a triple of high-level register state $\mathbb{Q}$, and program conunters `pc` and `npc`. The high-level register state $\mathbb{Q}$ consists : the high-level register file $\mathbb{R}$, the block of the current stack frame b, and the high-level frame list $\mathbb{F}$. We can find that the high-level register names $\hat{\mathtt{rn}}$ do not include the `cwp` and special registers `sr`, because we require that only the implementation of the abstract assembly primitives will operate them. So, we also *do not* give the semantics for instructions, like `wr`, `rd`, `save`, and

$$\frac{\Pi = (C, \Omega) \qquad C \Vdash (\mathcal{K}, M) \circ\xrightarrow{\tau} (\mathcal{K}', M')}{(\Pi, (T, t, \mathcal{K}, M)) :\Longrightarrow (\Pi, (T, t, \mathcal{K}', M'))} \qquad \frac{\Pi = (C, \Omega) \qquad C \Vdash (\mathcal{K}, M) \circ\xrightarrow{\mathsf{out}(v)} (\mathcal{K}', M)}{(\Pi, (T, t, \mathcal{K}, M)) :\xRightarrow{\mathsf{out}(v)} (\Pi, (T, t, \mathcal{K}', M))}$$

$$\frac{\begin{array}{c}\Pi = (C, \Omega) \qquad C \Vdash (\mathcal{K}, M) \circ\xrightarrow{\mathsf{call}(\mathtt{f}, \overline{v})} (\mathcal{K}', M) \qquad \Omega(\mathtt{f}) = \Upsilon \\ \Upsilon\,(\overline{v})(T, t, \mathcal{K}', M)(T', t', \mathcal{K}'', M')\end{array}}{(\Pi, (T, t, \mathcal{K}, M)) :\Longrightarrow (\Pi, (T', t', \mathcal{K}'', M'))}$$

(a) High-level Program Transition

$$\frac{C(\mathtt{pc}) = \mathtt{i} \qquad \mathbf{execi}(\mathtt{i}, (\mathbb{Q}, M)) =_{\mathsf{H}} (\mathbb{Q}', M')}{C \Vdash ((\mathbb{Q}, \mathtt{pc}, \mathtt{npc}), M) \circ\xrightarrow{\tau} ((\mathbb{Q}', \mathtt{npc}, \mathtt{npc} + 4), M')}$$

$$\frac{C(\mathtt{pc}) = \mathtt{call\ f} \quad \mathtt{r}_{15} \in \mathrm{dom}(\mathbb{R})}{C \Vdash (((\mathbb{R}, b, \mathbb{F}), \mathtt{pc}, \mathtt{npc}), M) \circ\xrightarrow{\tau} (((\mathbb{R}\{\mathtt{r}_{15} \rightsquigarrow \mathtt{pc}\}, b, \mathbb{F}), \mathtt{npc}, \mathtt{f}), M)}$$

$$\frac{C(\mathtt{pc}) = \mathtt{retl} \qquad \mathbb{R}(\mathtt{r}_{15}) = \mathtt{f}}{C \Vdash (((\mathbb{R}, b, \mathbb{F}), \mathtt{pc}, \mathtt{npc}), M) \circ\xrightarrow{\tau} (((\mathbb{R}, b, \mathbb{F}), \mathtt{npc}, \mathtt{f} + 8), M)}$$

$$\frac{C(\mathtt{pc}) = \mathtt{print} \qquad \mathbb{R}(\%\mathtt{o}_0) = v}{C \Vdash (((\mathbb{R}, b, \mathbb{F}), \mathtt{pc}, \mathtt{npc}), M) \circ\xrightarrow{\mathsf{out}(v)} (((\mathbb{R}, b, \mathbb{F}), \mathtt{npc}, \mathtt{npc} + 4), M)}$$

$$\frac{\mathtt{pc} \notin \mathrm{dom}(C) \qquad \mathtt{npc} = \mathtt{pc} + 4 \qquad \mathbf{args}(\mathbb{Q}, M, \overline{v})}{C \Vdash ((\mathbb{Q}, \mathtt{pc}, \mathtt{npc}), M) \circ\xrightarrow{\mathsf{call}(\mathtt{pc}, \overline{v})} ((\mathbb{Q}, \mathtt{pc}, \mathtt{npc}), M)}$$

(b) High-level Control Transfer Instruction Transition

$$\frac{\begin{array}{c}\mathbb{Q} = (\mathbb{R}, b_0, \mathbb{F}) \qquad \mathbb{R}' = \mathbb{R}\{\mathsf{out} \rightsquigarrow \_, \mathsf{local} \rightsquigarrow \_, \mathsf{in} \rightsquigarrow \mathbb{R}([\mathsf{out}])\}\{\%\mathtt{sp} \rightsquigarrow (b, 0)\} \\ \boldsymbol{alloc}(M, b, 64, w) = M' \qquad \mathbb{Q}' = (\mathbb{R}', b, (b_0, \mathbb{R}([\mathsf{local}]), \mathbb{R}([\mathsf{in}])) :: \mathbb{F})\end{array}}{\mathbf{execi}(\mathtt{Psave}\ w, (\mathbb{Q}, M)) =_{\mathsf{H}} (\mathbb{Q}', M')}$$

$$\frac{\begin{array}{c}\mathbb{Q} = (\mathbb{R}, b, (b_0, \mathrm{fm}_1, \mathrm{fm}_2) :: \mathbb{F}) \qquad \boldsymbol{free}(b, M) = M' \\ \mathbb{R}' = \mathbb{R}\{\mathsf{out} \rightsquigarrow \mathbb{R}([\mathsf{in}]), \mathsf{local} \rightsquigarrow \mathrm{fm}_1, \mathsf{in} \rightsquigarrow \mathrm{fm}_2\} \qquad \mathbb{Q}' = (\mathbb{R}', b_0, \mathbb{F})\end{array}}{\mathbf{execi}(\mathtt{Prestore}, (\mathbb{Q}, M)) =_{\mathsf{H}} (\mathbb{Q}', M)}$$

(c) High-level Instruction Transition

Fig. 16. Seletcted operational semantics rules for high-level program

`restore`, whose execution will operate these registers, in high-level program. The high-level frame list $\mathbb{F}$ is also different with the F. It's a list of the triple $(b, \mathrm{fm}_1, \mathrm{fm}_2)$. Here, b is the block id of the stack frame of the previous procedure, and fm_1 and fm_2 are contexts (local and in registers) of this procedure saved. After introducing the state of high-level program, We can define switch primitive switch as an instantiation of Υ following:

$$\begin{array}{l}\mathsf{switch} ::= \lambda\,\overline{v}, \mathbb{S}, \mathbb{S}'.\ \exists\, t'.\ M(\mathsf{TaskNew}) = (t', 0)\ \wedge\ T(t') = (\mathbb{Q}', \mathtt{pc}', \mathtt{npc}') \\ \qquad\qquad\qquad\qquad \wedge\, T' = T\{t \rightsquigarrow (\mathbb{Q}, \mathtt{pc}, \mathtt{npc})\}\ \wedge\ t \neq t'\ \wedge\ \overline{v} = \mathrm{nil} \\ \quad \text{where}\ \ \mathbb{S} = (T, t, (\mathbb{Q}, \mathtt{pc}, \mathtt{npc}), M),\ \mathbb{S}' = (T', t', (\mathbb{Q}', \mathtt{f} + 8, \mathtt{f} + 12), M), \mathtt{f} = \mathbb{Q}'.\mathbb{R}(\mathtt{r}_{15}).\end{array}$$

The execution of switch primitive takes no arguments ($\overline{v} = \mathrm{nil}$), and change the id of current thread according to the pointer saved in location TaskNew.

Operational Semantics in High-level. The operational semantics for high-level Pseudo-SPARCv8 program is defined in Fig. 16. The high-level program transition relation $(\Pi, \mathbb{S}) :\stackrel{\alpha}{\Longrightarrow} (\Pi, \mathbb{S}')$ is defined in Fig. 16 (a). In each step, the program may either execute the instruction pointed by `pc`, and occur empty message τ or an output $\mathsf{out}(v)$, or call an abtract assembly primitive in primitive set. When calling an abstract assembly primitive, the execution of current thread (defined as $\_ \Vdash \_ \circ\!\!\stackrel{-}{\longrightarrow} \_$ in Fig. 16 (b)) will occur a message $\mathsf{call}(\mathtt{f}, \overline{v})$, which means that it hopes to call the abstract assembly primitive Υ labelled `f`, which is *not* in the domain of code heap C, with arguments $\overline{v}$ (we use " $\mathbf{args}(\mathbb{Q}, M, \overline{v})$ " to get arguments $\overline{v}$ from high-level state, and its definition is omitted here).

The control transfer step is defined in Fig. 16 (b). Here, the step for simple instruction `i` is represented as " $\mathbf{execi}(\mathtt{i}, \_) =_{\mathsf{H}} \_$ ". We show the state transition relation for pseudo instructions `Psave` w and `Prestore` in Fig. 16 (c). Supposing the current register state $\mathbb{Q}$ is $(\mathbb{R}, b_0, \mathbb{F})$, executing instruction `Psave` w will save the local and in registers and the block id b_0 of the current stack frame into high-level frame list $\mathbb{F}$. It also allocates a new block b as a new stack frame in memory (represented as $\boldsymbol{alloc}(M, b, 64, w) = M'$). The size of the block b is from 64 byte to w byte. The reason why it starts from 64 byte is that the 0 to 64 bytes (16 words) in a stack frame are usually reserved to save the contents of the register window in convention [4] when needed, because the number of register windows is *limited*. However, we don't need to save the register windows into memory in high-level, because they have already been saved in high-level stack frame $\mathbb{F}$, whose upper bound is *unlimited*. The instruction `Prestore` does the reverse, freeing the block of current stack frame (represented as $\boldsymbol{free}(b, M) = M'$), and restoring the contexts of the previous procedure saved in $\mathbb{F}$. More details of the high-level Pseudo SPARCv8 program can be seen in Appendix A.

5.2 Low-level SPARCv8 Program

The low-level SPARCv8 program are very closed to the SPARCv8 program defined in Fig. 2. The only difference here is that we use simple instructions and commands defined in Fig. 14. So, the global program transition of the low-level SPARCv8 program is defined as the following form :

$$\frac{(R, D) \rightrightarrows (R', D') \qquad C \vdash ((M, (R', F), D'), \mathtt{pc}, \mathtt{npc}) \circ\!\!\xrightarrow{\tau/\mathsf{out}(v)} (M', (R'', F'), D'')}{(C, (M, (R, F), D), \mathtt{pc}, \mathtt{npc}) ::\stackrel{\tau/\mathsf{out}(v)}{\Longrightarrow} (C, (M, (R'', F'), D''))}$$

Each step of the program produces either an empty message τ, or an output $\mathsf{out}(v)$, which is produced by the instruction `print` and acts as an observable behavior. More details of the low-level program can be found in Appendix B.

5.3 State Relation between Low- and High-level Program

In order to achieve refinement verification, we need to establish the states relation as an *invariant* between low- and high-level program. We illustrate this relation, defined as " $S \sim \mathbb{S}$ " formally in Fig. 18, with Fig. 17.

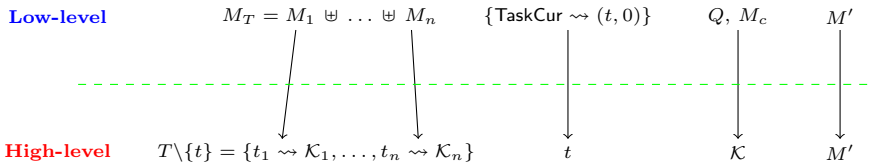

Fig. 17. State Relation between Low- and High-level Program State

$$\frac{M = M_c \uplus M_T \uplus \{\mathsf{TaskCur} \rightsquigarrow (t, 0)\} \uplus M' \quad (M_c, Q) \Downarrow_{\mathsf{c}} (t, \mathcal{K}) \quad M_T \Downarrow_{\mathsf{r}} T\backslash\{t\} \quad D = \text{nil}}{(M, Q, D) \sim (T, t, \mathcal{K}, M')}$$

Fig. 18. Relation for Whole Program State

The low-level memory M is splitted into four parts : M_c used to save the context of the current thread t; M_T saving the contexts of the ready threads; a singleton memory cell located $\mathsf{TaskCur}$ saving the current thread id; and shared memory M'. Note, ready threads are threads in thread pool, except the current thread t. The delayed buffer D is nil, because the execution of high-level program will never operate the special registers. The memory M_T used to save the contexts of the ready threads is *abstracted* as a thread pool in high-level program. Their relation is represented as " $M_T \Downarrow_{\mathsf{r}} T\backslash\{t\}$ ". We use " $(M_c, Q) \Downarrow_{\mathsf{c}} (t, \mathcal{K})$ " to represent the state relation of current thread t in low- and high-level program. Full definitions of the state relation can be found in Appendix C.

5.4 Correctness of Abstract Assembly Primitive

The correctness of abstract assembly primitive can be defined in terms of *contextual refinement*. Below we will give the formal definition in Def. 5.1. And we use *event trace refinement* proposed by Liang *et al.* [19] here.

Definition 5.1 (Primitive Correctness). $C_{\text{as}} \sqsubseteq \Omega$ iff
for any C, S, $\mathbb{S}$, $\mathtt{pc}$ and $\mathtt{npc}$, if $S \sim \mathbb{S}$, and $\mathsf{ProgSafe}(\mathbb{P})$, then $\mathrm{P} \sqsubseteq \mathbb{P}$ holds.
where $\mathrm{P} = (C \uplus C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc})$, $\mathbb{P} = ((C, \Omega), \mathbb{S})$, and $\mathbb{S}.\mathcal{K} = (\_, \mathtt{pc}, \mathtt{npc})$.

Here, we use C_{as}, which is a code heap, to represent the implementations of abstract assembly primitives and Ω to represent the set of corresponding abstract assembly primitives. The contextual refinement between C_{as} and Ω, denoted as $C_{\text{as}} \sqsubseteq \Omega$, says that if and only if for any client code C, low-level program state S, high-level program state $\mathbb{S}$, program counters $\mathtt{pc}$ and $\mathtt{npc}$, if the low- and high-level program states satisfy the state relation $S \sim \mathbb{S}$ and the high-level program will never get stuck (shown as $\mathsf{ProgSafe}(\mathbb{P})$), then there is an *event trace refinement* relation between low- and high-level program. The property $\mathsf{ProgSafe}(\mathbb{P})$, is defined below formally :

$$\mathsf{ProgSafe}(\mathbb{P}) \ ::= \forall \mathbb{P}'.\, (\mathbb{P} :\Longrightarrow^* \mathbb{P}') \Longrightarrow (\exists \mathbb{P}'.\, \mathbb{P}' :\Longrightarrow \mathbb{P}'')$$

$$\text{(EvtTrace)}\quad \mathcal{B} ::= \mathsf{out}(v) :: \mathcal{B} \mid \text{nil} \mid \mathbf{abort} \qquad \text{(co-induction)}$$

$$\frac{\nexists\, \mathrm{P}'.\ \mathrm{P} ::\Longrightarrow^{+} \mathrm{P}'}{\mathit{Etr}(\mathrm{P}, \mathbf{abort})} \qquad \frac{\mathrm{P} ::\xRightarrow{\tau}{}^{+} \mathrm{P}' \qquad \mathit{Etr}(\mathrm{P}', \text{nil})}{\mathit{Etr}(\mathrm{P}, \text{nil})} \qquad \frac{\mathrm{P} ::\xRightarrow{\mathsf{out}(v)}{}^{+} \mathrm{P}' \qquad \mathit{Etr}(\mathrm{P}', \mathcal{B})}{\mathit{Etr}(\mathrm{P}, \mathsf{out}(v) :: \mathcal{B})}$$

$$\frac{\nexists\, \mathbb{P}'.\ \mathbb{P} :\Longrightarrow^{+} \mathbb{P}'}{\mathit{Etr}(\mathbb{P}, \mathbf{abort})} \qquad \frac{\mathbb{P} :\xRightarrow{\tau}{}^{+} \mathbb{P}' \qquad \mathit{Etr}(\mathbb{P}', \text{nil})}{\mathit{Etr}(\mathbb{P}, \text{nil})} \qquad \frac{\mathbb{P} :\xRightarrow{\mathsf{out}(v)}{}^{+} \mathbb{P}' \qquad \mathit{Etr}(\mathbb{P}', \mathcal{B})}{\mathit{Etr}(\mathbb{P}, \mathsf{out}(v) :: \mathcal{B})}$$

$$\mathrm{P} \sqsubseteq \mathbb{P} \ ::= \forall \mathcal{B}.\ \mathit{Etr}(\mathrm{P}, \mathcal{B}) \Longrightarrow \mathit{Etr}(\mathbb{P}, \mathcal{B})$$

Fig. 19. Event Trace Refinement

Event Trace Refinement. We define the event trace refinement co-inductively in Fig. 19. " nil " means an empty trace. A trace is a sequence of output $\mathsf{out}(v)$, and may end with a abort marker **abort**. $\mathrm{P} \sqsubseteq \mathbb{P}$ means that all the event trace generated by the low-level program P can also generated by the high-level program $\mathbb{P}$. We introduce our extended program logic to ensure the event trace refinement between low- and high-level program in the following subsections.

5.5 Relational Assertion Language

$$\begin{array}{rl} \textit{(RelAsrt)} & \mathtt{p}, \mathtt{q} \ ::= \ \hat{\mathtt{rn}} \rightarrowtail v \mid l \rightarrowtail v \mid \mathtt{Emp} \mid t \rightsquigarrow_{\mathsf{c}} \mathcal{K} \mid t \rightsquigarrow_{\mathsf{r}} \mathcal{K} \mid p \mid (\!|A|\!) \mid \blacklozenge(w) \\ & \quad \mid \ \mathtt{p}{\downarrow} \mid \mathtt{p} \wedge \mathtt{q} \mid \mathtt{p} \vee \mathtt{q} \mid \mathtt{p} * \mathtt{q} \mid \ \ldots \end{array}$$

Fig. 20. Syntax of Relational Assertion

Fig. 20 gives the *relational* assertion language, and its semantics is given in Fig. 21. The relational assertions are interpreted over relational states $(S, \mathbb{S}, A, w)$, which contains the low-level state S, the high-level state $\mathbb{S}$, the abstract assembly primitive command A defined in Fig. 21, and a word w recording the number of the tokens. The high-level primitive command A is either an abstract assembly primitive Υ parametered with its arguments $\overline{v}$, or a $\perp$ meaning that the primitive has already been executed. Compared with the assertion defined before in Fig. 7, relational asssertion $\mathtt{p}$ reserves original assertion p describing the low-level state S, and adds some assertions describing the high-level state.

$\hat{\mathtt{rn}} \rightarrowtail v$ says that value in register $\hat{\mathtt{rn}}$ in high-level register file is v. $l \rightarrowtail v$ specifies a singleton memory cell with value v in the address of the location l in high-level memory. The assertion $\mathtt{Emp}$ says that the high-level memory and thread pool are both empty, and the low-level state satisfies emp defined in Fig. 8. The assertion $t \rightsquigarrow_{\mathsf{c}} \mathcal{K}$ and $t \rightsquigarrow_{\mathsf{r}} \mathcal{K}$ represent the thread local state of current thread and ready thread respectively. Note the threads in thread pool are viewed resources and can be separated by separation conjunction.

$$
\begin{array}{ll}
(S,\mathbb{S},A,w) \models p & ::= S \models p \wedge \mathbb{S}.M = \emptyset \wedge \mathbb{S}.T = \emptyset \\
(S,\mathbb{S},A,w) \models \hat{\mathtt{rn}} \rightarrowtail v & ::= \exists t, \mathcal{K}.\, (S,\mathbb{S},A,w) \models t \rightsquigarrow_{\mathsf{c}} \mathcal{K} \wedge \mathcal{K}.\mathbb{Q}.\mathbb{R}(\hat{\mathtt{rn}}) = v \\
(S,\mathbb{S},A,w) \models l \rightarrowtail v & ::= \mathbb{S}.M = \{l \rightsquigarrow v\} \wedge \mathbb{S}.T = \emptyset \wedge S \models \mathsf{emp} \\
(S,\mathbb{S},A,w) \models \mathtt{Emp} & ::= \mathbb{S}.M = \emptyset \wedge \mathbb{S}.T = \emptyset \wedge S \models \mathsf{emp} \\
(S,\mathbb{S},A,w) \models t \rightsquigarrow_{\mathsf{c}} \mathcal{K} & ::= \mathbb{S}.t = t \wedge \mathbb{S}.\mathcal{K} = \mathcal{K} \wedge (S,\mathbb{S},A,w) \models \mathtt{Emp} \\
(S,\mathbb{S},A,w) \models t \rightsquigarrow_{\mathsf{r}} \mathcal{K} & ::= \mathbb{S}.T = \{t \rightsquigarrow \mathcal{K}\} \wedge \mathbb{S}.M = \emptyset \wedge S \models \mathsf{emp} \\
(S,\mathbb{S},A,w) \models (\!|A'|\!) & ::= A = A' \wedge (S,\mathbb{S},A,A) \models \mathtt{Emp} \\
(S,\mathbb{S},A,w) \models \blacklozenge(w') & ::= w' \leq w \wedge (S,\mathbb{S},A,w) \models \mathtt{Emp} \\
(S,\mathbb{S},A,w) \models \mathtt{p}\downarrow & ::= \exists S'.\, ((S',\mathbb{S},A,w) \models \mathtt{p}) \wedge (R',D') \rightrightarrows (R,D) \\
 & \quad \text{where } S = (M,(R,F),D),\ S' = (M,(R',F),D') \\
(S,\mathbb{S},A,w) \models \mathtt{p} * \mathtt{q} & ::= \exists S_1,S_2,\mathbb{S}_1,\mathbb{S}_2,w_1,w_2.\, S = S_1 \uplus S_2 \wedge \mathbb{S} = \mathbb{S}_1 \uplus \mathbb{S}_2 \wedge \\
 & \quad w = w_1 + w_2 \wedge (S_1,\mathbb{S}_1,A,w_1) \models \mathtt{p} \wedge (S_2,\mathbb{S}_2,A,w_2) \models \mathtt{q}
\end{array}
$$

$$
\mathbb{S}_1 \uplus \mathbb{S}_2 \ ::= \begin{cases} (T_1 \cup T_2, t, \mathcal{K}, M_1 \cup M_2) & \textbf{if } T_1 \perp T_2 \wedge M_1 \perp M_2 \wedge \mathcal{K} = \mathcal{K}_1 \uplus \mathcal{K}_2 \\ & \quad S_1 = (T_1,t,\mathcal{K}_1,M_1) \wedge S_2 = (T_2,t,\mathcal{K}_2,M_2) \\ \textbf{undefined} & \textbf{otherwise} \end{cases}
$$

$$
\text{(HPrimCom)} \quad A \ ::= \Upsilon(\overline{v}) \mid \perp \qquad \frac{\Upsilon\,(\overline{v})(\mathbb{S})(\mathbb{S}')}{(\Upsilon(\overline{v}),\mathbb{S}) \dashrightarrow (\perp,\mathbb{S}')}
$$

Fig. 21. Semantics of Relation Assertion

The assertion $(\!|A|\!)$ means the current high-level primitive command is A. And the assertion $\blacklozenge(w)$ takes a word w to record the number of the tokens. In the introduce of the inference rules following, we use tokens to avoid infinite loops and recursive calls to make sure the termination preserving refinement.

5.6 Inference Rules for Refinement Verification

The code specification $\hat{\theta}$ and code heap specification Ψ for refinement verification are defined below :

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

Here, $\mathtt{fp}$ and $\mathtt{fq}$ are relational assertions paramterized over a list of values ι. We give a simple example in Fig. 22 to show a specification for a function, which is the same function shown in Fig. 9. We require that, in precondition $\mathtt{fp}$ and postcondition $\mathtt{fq}$, the values of the general registers $\%\mathtt{i}_0$, $\%\mathtt{i}_1$, $\%\mathtt{i}_2$, $\%\mathtt{l}_7$ and $\mathtt{r}_{15}$ are equal with the values of the same registers in high-level register file, which is restricted by state relation defined previously between low- and high-level program. The corresponding abstract assembly primitve ADD is described

$- \{(\mathtt{fp}, \mathtt{fq})\}$
`add` $\%\mathtt{i}_0, \%\mathtt{i}_1, \%\mathtt{l}_7$
`add` $\%\mathtt{l}_7, \%\mathtt{i}_2, \%\mathtt{l}_7$
`retl`
`nop`

$\mathtt{fp} ::= \lambda\, lv.\, (\%\mathtt{i}_0 = \%\mathtt{i}'_0) * (\%\mathtt{i}_1 = \%\mathtt{i}'_1) * (\%\mathtt{i}_2 = \%\mathtt{i}'_2) * (\%\mathtt{l}_7 = \%\mathtt{l}'_7) * (\mathtt{r}_{15} = \mathtt{r}'_{15}) * (\!|\mathsf{ADD(nil)}|\!)$

$\mathtt{fq} ::= \lambda\, lv.\, (\%\mathtt{i}_0 = \%\mathtt{i}'_0) * (\%\mathtt{i}_1 = \%\mathtt{i}'_1) * (\%\mathtt{i}_2 = \%\mathtt{i}'_2) * (\%\mathtt{l}_7 = \%\mathtt{l}'_7) * (\mathtt{r}_{15} = \mathtt{r}'_{15}) * (\!|\bot|\!)$

$\mathsf{ADD} ::= \lambda\, \overline{v}, \mathbb{S}, \mathbb{S}'.\, \mathbb{S}' = (T, t, ((\mathbb{R}', b, \mathbb{F}), \mathbb{R}(\mathtt{r}_{15})+8, \mathbb{R}(\mathtt{r}_{15})+12), M)$
where $\mathbb{S} = (T, t, ((\mathbb{R}, b, \mathbb{F}), \mathtt{pc}, \mathtt{npc}), M)$, $\mathbb{R}' = \mathbb{R}\{\%\mathtt{l}_7 \rightsquigarrow (\mathbb{R}(\%\mathtt{i}_0)+\mathbb{R}(\%\mathtt{i}_1)+\mathbb{R}(\%\mathtt{i}_2))\}$

$\mathtt{rn} = \hat{\mathtt{rn}}' ::= \exists\, v.\, \mathtt{rn} \mapsto v * \hat{\mathtt{rn}} \rightarrowtail v$

Fig. 22. Example for Function Specification for Refinement Verification

in the precondition $\mathtt{fp}$. In the postcondition $\mathtt{fq}$, we require that the abstract assembly primitive has been done (shown as $(\!|\bot|\!)$).

Fig. 23 shows selected inference rules for refinement verification in our logic. The top rule **WfPrim** verifies the contextual refinement between the code heap C_{as} and the corresponding abstract assembly primitive set Ω. It requires that there exisits a code heap specification Ψ_{i}, such that the internal functions in code heap C_{as} are well-formed ($\vdash C_{\text{as}} : \Psi_{\mathsf{i}}$, **WfInt**), and each implementation of abstract assembly primitive ($C_{\text{as}}[\mathtt{f}]$) is correct with respect to its corresponding primitive Υ. The $\mathsf{wdSpec}(\mathtt{fp}, \mathtt{fq}, \Upsilon)$ is used to restrict the form of the function specification, and we will discuss it in more details following. Most of the inference rules for verifying the instruction sequence ($\Psi_{\mathsf{i}} \vdash \{(\mathtt{p}, \mathtt{q})\}\, \mathtt{f} : \mathbb{I}$) are similar with the rules shown in Fig. 10. Here, we require that verifying the instruction `jmp` and `call` will consume a token, shown as $\blacklozenge(1)$, in order to avoid infinite loop and recursive function call. There is an omitted side condition in presentation that: the state of *current thread*, such as $t \rightsquigarrow_{\mathsf{c}} \mathcal{K}$, and *high-level primitive command* do not described in frame $\mathtt{p}_r$, because they are not separated by separation conjunction $*$. The **ABSCSQ** rule allows us to execute the high-level primitive command described in precondition. The implication $\mathtt{p} \Rrightarrow \mathtt{p}'$ is defined below.

$$\forall\, S, \mathbb{S}, A, w.\, ((S, \mathbb{S}, A, w) \models \mathtt{p}) \Longrightarrow (\exists\, \mathbb{S}', A', w'.\, ((A, \mathbb{S}) \dashrightarrow^* (A', \mathbb{S}')) \wedge ((S, \mathbb{S}', A', w') \models \mathtt{p}')) \vee (S, \mathbb{S}, A, w) \models \mathtt{p}'$$

The inference rules for instructions are omitted here, because they are no difference with the well-formed instruction rules shown in Fig. 10.

Well-defined Specification. We define $\mathsf{wdSpec}(\mathtt{fp}, \mathtt{fq}, \Upsilon)$ formally in Def. 5.2. It contains two properities that the specifications need to satisfy, and we explain them in turn in the following.

Definition 5.2 (Well-defined Specification). $\mathsf{wdSpec}(\mathtt{fp}, \mathtt{fq}, \Upsilon)$ holds, iff

1. for any $\overline{v}, \mathbb{S}, \mathbb{S}', \mathbb{S}_r$. if $\Upsilon(\overline{v})(\mathbb{S})(\mathbb{S}')$, and $\mathbb{S} \perp \mathbb{S}_r$, then the following holds :
   - $\mathbb{S}'.\mathcal{K}.\mathtt{pc} = \mathtt{f} + 8$, $\mathbb{S}'.\mathcal{K}.\mathtt{npc} = \mathtt{f} + 12$ (where $\mathbb{S}'.\mathcal{K}.\mathbb{Q}.\mathbb{R}(\mathtt{r}_{15}) = \mathtt{f}$);
   - there exists $\mathbb{S}'', \mathbb{S}'_r$, $\Upsilon(\overline{v})(\mathbb{S} \uplus \mathbb{S}_r)(\mathbb{S}'')$, $\mathbb{S}'' = \mathbb{S}' \uplus \mathbb{S}'_r$, and $\mathbb{S}_r.T = \mathbb{S}'_r.T$, $\mathbb{S}_r.M = \mathbb{S}'_r.M$;

$\boxed{\Psi \vdash C_{\text{as}} : \Psi_{\text{i}}}$ **(Well-formed Primitive)**

$$\frac{\begin{array}{l}\vdash C_{\text{as}} : \Psi_{\text{i}} \\ \text{for all } \mathtt{f} \in \text{dom}(\Omega),\ \iota : \\ \quad \Psi(\mathtt{f}) = (\mathtt{f}\mathtt{p}, \mathtt{f}\mathtt{q}) \quad \Omega(\mathtt{f}) = \Upsilon \quad \mathsf{wdSpec}(\mathtt{f}\mathtt{p}, \mathtt{f}\mathtt{q}, \Upsilon) \\ \quad \Psi_{\text{i}} \vdash \{(\mathtt{f}\mathtt{p}\ \iota, \mathtt{f}\mathtt{q}\ \iota)\}\,\mathtt{f} : C_{\text{as}}[\mathtt{f}]\end{array}}{\Psi \vdash C_{\text{as}} : \Omega}\ \textbf{(WfPrim)}$$

$\boxed{\vdash C_{\text{as}} : \Psi_{\text{i}}}$ **(Well-formed Internal Function)**

$$\frac{\text{for all } \mathtt{f} \in \text{dom}(\Psi_{\text{i}}),\ \iota :\ \Psi_{\text{i}}(\mathtt{f}) = (\mathtt{f}\mathtt{p}, \mathtt{f}\mathtt{q}) \quad \Psi_{\text{i}} \vdash \{(\mathtt{f}\mathtt{p}\ \iota, \mathtt{f}\mathtt{q}\ \iota)\}\,\mathtt{f} : C_{\text{as}}[\mathtt{f}]}{\vdash C_{\text{as}} : \Psi_{\text{i}}}\ \textbf{(WfInt)}$$

$\boxed{\Psi_{\text{i}} \vdash \{(\mathtt{p}, \mathtt{q})\}\,\mathtt{f} : \mathbb{I}}$ **(Well-formed Instruction Sequences)**

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

$$\frac{\begin{array}{c}\mathtt{p}\downarrow \Rightarrow\ (\mathtt{a} =_a \mathtt{f}') \quad \mathtt{f}' \in \text{dom}(\Psi_{\text{i}}) \quad \Psi_{\text{i}}(\mathtt{f}') = (\mathtt{f}\mathtt{p}, \mathtt{f}\mathtt{q}) \\ \vdash \{\mathtt{p}\downarrow\downarrow\}\,\mathtt{i}\,\{\mathtt{p}' * \blacklozenge(1)\} \quad \exists \iota, \mathtt{p}_r.\ (\mathtt{p}' \Rightarrow \mathtt{f}\mathtt{p}\ \iota * \mathtt{p}_r) \wedge (\mathtt{f}\mathtt{q}\ \iota * \mathtt{p}_r \Rrightarrow \mathtt{q})\end{array}}{\Psi_{\text{i}} \vdash \{(\mathtt{p}, \mathtt{q})\}\,\mathtt{f} : \mathtt{jmp\ a}; \mathtt{i}}\ \textbf{(JMP)}$$

$$\frac{\begin{array}{c}\mathtt{f}' \in \text{dom}(\Psi_{\text{i}}) \quad \Psi_{\text{i}}(\mathtt{f}') = (\mathtt{f}\mathtt{p}, \mathtt{f}\mathtt{q}) \quad \Psi_{\text{i}} \vdash \{(\mathtt{p}', \mathtt{q})\}\,\mathtt{f}+8 : \mathbb{I} \\ \mathtt{p}\downarrow \Rightarrow\ (\mathtt{r}_{15} \mapsto \_) * \mathtt{p}_1 \quad \vdash \{(\mathtt{r}_{15} \mapsto \mathtt{f} * \mathtt{p}_1)\downarrow\}\,\mathtt{i}\,\{\mathtt{p}_2 * \blacklozenge(1)\} \\ \exists \iota, \mathtt{p}_r.\ (\mathtt{p}_2 \Rightarrow \mathtt{f}\mathtt{p}\ \iota * \mathtt{p}_r) \wedge (\mathtt{f}\mathtt{q}\ \iota * \mathtt{p}_r \Rightarrow \mathtt{p}') \wedge (\mathtt{f}\mathtt{q}\ \iota \Rightarrow \mathtt{r}_{15} = \mathtt{f})\end{array}}{\Psi_{\text{i}} \vdash \{(\mathtt{p}, \mathtt{q})\}\,\mathtt{f} : \mathtt{call\ f}'; \mathtt{i}; \mathbb{I}}\ \textbf{(CALL)}$$

$$\frac{\mathtt{p}\downarrow\downarrow \Rightarrow (\mathtt{r}_{15} \mapsto \mathtt{f}') * \mathtt{p}_1 \quad \vdash \{\mathtt{p}_1\}\,\mathtt{i}\,\{\mathtt{p}_2\} \quad (\mathtt{r}_{15} \mapsto \mathtt{f}') * \mathtt{p}_2 \Rrightarrow \mathtt{q}}{\Psi_{\text{i}} \vdash \{(\mathtt{p}, \mathtt{q})\}\,\mathtt{f} : \mathtt{retl}; \mathtt{i}}\ \textbf{(RETL)}$$

$$\frac{\mathtt{p} \Rrightarrow \mathtt{p}' \quad \Psi_{\text{i}} \vdash \{(\mathtt{p}', \mathtt{q})\}\,\mathtt{f} : \mathbb{I}}{\Psi_{\text{i}} \vdash \{(\mathtt{p}, \mathtt{q})\}\,\mathtt{f} : \mathbb{I}}\ \textbf{(ABSCSQ)}$$

Fig. 23. Selected Inference Rules for Refinement Verification

2. for any ι, there exists $\overline{v}$, $\mathtt{f}\mathtt{p}\ \iota \Longrightarrow (\!|\Upsilon(\overline{v})|\!) * \text{true}$, and $\mathtt{f}\mathtt{q}\ \iota \Longrightarrow (\!|\bot|\!) * \text{true}$;
3. for any $\overline{v}$, there exists $\iota, \mathtt{p}_r$ and w, and the following holds:
   $\mathsf{INV}(\Upsilon(\overline{v}), w, \overline{v}) \Longrightarrow \mathtt{f}\mathtt{p}\ \iota * \mathtt{p}_r, \mathtt{f}\mathtt{q}\ \iota * \mathtt{p}_r \Longrightarrow \mathsf{INV}(\bot, \_, \_)$, and $\mathsf{Sta}(\Upsilon(\overline{v}), \mathtt{p}_r)$.

First, we should give some restrictions for abstract assembly primitive. The return code pointers should be equal to $\mathtt{f}+8$ and $\mathtt{f}+12$, where $\mathtt{f}$ is contained in $\mathtt{r}_{15}$ register after the execution of abstract assembly primitive. This restriction ensures that the low-level function and high-level abstract assembly primtive will return to the same code pointers after executions, because the **RETL** rule in our logic also restricts that the execution of low-level program will return to code pointers according to $\mathtt{r}_{15}$ register in final state of function. We also restrict that the execution of abstract assembly primitive should satisfy *locality*, which means that if an abstract assembly primitive can execute safely on a subset of program state, it can also execute safely on the whole program state, and additional

program state keeps unchanged. **Second**, the abtract assembly primitive should be specifid in the precondition, and it's execution should be done in the final state. **Third**, there is an *invariant* between low- and high-level program, holding at the entry of the function, and our logic needs to ensure that such invariant can be reestablished at the exit of function. We define INV to represent this invariant below formally:

$$\mathsf{INV}(\bot, w, \overline{v}) \ ::= \{(S, \mathbb{S}, A, w) \mid S \sim \mathbb{S} \wedge \mathbf{args}(\mathbb{S}.\mathcal{K}.\mathbb{Q}, \mathbb{S}.M, \overline{v})\}$$

$$\mathsf{INV}(A, w, \overline{v}) \ ::= \{(S, \mathbb{S}, A, w) \mid S \sim \mathbb{S} \wedge \exists \mathbb{S}'. (A, \mathbb{S}) \dashrightarrow (\bot, \mathbb{S}') \wedge \mathbf{args}(\mathbb{S}.\mathcal{K}.\mathbb{Q}, \mathbb{S}.M, \overline{v})\}$$

The invariant consists of the state relation between low- and high-level program state (define as $S \sim \mathbb{S}$ in Fig. 17), and the safe execution of the primitive command. Including the safe execution of the primitive command is important because we can get some knowledges of high-level program state from safe execution of specific primitive command A. For example, if $\mathsf{INV}(\mathsf{switch}(\mathrm{nil}), w, \mathrm{nil})$ holds, we can know that the location $\mathsf{TaskNew}$ must save a pointer pointing to a ready thread. Otherwise, the execution of primtive switch will not be safe.

$$\mathsf{INV}(\mathsf{switch}(\mathrm{nil}), w, \mathrm{nil}) \Longrightarrow \exists t, \mathcal{K}. \, \mathsf{TaskNew} \rightarrowtail (t, 0) * (t \rightsquigarrow_{\mathsf{r}} \mathcal{K}) * \mathrm{true}$$

We introduce frame $\mathtt{p}_r$ here for local reasoning, and it should be stable under the execution of the abstract assembly primitive, and we define this property $(\mathsf{Sta}(\Upsilon(\overline{v}), \mathtt{p}_r))$ formally below :

$$\begin{array}{l}\forall \mathbb{S}_c, \mathbb{S}'_c, \mathbb{S}_r, w, S, \mathtt{p}_r. \\ \quad (\Upsilon(\overline{v}), \mathbb{S}_c) \dashrightarrow (\bot, \mathbb{S}'_c) \wedge \mathbb{S}_c \perp \mathbb{S}_r \wedge (S, \mathbb{S}_r, \Upsilon(\overline{v}), w) \models \mathtt{p}_r \Longrightarrow \\ \qquad \exists \mathbb{S}'_r. (\Upsilon(\overline{v}), \mathbb{S}_c \uplus \mathbb{S}_r) \dashrightarrow (\bot, \mathbb{S}'_c \uplus \mathbb{S}'_r) \wedge \mathbb{S}'_c \perp \mathbb{S}'_r \wedge (S, \mathbb{S}'_r, \bot, w) \models \mathtt{p}_r\end{array}$$

5.7 Logic Ensuring Contextual Refinement

We give the semantics of **WfPrim** rule in Def. 5.3. It says that any high-level abstract assemly primitive can establish a simulation relation with its low-level implementation in code heap C_{as}. We define this simulation relation in Def. 5.4, which means that if there exists a relational state $(S, \mathbb{S}, A, w)$ satisfies the precondition $\mathtt{p}$, then we have the simulation $\mathtt{q} \models (C_{\mathrm{as}}, S, \mathtt{f}, \mathtt{f}+4) \preccurlyeq_i^0 (A, \mathbb{S})$ defined in Def. 5.5.

Definition 5.3 (Well-defined Primitive Set Semantics).

$$\begin{array}{l}\Psi \models C_{\mathrm{as}} : \Omega ::= \forall \mathtt{f} \in \mathrm{dom}(\Omega), \iota. \, \exists \Upsilon, \overline{v}, \mathtt{fp}, \mathtt{fq}. \\ \qquad \mathsf{wdSpec}(\mathtt{fp}, \mathtt{fq}, \Upsilon) \wedge (\mathtt{fp} \ \iota \Longrightarrow (\!| \Upsilon(\overline{v}) |\!) * \mathrm{true}) \wedge (C_{\mathrm{as}}, \mathtt{f}) \preccurlyeq^{\mathtt{fp} \ \iota, \mathtt{fq} \ \iota} \Upsilon(\overline{v}) \\ \quad \text{where } \Omega(\mathtt{f}) = \Upsilon, \Psi(\mathtt{f}) = (\mathtt{fp}, \mathtt{fq}), \text{ and } (C_{\mathrm{as}}, \mathtt{f}) \preccurlyeq^{\mathtt{p}, \mathtt{q}} A \text{ is defined Def. 5.4.}\end{array}$$

Definition 5.4 (Simulation for Implementation and Primitive).

$$\begin{array}{l}(C_{\mathrm{as}}, \mathtt{f}) \preccurlyeq^{\mathtt{p}, \mathtt{q}} A ::= \forall S, \mathbb{S}, w. (S, \mathbb{S}, A, w) \models \mathtt{p} \Longrightarrow \\ \qquad \exists i \in \mathit{Index}. \, \mathtt{q} \models (C_{\mathrm{as}}, S, \mathtt{f}, \mathtt{f}+4) \preccurlyeq_i^0 (A, \mathbb{S}) \\ \text{where } \mathtt{q} \models (C_{\mathrm{as}}, S, \mathtt{pc}, \mathtt{npc}) \preccurlyeq_i^k (A, \mathbb{S}) \text{ is defined in Def. 5.5.}\end{array}$$

Definition 5.5. Whenever $\mathfrak{q} \models (C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc}) \preccurlyeq_i^k (A, \mathbb{S})$ holds, we have the following holds :

1. if $C_{\text{as}}(\mathtt{pc}) = \mathtt{i}$, then:
   - there exists $S', \mathtt{pc}', \mathtt{npc}'$, such that $(C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc}) ::\xRightarrow{\tau} (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}')$;
   - for any $S', \mathtt{pc}', \mathtt{npc}'$, if $(C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc}) ::\xRightarrow{\tau} (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}')$, then one of the following holds:
     (a) $\exists\, j < i.\ \mathfrak{q} \models (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}') \preccurlyeq_j^k (A, \mathbb{S})$;
     (b) there exists $\mathbb{S}'$, $j \in \mathit{Index}$, such that $(A, \mathbb{S}) \dashrightarrow (\bot, \mathbb{S}')$ and $\mathfrak{q} \models (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}') \preccurlyeq_j^k (\bot, \mathbb{S}')$ holds;
2. if $C_{\text{as}}(\mathtt{pc}) = \mathtt{call\ f}$, then:
   - there exists S', $\mathtt{pc}'$, $\mathtt{npc}'$, such that $(C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc}) ::\xRightarrow{\tau}{}^2 (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}')$;
   - for any $S', \mathtt{pc}', \mathtt{npc}'$, if $(C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc}) ::\xRightarrow{\tau}{}^2 (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}')$, then one of the following holds:
     (a) $\exists\, j < i.\ \mathfrak{q} \models (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}') \preccurlyeq_j^{k+1} (A, \mathbb{S})$;
     (b) there exists $\mathbb{S}'$, $j \in \mathit{Index}$, such that $(A, \mathbb{S}) \dashrightarrow (\bot, \mathbb{S}')$ and $\mathfrak{q} \models (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}') \preccurlyeq_j^{k+1} (\bot, \mathbb{S}')$ holds;
3. if $C_{\text{as}}(\mathtt{pc}) = \mathtt{retl}$, then:
   - there exists S', $\mathtt{pc}'$, $\mathtt{npc}'$, such that $(C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc}) ::\xRightarrow{\tau}{}^2 (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}')$;
   - for any $S', \mathtt{pc}', \mathtt{npc}'$, if $(C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc}) ::\xRightarrow{\tau}{}^2 (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}')$ then there exists $j \in \mathit{Index}$, $\mathbb{S}'$ and A', such that the following holds:
     (a) either $j < i$, $\mathbb{S}' = \mathbb{S}$ and $A' = A$; or $(A, \mathbb{S}) \dashrightarrow (A', \mathbb{S}')$;
     (b) if $k = 0$, then there exists w': (where $S'.Q.R(\mathtt{r}_{15}) = \mathtt{f}$)
     $(S', \mathbb{S}', A', w') \models \mathfrak{q}$, $A' = \bot$, $\mathtt{pc}' = \mathtt{f}' + 8$, and $\mathtt{npc}' = \mathtt{f}' + 12$;
     else
     $\mathfrak{q} \models (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}') \preccurlyeq_j^{k-1} (A', \mathbb{S}')$

The definition of simulation $\mathfrak{q} \models (C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc}) \preccurlyeq_i^k (A, \mathbb{S})$ carries an index i, which is used to ensure the termination preserving, and the depth k of function call. The simulation relation can not only make sure the safe execution of low-level SPARCv8 function, which is similar with the safe defined in Def. 3.4, but also ensure the safe execution of the corresponding high-level abstract assembly primitive. The following theorem, whose correctness can be derived from Lemmas 5.6 and 5.7, shows the soundness of our logic, which means that our logic can imply the contextual refinement between implementation C_{as} and the set of abstract assembly primitives Ω.

Lemma 5.6 (Logic Ensures Simulation).

$$\Psi \vdash C_{\text{as}} : \Omega \Longrightarrow \Psi \models C_{\text{as}} : \Omega$$

Lemma 5.7 (Simulation Implies Primitive Correctness).

$$\Psi \models C_{\text{as}} : \Omega \Longrightarrow C_{\text{as}} \sqsubseteq \Omega$$

Theorem 5.8 (Logic Soundness).

$$\Psi \vdash C_{\text{as}} : \Omega \Longrightarrow C_{\text{as}} \sqsubseteq \Omega$$

6 Related Work and Conclusion

There has been much work on assembly or machine code verification. Most of them do not support function calls or simply treat function calls in the continuation-passing style where return addresses are viewed as first class code pointers [23, 5, 20, 21, 31, 24, 27]. SCAP [11] supports assembly code verification with various stack-based control abstractions, including function call and return. We follow the same idea here. However, SCAP gives a syntactic-based soundness proof by establishing the preservation of the syntactic judgment, which makes it difficult to interact with other modules verified in different logic. Since our goal is to verify inline assembly and link the verified code with the verified C programs, we give a direct-style semantic model of the logic judgments. Also SCAP is based on a simplified subset of assembly instructions, while our work is focused on a realistically modeled subset of SPARCv8 instructions.

In terms of the support of realistic instruction sets, previous work on proof-carrying code (PCC) and typed assembly language (TAL) mostly supports subsets of x86. Myreen's work [22] presents a framework for ARM verification based on a realistic model (but it doesn't support function call and return).

As part of the Foundational Proof-Carrying Code (FPCC) project [5], Tan and Appel present a program logic $\mathcal{L}_c$ for reasoning about control flow in assembly code [27]. Although $\mathcal{L}_c$ is implemented on top of SPARC machine language, the underlying logic is a type system instead of a full-blown program logic for functional correctness. It reasons about functions in the continuation-passing style. Also handling SPARC features such as delayed writes or delayed control transfers is not the focus of $\mathcal{L}_c$. There has been work on mechanized semantics of the SPARCv8 ISA. Hou *et al.* [32] model the SPARCv8 ISA in Isabelle/HOL, and test their formal model against LENON3 simulation board [1], which is a synthesisable VHDL model of a 32-bit processor compliant with the SPARC V8 architecture, through more than 100,000 instruction instances. Wang *et al.* [28] formalize its semantics in Coq. Our operational semantics of SPARCv8 follows Wang *et al.* [28]. But Wang *et al.* do not validate their formalization against actual hardware, we remain it as a future work.

Ni *et al.* [25] verify a context switch module of 19 lines in x86 code to show case the support of embedded code pointers (ECP) in XCAP [24]. The context switch module we verify comes from a practical OS kernel, which is more realistic and consists of more than 250 lines of assembly code, but our logic (**CALL** rule) does not really support the switch of return pointers, which requires further extension like OCAP [9]. Our focus is to verify the code manages the register windows, and the function calls made internally. We address this problem of calling context switch routine in another way in our work, such that we can use the extended program logic to verify the contextual refinement between context switch routine and switch primitive. The method of verifying context switch primitive in our work is inspired by the approach used by Guo *et al.* [13] that protects the TCBs through abstraction, but the state relation between low- and high-level program state in our work is more sophisticated than theirs, because of some specific mechanism, like register windows, in SPARCv8. They also do

not develop a general program logic for refinement verification of the assembly. Our extended program logic is based on the relational program logic, which has already been well used successfully in the refinement verification of C language [29, 18, 19].

Yang and Hawblitzel [30] verify Verve, an x86 implementation of an experimental operating system. Verve has two levels, the high-level TAL code and the low-level "Nucleus" that provides primitive access to hardware and memory. The Nucleus code is verified automatically using the Z3 SMT solver, while the goal of our work is to generate machine checkable proofs. Another key difference is the use of different ISAs. Here we give details to verify specific features of SPARCv8 programs.

There have been many techniques and tools proposed for automated program verification (*e.g.* [7, 6]). It is possible to adapt them to verify SPARCv8 code. We propose a new program logic and do the verification in Coq mainly because the work is part of a big project for a fully certified OS kernel for aerospace crafts whose inline assembly is written in SPARCv8. We already have a program logic implemented in Coq for C programs, which allows us to verify C code with Coq proofs. Therefore we want to have a program logic for SPARCv8 so that it can be linked with the logic for C and can generate machine-checkable Coq proofs too. That said, many of the automated verification techniques can be applied to reduce the manual efforts to write Coq proofs, which we would like to study in the future work. We have already shown that it's possible to apply some Coq tactics based on separation logic [8] in our work.

Conclusion. We present a program logic for SPARCv8. Our logic is based on a realistic semantics model and supports main features of SPARCv8, including delayed control transfer, delayed writes, and register windows. We have applied the program logic to verify the main body of the context switch routine in a realistic embedded OS kernel. And we also extend the program logic to support refinement verification. The extended program logic in our current work is designed for verifying the *contextual refinement* between the implementation of context switch routine and switch primitive mainly. It will be interesting to improve and use our extended program logic to verify the correctness of more abstract assembly primitives, like thread creating primitive spawn, and thread joining primitive join, whose execution will add or delete some threads. Our current work can only handle sequential SPARCv8 program verification, and do not consider interrupts in our machine model. Finally, the client code in our contextual refinement verification is SPARCv8 code. However, many works, like [29], use C program as the client code of calling abstract assembly primitive. we would like to link the verified inline assembly with verified C code for whole system verification in the future. Maybe it's possible to follow some complier verification works, *e.g.*[16, 14], to make sure that the behaviors of the client C code is equivalent with the complied SPARCv8 code, which acts as a client code in our work.

References

[1] Leon3. `https://www.gaisler.com/index.php/products/processors/leon3`.

[2] Linux v2.6.17.10. `https://elixir.bootlin.com/linux/v2.6.17.10/source/arch`.

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

[4] SPARC. `https://gaisler.com/doc/sparcv8.pdf`.

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

[6] J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2005.

[7] J. Berdine, C. Calcagno, and P. O'Hearn. Symbolic execution with separation logic. In *APLAS*, 2005.

[8] J. Cao, M. Fu, and X. Feng. Practical tactics for verifying c programs in coq. In *CPP*, pages 97–108, January 2015.

[9] X. Feng, Z. Ni, Z. Shao, and Y. Guo. An open framework for foundational proof-carrying code. In *TLDI*, pages 67–78, 2007.

[10] X. Feng, Z. Shao, Y. Guo, and Y. Dong. Certifying low-level programs with hardware interrupts and preemptive threads. In *PLDI*, pages 170–182, 2008.

[11] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular Verification of Assembly Code with Stack-Based Control Abstractions. In *PLDI*, June 2006.

[12] 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 *POPL*, pages 595–608, Jan 2015.

[13] Y. Guo, X. Feng, Z. Shao, and P. Shi. Modular Verification of Concurrent Thread Management. In *APLAS*, page 315–331, December 2012.

[14] H. Jiang, H. Liang, S. Xiao, J. Zha, and X. Feng. Towards Certified Separate Compilation for Concurrent Programs. In *PLDI*, pages 111–125, June 2019.

[15] 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 *SOSP*, pages 207–220, Oct 2009.

[16] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(363), December 2009.

[17] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, July 2008.

[18] H. Liang and X. Feng. Modular verification of linearizability with non-fixed linearization points. In *PLDI*, pages 459–470, June 2013.

[19] H. Liang, X. Feng, and Z. Shao. Compositional verification of termination-preserving refinement of concurrent programs. In *LICS*, July 2014.

[20] 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.

[21] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *POPL*, pages 85–97, Jan 1998.

[22] 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.

[23] 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.
[24] Z. Ni and Z. Shao. Certified Assembly Programming with Embedded Code Pointers. In *POPL*, pages 320–333, 2006.
[25] Z. Ni, D. Yu, and Z. Shao. Using XCAP to Certify Realistic Systems code: Machine context management. In *TPHOLs*, Sept 2007.
[26] J. Reynolds. Separation logic: a logic for shared mutable data structures. July 2002.
[27] G. Tan and A. W. Appel. A compositional logic for control flow. In *VMCAI*, Jan 2006.
[28] J. Wang, M. Fu, L. Qiao, and X. Feng. Formalizing SPARCv8 Instruction Set Architecture in Coq. In *SETTA*, Oct 2017.
[29] F. Xu, M. Fu, X. Feng, X. Zhang, H. Zhang, and Z. Li. A practical verification framework for preemptive os kernels. In *CAV*, pages 59–79, July 2016.
[30] J. Yang and C. Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *PLDI*, pages 99–110, 2010.
[31] 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.
[32] H. Zhe, D. Sanan, A. Tiu, Y. Liu, and K. C. Hoa. An Executable Formalisation of the SPARCv8 Instruction Set Architecture: A Case Study for the LEON3 Processor. In *FM*, 2016.

A More about High-level Instructions Execution

We give some supplements about the execution of high-level instructions. As we have explained in Sec. 5.1, the register windows and delayed buffer in pyhsical SPARCv8 program state are omitted in high-level Pseudo-SPARCv8 program state. So, we do not define state transtion rules for instructions `save`, `restore`, `rd`, and `wr`. The instruction transition rules for the rest of instructions, like `ld` and `add`, have no much difference with the rules in pyhical SPARCv8 program.

$$\frac{[\![\mathtt{a}]\!]_{\mathbb{R}} = l \qquad M(l) = v \qquad \mathbb{R}' = \mathbb{R}\{\!|\, \mathtt{r}_d \rightsquigarrow v \,|\!\}}{\mathbf{execi}(\mathtt{ld\ a\ r}_d, ((\mathbb{R}, b, \mathbb{F}), M)) =_{\mathsf{H}} ((\mathbb{R}', b, \mathbb{F}), M)}$$

$$\frac{\mathbb{R}(\mathtt{r}_s) = v_1 \qquad [\![\mathtt{o}]\!]_{\mathbb{R}} = v_2 \qquad \mathtt{r}_d = \mathrm{dom}(\mathbb{R}) \qquad \mathbb{R}' = \mathbb{R}\{\!|\, \mathtt{r}_d \rightsquigarrow v \,|\!\}}{\mathbf{execi}(\mathtt{add\ r}_s, \mathtt{o}, \mathtt{r}_d, ((\mathbb{R}, b, \mathbb{F}), M)) =_{\mathsf{H}} ((\mathbb{R}', b, \mathbb{F}), M)}$$

Fig. 24. Transition rules for instructions `ld` and `add` in high-level

We show the state transition rules for instructions `ld` and `add` in high-level in Fig. 24. The register file updating operation is defined formally below :

$$\mathbb{R}\{\!|\, \hat{\mathtt{rn}} \rightsquigarrow v \,|\!\} \ ::= \mathbb{R}\{\hat{\mathtt{rn}} \rightsquigarrow v\} \qquad \text{where } \hat{\mathtt{rn}} \notin \{\%\mathtt{sp}, \%\mathtt{fp}\}$$

According to the definition, we can find that updating the register %sp (alias of $\mathtt{r}_{14}$), which is used to point to the top of the current stack frame, and %fp (alias of $\mathtt{r}_{14}$), which is used to point to the top of the previous stack frame is not allowed. Only the execution of instructions Psave, which is used to allocate a new stack frame, and Prestore, which is used to free the current stack frame, can modify them. The evaluation of the opand and address expression in high-level is defined formally below :

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

The " Psave w " can be viewed as a macro of " save %sp, $-w$, %sp ", and " Prestore " can be viewed as a marco of " restore $\%\mathtt{g}_0, \%\mathtt{g}_0, \%\mathtt{g}_0$ ".

save	%sp, −64, %sp		Psave	w
add	$\%\mathtt{i}_0, \%\mathtt{i}_1, \%\mathtt{i}_0$		add	$\%\mathtt{i}_0, \%\mathtt{i}_1, \%\mathtt{i}_0$
ret			ret	
restore	$\%\mathtt{g}_0, \%\mathtt{g}_0, \%\mathtt{g}_0$		Prestore	
	(a)			(b)

Fig. 25. Realistic SPARCv8 Code and Pseudo-SPARCv8 Code

Fig. 25 gives a simple comparision with the realistic SPARCv8 code and our Pseudo SPARCv8 code in high-level. Fig. 25 (a) is the realistic SPARCv8 code. It uses instruction " save %sp, −64, %sp " to store the caller's context and allocate a new stack frame size 64 bytes for the current procedure, and use instruction " restore $\%\mathtt{g}_0, \%\mathtt{g}_0, \%\mathtt{g}_0$ " to restore the caller's context at the exitance of the current procedure. Fig. 25 (b) is the same function in Pseudo-SPARCv8 code, and we can find that the instructions that is responsible for saving and restoring the context of caller is replaced by " Psave w " and " Prestore ".

B More about Low-level Language

The machine states and syntax low-level SPARCv8 language (defined in Fig. 26) are taken from the model of SPARCv8 defined in Fig. 2. So, we omit some definitions, like RegName and DelayCycle, which are same as ones defined in Fig. 2 here.

The low-level program P is a tuple including the code heap C, low-level program state S, program counter pc and npc. The code heap C is defined in Fig. 14. The low-level program state S uses the block-based memory model M, which is the same as the high-level program. The low-level message does not need $\mathsf{call}(\mathtt{f}, \overline{v})$, because the low-level program does not call abstract assembly primitive, but call its corresponding implementation, which is a function.

$$
\begin{array}{rrlrrl}
\text{(LProg)} & \mathrm{P} & ::= (C, S, \mathtt{pc}, \mathtt{npc}) & \text{(LState)} & S & ::= (M, Q, D) \\
\text{(LRstate)} & Q & ::= (R, F) & \text{(LRegFile)} & R & ::= \mathrm{RegName} \rightharpoonup \mathrm{Val} \\
\text{(LFrmList)} & F & ::= \mathrm{nil} \mid \mathrm{fm} :: F & \text{(LFrame)} & \mathrm{fm} & ::= [v_0, \ldots, v_7] \\
\text{(DBuf)} & D & ::= \mathrm{nil} \mid (t, X, v) & \text{(LMsg)} & \beta & ::= \tau \mid \mathtt{out}(v)
\end{array}
$$

Fig. 26. Machine States and Syntax for Low-level SPARCv8 Language

Operational Semantics for Low-level Code. The operational semantics for low-level program is defined in Fig. 27. Most of the state transition rules are taken from Fig. 6. Here, we use "$\mathbf{execi}(\mathtt{i}, \_) =_{\mathtt{L}} \_$" to represent the step for simple instruction `i`.

The execution of instruction `Psave` is discussed in divided into two cases : (1) if we can successfully set the next register window as the current one (represented as $\mathbf{save}(R, F) = (R', F')$), a new stack frame in memory will be allocated (shown as $\boldsymbol{alloc}(M, b, 0, w) = M'$); (2) if we can't set the next register window as the current one (represented as $\mathbf{save}(R, F) = \mathbf{undefined}$), a windows overflow trap will be triggered, and we redo the instruction `Psave`. The execution of instruction `Prestore` does the reverse. Here, we use "$\_ \Uparrow \_$" to represent the state transition caused by window overflow trap and use "$\_ \Downarrow \_$" to represent the state transition caused by window underflow trap. Their formal definitions are shown in Fig. 29.

C More about State Relation Between Low- and High-level Program

After introducing the definitions of low- and high-level program, we establish the state relation between low- and high-level program in this section. Establishing their state relation is not a trivial task, because there are two major differences low- and high-level program states. **First**, all the procedures' contexts of a specific thread are saved in high-level frame list $\mathbb{F}$. However, for low-level program, part of the contexts are saved in register windows (modeled as low-level frame list F), the other part of the contexts are saved in corresponding stack frame in memory, because the number of register windows is limited; **Second**, the high-level concurrent Pseudo-SPARCv8 program is multithreaded, but the low-level SPARCv8 program does not have the concept of thread pool.

Relation for low- and high-level FrameList. The relation between low- and high-level frame list is defined in Fig. 30. We represent this relation as form "$(b, F, M_K) \Downarrow \mathbb{F}$", The tuple of b, F and M_K is the state of stack in low-level program, because, in the low-level program, part of the produces' contexts are saved in frame list F, which can also be understand as a prefix the whole frame list describe in assertion $\mathtt{cwp} \mapsto (\!| \_, F |\!)$, the other part of the contexts are saved in corresponding frame list represent as M_K. The high-level frame list $\mathbb{F}$ represents the state of stack in high-level program. Fig. 31 gives a more intuition understanding of this relation. Here, some part of the contexts F (the pink part in the left side of the Fig. 31) are saved in register windows, and the other part of contexts M_K (the green part in the left side of the Fig. 31) are saved in stack

$$\frac{(R,F) \rightrightarrows (R',F') \qquad C \vdash ((M,(R',F),D'),\mathtt{pc},\mathtt{npc}) \circ\!\xrightarrow{\beta} (M',(R'',F'),D'')}{(C,(M,(R,F),D),\mathtt{pc},\mathtt{npc}) ::\!\xRightarrow{\beta} (C,(M,(R'',F'),D''))}$$

(a) Low-level Program Transition

$$\frac{C(\mathtt{pc}) = \mathtt{i} \qquad \mathbf{execi}(\mathtt{i},(M,Q,D)) =_{\mathsf{L}} (M',Q',D')}{C \vdash ((M,Q,D),\mathtt{pc},\mathtt{npc}) \circ\!\xrightarrow{\tau} ((M',Q',D'),\mathtt{npc},\mathtt{npc}+4)}$$

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

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

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

$$\frac{C(\mathtt{pc}) = \mathtt{print} \qquad R(\%\mathtt{o}_0) = v}{C \vdash ((M,(R,F),D),\mathtt{pc},\mathtt{npc}) \circ\!\xrightarrow{\mathsf{out}(v)} ((M,(R,F),D),\mathtt{npc},\mathtt{npc}+4)}$$

$$\frac{C(\mathtt{pc}) = \mathtt{Psave}\ w \qquad \mathbf{save}(R,F) = \mathbf{undefined} \qquad (M,R,F) \Uparrow (M',R',F')}{C \vdash ((M,(R,F),D),\mathtt{pc},\mathtt{npc}) \circ\!\xrightarrow{\tau} ((M',(R',F'),D),\mathtt{pc},\mathtt{npc})}$$

$$\frac{C(\mathtt{pc}) = \mathtt{Prestore} \qquad \mathbf{restore}(R,F) = \mathbf{undefined} \qquad (M,R,F) \Downarrow (M',R',F')}{C \vdash ((M,(R,F),D),\mathtt{pc},\mathtt{npc}) \circ\!\xrightarrow{\tau} ((M',(R',F'),D),\mathtt{pc},\mathtt{npc})}$$

(b) Low-level Control Transfer Instruction Transition

$$\frac{\mathit{alloc}(M,b,0,w) = M' \qquad \mathbf{save}(R,F) = (R',F') \qquad R'' = R'\{\%\mathtt{sp} \rightsquigarrow (b,0)\}}{\mathbf{execi}(\mathtt{Psave}\ w,(M,(R,F),D)) =_{\mathsf{L}} (M',(R',F'),D)}$$

$$\frac{\mathit{free}(b,M) = M' \qquad \mathbf{restore}(R,F) = (R',F')}{\mathbf{execi}(\mathtt{Prestore},(M,(R,F),D)) =_{\mathsf{L}} (M',(R'',F'),D)}$$

$$\frac{\mathbf{save}(R,F) = (R'',F') \qquad [\![\mathtt{o}]\!]_R = v \qquad R'' = R'\{\mathtt{r}_d \rightsquigarrow R(\mathtt{r}_s)+v\}}{\mathbf{execi}(\mathtt{save\ r}_s\ \mathtt{o}\ \mathtt{r}_d,(M,(R,F),D)) =_{\mathsf{L}} (M',(R',F'),D)}$$

$$\frac{\mathbf{restore}(R,F) = (R',F') \qquad [\![\mathtt{o}]\!]_R = v \qquad R'' = R'\{\mathtt{r}_d \rightsquigarrow R(\mathtt{r}_s)+v\}}{\mathbf{execi}(\mathtt{restore\ r}_s\ \mathtt{o}\ \mathtt{r}_d,(M,(R,F),D)) =_{\mathsf{L}} (M,(R'',F'),D')}$$

(c) Low-level Instruction Transition

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

(d) Low-level Expression Semantics

Fig. 27. Selected operational semantics rules for low-level program

$$\mathbf{fresh}(b, M) \ ::= \forall\, w.\, (b, w) \notin \mathrm{dom}(M)$$

$$\begin{aligned}\textbf{\textit{alloc}}(M, b, w_l, w_h) = M' \ ::= (M' = M \ \wedge\ w_l = w_h) \vee \\ (M' = M\{(b, w_l) \rightsquigarrow \_, \ldots, (b, w_h - 1) \rightsquigarrow \_\} \ \wedge\ \mathbf{fresh}(b, M) \ \wedge\ w_l < w_h)\end{aligned}$$

$$\textbf{\textit{free}}(b, M) = M' \ ::= \forall\, b' \neq b, w'.\ M'(b', w') = M(b', w') \ \wedge\ \nexists\, w.\ (b, w) \in \mathrm{dom}(M)$$

Fig. 28. Auxiliary Definitions for Memory Operation

$$\frac{\begin{array}{c} F = F_1 \cdot \mathrm{fm}_1 \cdot \mathrm{fm}_2 \cdot \mathrm{fm}_3 \cdot \mathrm{fm}_4 \qquad \mathrm{fm}_1[6] = (b, 0) \\ R(\texttt{wim}) = 2^n \qquad \{(b_0, 0), \ldots, (b_0, 15)\} \subseteq \mathrm{dom}(M) \qquad R' = R''\{\texttt{wim} \rightsquigarrow 2^{\mathbf{next\_cwp}(n)}\} \\ M' = M\{[(b_0, 0), \ldots, (b_0, 7)] \rightsquigarrow R_0([\mathsf{local}])\}\{[(b_0, 8), \ldots, (b_0, 15)] \rightsquigarrow R_0([\mathsf{in}])\} \end{array}}{(M, (R, F)) \Uparrow (M', (R', F'))}$$

$$\frac{\begin{array}{c} F = \mathrm{fm}_1 :: \mathrm{fm}_2 :: F'' \qquad R(\mathtt{r}_{30}) = (b, 0) \qquad R(\texttt{wim}) = 2^n \\ \{[(b_0, 0), \ldots, (b_0, 7)] \rightsquigarrow \mathrm{fm}_1', [(b_0, 8), \ldots, (b_0, 15)] \rightsquigarrow \mathrm{fm}_2'\} \subseteq M' \\ R' = R''\{\texttt{wim} \rightsquigarrow 2^{\mathbf{prev\_cwp}(n)}\} \qquad F' = \mathrm{fm}_1' :: \mathrm{fm}_2' :: F'' \end{array}}{(M, (R, F)) \Downarrow\!\Downarrow (M, (R', F'))}$$

Fig. 29. Windows Over- and UnderFlow

$$\frac{}{(b, \mathrm{nil}, \emptyset) \Downarrow \mathrm{nil}} \qquad \frac{M_K = \{(b, \mathrm{fm}_1, \mathrm{fm}_2)\} \uplus M_K' \qquad \mathrm{fm}_2[6] = (b', 0) \qquad (b', \mathrm{nil}, M_K') \Downarrow \mathbb{F}}{(b, \mathrm{nil}, M_K) \Downarrow (b, \mathrm{fm}_1, \mathrm{fm}_2) :: \mathbb{F}}$$

$$\frac{\mathrm{fm}_1 = \mathrm{fm}_1' \qquad \mathrm{fm}_2 = \mathrm{fm}_2' \qquad M_K = \{(b, \_, \_)\} \uplus M_K' \qquad \mathrm{fm}_2[6] = (b', 0) \qquad (b', F, M_K') \Downarrow \mathbb{F}}{(b, \mathrm{fm}_1 :: \mathrm{fm}_2 :: F, M_K) \Downarrow (b, \mathrm{fm}_1', \mathrm{fm}_2') :: \mathbb{F}}$$

Fig. 30. Relation for low- and high-level FrameList

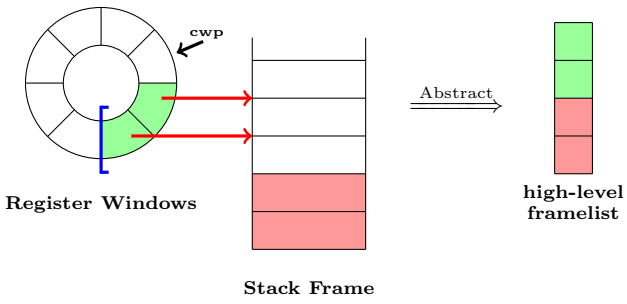

Fig. 31. Abstraction of Register Windows and Memory

frame in memory. However, in high-level state, they are abstracted as list named high-level frame list $\mathbb{F}$.

$$\mathsf{ctxfm}(R,F) ::= \begin{cases} F_1 & \textbf{if } R(\mathtt{cwp}) = w_{id},\ R(\mathtt{wim}) = 2^n,\ \mathtt{cwp} \neq n, \\ & \quad F = F_1 \cdot F_2,\ 0 \leq w_{id}, n \leq N,\ |F_1| = 2 \times (N + n - w_{id} - 1)\%N \\ \bot & \textbf{otherwise} \end{cases}$$

$$\begin{aligned} R \hookrightarrow \mathbb{R} \quad ::= & (\forall\, i \in \{0,\ldots,31\}.\, R(\mathtt{r}_i) = \mathbb{R}(\mathtt{r}_i)) \wedge (\forall\, \mathtt{sr}.\, \exists\, w.\ R(\mathtt{sr}) = w) \\ & \wedge R(\mathtt{n}) = \mathbb{R}(\mathtt{n}) \wedge R(\mathtt{z}) = \mathbb{R}(\mathtt{z}) \wedge R(\mathtt{c}) = \mathbb{R}(\mathtt{c}) \wedge R(\mathtt{v}) = \mathbb{R}(\mathtt{v}) \end{aligned}$$

$$\frac{\begin{array}{c} M_c = M_{\mathrm{ctx}} \uplus M_K \qquad \mathrm{dom}(M_{\mathrm{ctx}}) = \mathsf{DomCtxM}(t,b) \\ \mathsf{ctxfm}(R,F) = F' \qquad R(\mathtt{r}_{30}) = (b',0) \qquad (b',F',M_K) \Downarrow \mathbb{F} \qquad R \hookrightarrow \mathbb{R} \end{array}}{(M_c,(R,F)) \Downarrow_{\mathsf{c}} (t,((\mathbb{R},b,\mathbb{F}),\mathtt{pc},\mathtt{npc}))}$$

$$\frac{M_1 \Downarrow_{\mathsf{r}} T_1 \qquad M_2 \Downarrow_{\mathsf{r}} T_2}{M_1 \uplus M_2 \Downarrow_{\mathsf{r}} T_1 \uplus T_2} \qquad \frac{M \blacktriangleright_t Q \qquad (M,Q) \Downarrow_{\mathsf{c}} (t,\mathcal{K})}{M \Downarrow_{\mathsf{r}} \{t \rightsquigarrow \mathcal{K}\}}$$

Fig. 32. Relation for Thread Pool and low-level Memory

As shown in Fig. 30, if the low-level frame list F is nil and the memory is $\emptyset$, and the high-level frame list $\mathbb{F}$ is nil, it means there is no context stored. If the frame list is nil but the high-level frame list is $(b, \mathrm{fm}_1, \mathrm{fm}_2) :: \mathbb{F}$, it means that the contexts fm_1 and fm_2 are saved in stack frame in memory, whose block identifier is b. Here, we use "$\{(b, \mathrm{fm}_1, \mathrm{fm}_2)\}$" defined below to represent the part of memory saving fm_1 and fm_2. This memory contains only one block b.

$$\begin{aligned} \{(b, \mathrm{fm}_1, \mathrm{fm}_2)\} ::= & \{(b,0) \rightsquigarrow v_0, (b,4) \rightsquigarrow v_1, \ldots, (b,28) \rightsquigarrow v_7\} \\ & \uplus \{(b,32) \rightsquigarrow v'_0, (b,36) \rightsquigarrow v'_1, \ldots, (b,60) \rightsquigarrow v'_7\} \\ & \text{where } \mathrm{fm}_1 = [v_0,\ldots,v_7],\ \mathrm{fm}_2 = [v'_0,\ldots,v'_7]. \end{aligned}$$

If the frame list is $\mathrm{fm}_1 :: \mathrm{fm}_2 :: F$ and the high-level frame list is $(b', \mathrm{fm}'_1, \mathrm{fm}'_2) :: \mathbb{F}$, it means that the contexts fm_1 and fm_2 have not been saved in block b'. So, we require the contexts fm_1 and fm_2 saved in low-level frame list and the fm'_1 and fm'_2 saved in high-level frame list are equal. The block b' used to save fm_1 and fm_2 has not been used yet, so we don't care about its contents.

Relation for ThreadPool and low-level Memory. In high-level program, the thread local state of each thread is saved in a thread pool T. However, in low-level program, the local state of each thread is saved in memory (TCB and stack). For example, in Sec. 4, we introduce that the execution of the context switch module will save the register state of current thread into its TCB and stack in memory. So, the thread pool in high-level program can be viewed as an abstraction of low-level memory used to store the contexts of threads.

We define the relation between high-level thread pool and the memory used to save context in Fig. 32 formally. We use "$(M_c,(R,F)) \Downarrow_{\mathsf{c}} (t,((\mathbb{R},b,\mathbb{F}),\mathtt{pc},\mathtt{npc}))$" to represent the relation between the thread local states of *current thread* of low- and high-level program. The memory M_c owned the current thread t can be splitted into two parts M_{ctx} and M_{K}. The M_{ctx} are use the register file, whose

domain is represented as $\mathsf{DomCtxM}(t, b)$. It takes two arguments : the identifier t of the current thread and the block b of the stack frame at the top of the stack. Because the context switch module may save the register file in TCB and the stack frame of the current procedure. The other part of the memory M_K is used to save the contexts of the previous procedures, which is abstracted as $\mathbb{F}$ in high-level program. We define $R \hookrightarrow \mathbb{R}$ to represent the relation between the register file R in low-level and $\mathbb{R}$ in high-level program. The operation $\mathsf{ctxfm}(R, F)$ is used to exact the prefix F_1 of the frame list F, which saves the contexts of the previous procedures. Suppoing the value of the `cwp` is w_{id}, meaning that the id of the current window is w_{id}, and the value of the `wim` is 2^n, meaning the id n register window is invalid. According to the introduction in Fig. 2.1, we usually set a window invalid to avoid over- and underflow of the register windows. So, we known that register windows id from $(w_{id} + 1)\%N$ to $(n - 1 + N)\%N$ save the contexts of the previous procedures. So, we extract the contents F_1 of them from the whole frame list F.

We define "$M \Downarrow_{\mathsf{r}} \{t \rightsquigarrow \mathcal{K}\}$" to represent the relation between the thread local states of *ready thread* of low- and high-level program. The operation "$M \blacktriangleright_t Q$" means that we can restore the register state Q from memory M. When the context of the ready thread has been restored, we can establish a relation "$(M, Q) \Downarrow_{\mathsf{c}} (t, \mathcal{K})$" between low- and high-level thread local states of thread t. Here, we don't represent the definitions of $\mathsf{DomCtxM}(t, b)$ and $M \blacktriangleright_t Q$ here, because their definitions are based on the implementation of the context switch routine in OS kernel. And the soundness of our extended program logic does not rely on their concrete definition.

Relation for Whole Program State. Finally, we introduce the state relation for whole program states between low- and high-level program below :

$$\frac{M = M_c \uplus M_T \uplus \{\mathsf{TaskCur} \rightsquigarrow (t, 0)\} \uplus M' \quad (M_c, Q) \Downarrow_{\mathsf{c}} (t, \mathcal{K}) \quad M_T \Downarrow_{\mathsf{r}} T\backslash\{t\} \quad D = \mathrm{nil}}{(M, Q, D) \sim (T, t, \mathcal{K}, M')}$$