A Parallel Programming Language with Backward Shared Variable Holding Declaration

(Simplified Version)

Yu Zhang Chen Wang Xiaopeng Fu Wei Zhang

School of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui 230026, China
Software Security Lab., Suzhou Institute for Advanced Study, University of Science and Technology of China, Suzhou, Jiangsu 215123, China
yuzhang@ustc.edu.cn

Abstract

SPC is short for *Shared resource usage declarations based Parallel C-style language*. In this paper, the definition of a mini-subset of SPC is presented.

1. Introduction

2. Syntax of mini-SPC

The mini-SPC has only one function (*i.e.,* `main`) and one type (*i.e.,* `int`). The concrete syntax and abstract syntax of the mini-SPC are shown in Figure 1 and Figure 2 respectively.

2.1 Concrete Syntax

In this subsection, we introduce the distinctive features of the SPC language, and neglect the similar features like C language.

2.1.1 Shared Variable Declarations

A variable declaration (*vardec*) out of the function definition (*fundef*) is called global variable declaration. For simplicity, all global variable declarations are regarded as shared variable declarations.

Note: If one only wishes part of global variable declarations shared, the qualifier shared can be introduced into a global variable declaration to emphasize that the declaration is shared. However, we neglect it in this paper.

2.1.2 Parallel Statements

A parallel statement cobegin *blocklist* coend represents that all blocks in the *blocklist* will be executed concurrently. An executor executing one block of a parallel statement is called a *sub thread* of the thread executing the parallel statement. When a thread *T* executes a parallel statement *S*, it needs to fork sub threads for each block of *S*, to start up these sub threads and to wait for their completion. After all sub threads complete, the thread *T* continues executing the following statements.

[copyright notice will appear here]

2.1.3 Shared Variable Access Restrictions

In an SPC program P, a mapping from all shared variables of *P* to values is a part of the program state, and called a *shared variable state*.

An atomic statement (*atomstmt*) or a test (*bexp*) of a control flow statement will be executed atomically and called an *atomic command*. In an atomic command A, all shared variables accessed (*i.e.,* read or written) by A should be accessed under the same shared variable state.

Holding a Shared Variable. When a thread T executes a piece of code C, we say "T is ***holding*** a shared variable s" if any other thread forbids writing or accessing s. If C contains write operations on s, we say *"T is holding s for write access during executing C"*. Otherwise, we say *"T is holding s for read access during executing C"*. In SPC when a thread T is in the period of holding a shared variable s for write access, any other thread forbids holding s for write/read access; when T is in the period of holding s for read access, any other thread forbids holding s for write access, but allows holding s for read access. In mini-SPC we only consider the holding for write access. That is to say, once a thread T acquires the holding of s, other threads must not hold s until T releases the holding of s.

The SPC language introduces last expressions and skip statements to declare restrictions on holding shared variables across several *atomic commands*, which are called last *holding* and skip *holding* respectively. Moreover, skip *holding* is a special case of last *holding*.

- last : when a shared variable s qualified with last appears in an atomic command A, it means that s should be held by a thread T during its execution period from the beginning of the atomic command A' to the end of A, where A' includes T's last access to s which is relative to the access to s in A.
- skip last id_1, ..., last id_n : a skip statement is essentially an empty statement, but it could be viewed as a statement accessing last id_1, ..., last id_n in order to adjust the holding periods of shared variables.

The simplest *holding period* is a period when executing an atomic command including shared variable accesses. The longer *holding period* could be extended by using the above last holding.

2.2 Abstract Syntax

In order to facilitate discussing the semantics and program analysis of mini-SPC programs, an abstract syntax for the language (Fig-

(Program)	*program*	::=	*vardeclist fundef*
(Variable Declaration List)	*vardeclist*	::=	ϵ | *vardeclist vardec*
(Variable Declaration)	*vardec*	::=	int *idlist*;
(Identifier List)	*idlist*	::=	*id* | *idlist, id*
(Funcition Definition)	*fundef*	::=	void `main`() *block*
(Statement Block List)	*blocklist*	::=	*block* | *blocklist block*
(Statement Block)	*block*	::=	{*vardeclist stmtlist*}
(Statement List)	*stmtlist*	::=	*stmt* | *stmtlist stmt*
(Statement)	*stmt*	::=	*atomstmt*
		|	return;
		|	if (*bexp*) *stmt*
		|	if (*bexp*) *stmt* else *stmt*
		|	while (*bexp*) *stmt*
		|	*block*
		|	cobegin *blocklist* coend
(Atomic Statement)	*atomstmt*	::=	*lval*=*exp*;
		|	skip *lvallist*;
(Expression)	*exp*	::=	*lval*
		|	(*exp*)
		|	*const*
		|	−*exp*
		|	*exp abop exp*
(Boolean Expression)	*bexp*	::=	(*bexp*)
		|	! *bexp*
		|	*exp rbop exp*
		|	*bexp bbop bexp*
(Ł-value Expression List)	*lvallist*	::=	last id | *lvallist*, last id
(Ł-value Expression)	*lval*	::=	id | last
(Constant)	*const*	::=	*intconst*
(Arithmetic Binary Operator)	*abop*	::=	`+` | `-` | `*` | `/` | `%`
(Relational Binary Operator)	*rbop*	::=	`==` | `!=` | `<` | `<=` | `>` | `>=`
(Boolean Binary Operator)	*bbop*	::=	`&&` | `||` | `==` | `!=`
(Integer Constant)	*intconst*	::=	[`0-9`][`0-9`]+
(Indentifier)		::=	[`_A-Za-z`][`0-9_A-Za-z`]∗

Figure 1. The concrete syntax of the mini-SPC.

(Stmt)	S	$\in$	$\mathbb{S}_tmt$	::=	$[\mathsf{skip}]^l \mid [\widetilde{x} = E]^l$
				|	$[(\overleftarrow{x_1}, ..., \overleftarrow{x_m})]^l$
				|	$S_1; S_2 \mid \mathsf{if}([B]^l)\ S_1\ \mathsf{else}\ S_2$
				|	$\mathsf{while}([B]^l)\ S \mid S_1 \| \ldots \| S_n$
(Atom)	A	$\in$	$\mathbb{A}$	::=	$[\mathsf{skip}]^l \mid [\widetilde{x} = E]^l \mid [B]^l$
(AExp)	E	$\in$	$\mathbb{E}$	::=	$x \mid \overleftarrow{x} \mid n \mid (E) \mid -E$
				|	$E_1\ op_a\ E_2$
(BExp)	B	$\in$	$\mathbb{B}$	::=	$(B) \mid \neg B \mid E_1\ op_r\ E_2$
				|	$B_1\ op_b\ B_2$
(Lab)	l	$\in$	$\mathbb{L}$	::=	$l_1 \mid l_2 \mid, ..., \mid l_n \mid ...$
(Var)	x	$\in$	$\mathbb{V}$		
(SVar)	s	$\in$	$\mathbb{V}_s$	$\subseteq$	$\mathbb{V}$

Figure 2. The abstract syntax of the mini-SPC.

ure 2) is introduced. In the abstract syntax, all atomic commands including empty commands (skip) are labeled in order to associate data flow information with them. We will often refer to the labeled items as *elementary blocks*.

We use $\mathbb{S}_{tmt}$, $\mathbb{A}$, $\mathbb{E}$, $\mathbb{B}$ stand for the domains of statements, atomic commands, arithmetic expressions and boolean expressions respectively. We assume some countable set of variables $\mathbb{V}$ is given, and $x \in \mathbb{V}$; numerals $n \in \mathbb{N}$, labels $l \in \mathbb{L}$, arithmetic operators $op_a \in \mathbf{Op}_a$, boolean operators $op_b \in \mathbf{Op}_b$ and relational operators $op_r \in \mathbf{Op}_r$ are not be further defined in Figure 2. As to a shared variable x, $\overleftarrow{x}$ is short for last x, and $\widetilde{x}$ is short for either $\overleftarrow{x}$ or x.

A statement S can be an atomic command (A), a sequential composition ($S_1; S_2$), an if conditional, a while loop or a parallel composition ($S_1 \| \ldots \| S_n$).

An atomic command A can be an empty command (skip), a test (B), a variable assignment ($\widetilde{x} = E$, *i.e.,* $x = E$ or $\overleftarrow{x} = E$) or a skip holding ($(\overleftarrow{x_1}, ..., \overleftarrow{x_m})$). We classify them into three kinds, *i.e., tests of conditionals*(C), *tests of loops*(L) and *non-tests*($\perp$), and let $\mathbb{A}_{tag} = \{\perp, \mathsf{C}, \mathsf{L}\}$.

3. Auxiliary functions for Program Analyses

In this section, we introduce a number of auxiliary functions for program analyses.

3.1 Initial and Final Labels

Functions

$$\mathsf{init} : \mathbb{S}_{tmt} \rightarrow \mathbb{L}$$
$$\mathsf{final} : \mathbb{S}_{tmt} \rightarrow \mathcal{P}(\mathbb{L})$$

return the *initial label* and the set of *final labels* of a statement respectively. They are defined by:

$$\begin{aligned}
\mathsf{init}([A]^l) &= l\\
\mathsf{init}(S_1;S_2) &= \mathsf{init}(S_1)\\
\mathsf{init}(\mathsf{if}([B]^l)\ S_1\ \mathsf{else}\ S_2) &= l\\
\mathsf{init}(\mathsf{while}([B]^l)\ S) &= l
\end{aligned}$$

$$\begin{aligned}
\mathsf{final}([A]^l) &= \{l\}\\
\mathsf{final}(S_1;S_2) &= \mathsf{final}(S_2)\\
\mathsf{final}(\mathsf{if}([B]^l)\ S_1\ \mathsf{else}\ S_2) &= \mathsf{final}(S_1) \cup \mathsf{final}(S_2)\\
\mathsf{final}(\mathsf{while}([B]^l)\ S) &= \{l\}
\end{aligned}$$

where A is any atomic command except a test.

3.2 Elementary Blocks: Atomic Commands

To access the atomic commands (*i.e.,* the labeled items) in a program we use a function

$$\mathsf{atoms} : \mathbb{S}_{tmt} \rightarrow \mathcal{P}(\mathbb{A} \times \mathbb{A}_{tag})$$

to return all atomic commands and their kinds in a statement. We also call atomic commands *elementary blocks*. The function is defined by:

$$\begin{aligned}
\mathsf{atoms}([A]^l) &= \{([A]^l, \bot)\}\\
\mathsf{atoms}(S_1;S_2) &= \mathsf{atoms}(S_1) \cup \mathsf{atoms}(S_2)\\
\mathsf{atoms}(\mathsf{if}([B]^l)\ S_1\ \mathsf{else}\ S_2) &= \{([B]^l, \mathsf{C})\} \cup \mathsf{atoms}(S_1) \cup \mathsf{atoms}(S_2)\\
\mathsf{atoms}(\mathsf{while}([B]^l)\ S) &= \{([B]^l, \mathsf{L})\} \cup \mathsf{atoms}(S)
\end{aligned}$$

where A is any atomic command except a test.

Then the set of labels occurring in a program is given by

$$\mathsf{labels} : \mathbb{S}_{tmt} \rightarrow \mathcal{P}(\mathbb{L})$$

where

$$\mathsf{labels}(S) = \{l | ([A]^l, _) \in \mathsf{atoms}(S)\}$$

A is any atomic command, and "_" in the above definition means we do not care the kind value.

We also introduce functions

$$\mathsf{isCondTest}, \mathsf{isLoopTest} : \mathbb{L} \rightarrow \mathbf{Bool}$$

which are not further defined here, to check whether an atomic command is a conditional test or a loop test.

3.3 Flows and Reverse Flows

An *edge* (or a *flow*) from an atomic command A to another atomic command A' represents that A' can be executed immediately after A. We define a function

$$\mathsf{flow} : \mathbb{S}_{tmt} \rightarrow \mathcal{P}(\mathbb{L} \times \mathbb{L} \times \mathbb{E}_{tag})$$

to return the tagged flows of a statement, where

$$\mathbb{E}_{tag} = \{\bot, \mathsf{T}, \mathsf{F}, \mathsf{X}, \mathsf{B}, \mathsf{BX}\}$$

is the set of edge tags. We use T and F to distinguish *true* and *false* edges of a conditional, and X, B and BX to distinguish *exit*, *back*, and both edges of a loop. The flow function is defined by:

$$\begin{aligned}
\mathsf{flow}([A]^l) &= \emptyset\\
\mathsf{flow}(S_1;S_2) &= \mathsf{flow}(S_1) \cup \mathsf{flow}(S_2) \cup\\
&\quad \{(l, \mathsf{init}(S_2), \mathsf{X}) \mid\\
&\qquad l \in \mathsf{final}(S_1) \wedge \mathsf{isLoopTest}(l)\} \cup\\
&\quad \{(l, \mathsf{init}(S_2), \bot) \mid\\
&\qquad l \in \mathsf{final}(S_1) \wedge \neg\mathsf{isLoopTest}(l)\}\\
\mathsf{flow}(\mathsf{if}([B]^l)\ S_1\ \mathsf{else}\ S_2) &= \mathsf{flow}(S_1) \cup \mathsf{flow}(S_2) \cup\\
&\quad \{(l, \mathsf{init}(S_1), \mathsf{T}), (l, \mathsf{init}(S_2), \mathsf{F})\}\\
\mathsf{flow}(\mathsf{while}([B]^l)\ S) &= \mathsf{flow}(S) \cup \{(l, \mathsf{init}(S), \mathsf{T})\} \cup\\
&\quad \{(l', l, \mathsf{BX}) \mid\\
&\qquad l' \in \mathsf{final}(S) \wedge \mathsf{isLoopTest}(l')\} \cup\\
&\quad \{(l', l, \mathsf{B}) \mid\\
&\qquad l' \in \mathsf{final}(S) \wedge \neg\mathsf{isLoopTest}(l')\}
\end{aligned}$$

where A is any atomic command.

Example 1. $\mathsf{if}([B_1]^1)\ [A_1]^2\ \mathsf{else}\ [A_2]^3; \mathsf{if}([B_2]^4)\ [A_3]^5\ \mathsf{else}\ [A_4]^6$
The initial label is 1, while the final labels are $\{5,6\}$.
The flows include:
(1, 2, T) (1, 3, F) (2, 4, $\bot$) (3, 4, $\bot$)
(4, 5, T) (4, 6, F)

Example 2. $\mathsf{if}([B_1]^1)\ [A_1]^2\ \mathsf{else}\ [A_2]^3; \mathsf{while}([B_2]^4)\ [A_3]^5$
The initial label is 1, while the final labels are $\{4\}$.
The flows include:
(1, 2, T) (1, 3, F) (2, 4, $\bot$) (3, 4, $\bot$)
(4, 5, T) (5, 4, B)

Example 3. $\mathsf{if}([B_1]^1)\ \mathsf{while}([B_2]^2)\ [A_1]^3\ \mathsf{else}\ [A_2]^4;$
The initial label is 1, while the final labels are $\{2,4\}$.
The flows include:
(1, 2, T) (1, 4, F) (2, 3, T) (3, 2, B)

Example 4. $\mathsf{while}([B_1]^1)\ \mathsf{if}([B_2]^2)\ [A_1]^3\ \mathsf{else}\ [A_2]^4; [A_3]^5$
The initial label is 1, while the final labels are $\{5\}$.
The flows include:
(1, 2, T) (2, 3, T) (2, 4, F) (3, 1, B)
(4, 1, B) (1, 5, X)

Example 5. $\mathsf{while}([B_1]^1)\ \mathsf{while}([B_2]^2)\ [A_1]^3;\ \mathsf{while}([B_3]^4)\ [A_2]^5$
The initial label is 1, while the final labels are $\{4\}$.
The flows include:
(1, 2, T) (2, 3, T) (3, 2, B) (2, 1, BX)
(1, 4, X) (4, 5, T) (5, 4, B)

We also introduce functions

$$\mathsf{succs}, \mathsf{preds} : \mathbb{L} \rightarrow \mathcal{P}(\mathbb{L})$$

to return the set of *successors* and *predecessors* of an atomic command in a program S.

$$\begin{aligned}
\mathsf{succs}(l) &= \{l' \mid (l, l', t) \in \mathsf{flow}(S)\}\\
\mathsf{preds}(l) &= \{l' \mid (l', l, t) \in \mathsf{flow}(S)\}
\end{aligned}$$

Obviously, for each atomic command l, it has at most 2 successors. When atomic command l has 2 successors, l must be a test, and we say l has two exits, one is tagged by T, the other is tagged by F or X or BX. Otherwise, l has only one exit, which is tagged by $\bot$ or B.

3.4 Shared Variables and Their Last Holding

Let $\mathbb{V}_s$ stand for the set of shared variables, where $\mathbb{V}_s \subseteq \mathbb{V}$. We use a function

$$\mathsf{svars} : \mathbb{L} \rightarrow \mathcal{P}(\mathbb{V}_s)$$

to return the set of shared variables appearing in an atomic command, a partial function

$$\mathsf{islasted} : \mathbb{V}_s \times \mathbb{L} \rightharpoonup \mathbf{Bool}$$

to check whether a shared variable s has at least one access with last modifier in a atomic command, a function

$$\mathsf{labels}_{sv} : \mathbb{S}_{tmt} \times \mathbb{V}_s \rightarrow \mathcal{P}(\mathbb{L})$$

to return the set of labels of atomic commands that contain accesses of a given shared variable in a program.

Last holding. In the program S of a thread, if there is a sequence of atomic commands $P = (l_1, \ldots, l_m)$, where

$$m > 1 \wedge \forall 1 \leq i < m.(l_i, l_{i+1}, _) \in \mathsf{flow}(S),$$

we call P a *flow path* of S. If there is a flow path $P = (l_1, \ldots, l_m)$ and a shared variable s, where

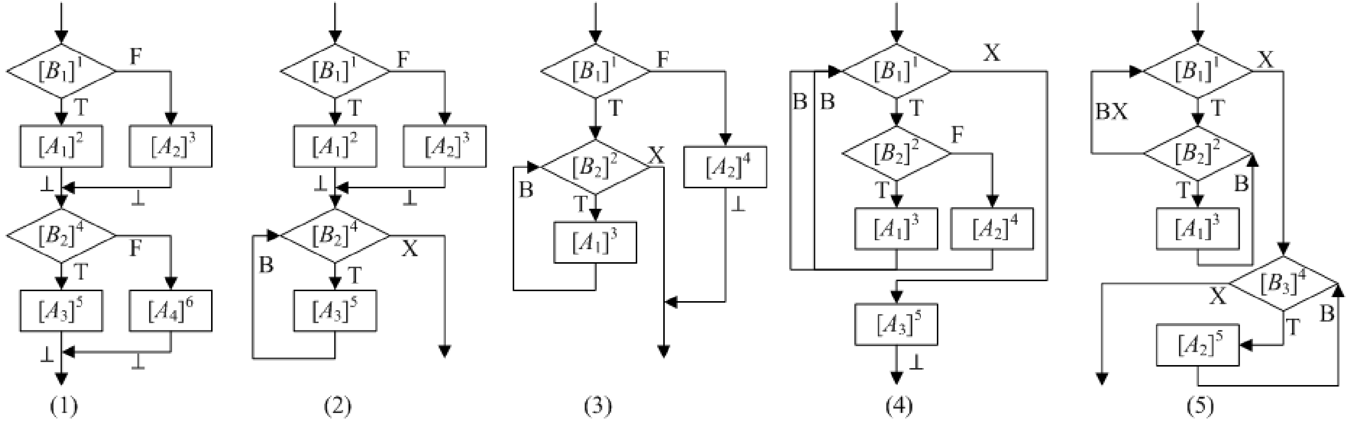


Figure 3. Some flow graphs.

$$\text{islasted}(s, l_m) \wedge s \in \text{svars}(l_1) \wedge \neg\text{islasted}(s, l_1)$$
$$\wedge(\forall 1 < i < m. s \notin \text{svars}(l_i) \vee \text{islasted}(s, l_i)),$$

we say s should be *held* by the thread when executing from the entry of l_1 to the exit of l_m.

4. Holding Analysis of Shared Variables

The *Holding Analysis of Shared Variables* will determine:

> For each atomic command A in a program, which shared variables begin to be held by current thread when program execution reaches at the entry of A; and which shared variables' holding need to be released by current thread when program execution reaches at the exit of A.

4.1 Computing Holding Information on Atomic Commands

In this section, we present a way to collect the shared variables holding information on each atomic command of a thread. It consists of the following two mappings:

$$\mathcal{H}_a : \mathbb{L} \rightarrow \mathcal{P}(\mathbb{V}_s)$$
$$\mathcal{H}_r : \mathbb{L} \rightarrow \mathcal{P}(\mathbb{V}_s \times \mathbb{E}_{tag})$$

where $\mathcal{H}_a(l)$ returns the set of shared variables to start being held by a thread when it reaches at the entry of atomic command l, and $\mathcal{H}_r(l)$ returns the set of pairs(s, t), $t \in \{\bot, \mathsf{T}, \mathsf{F}, \mathsf{X}\}$. (s, t) means the thread needs to release the holding of shared variable s at the exit of l tagged with t.

4.1.1 Phase 1. Last Holding Analysis.

The *last holding analysis* is to determine the set of held variables before or after each atomic command in a thread according to holding declarations across atomic commands by using last holding constructs. Shared variable holding inside an atomic command is not considered here.

The last holding analysis is defined in Figure 4, where S is the code of a thread T. The function

$$\text{kill}_{\text{LH}} : \mathbb{A} \rightarrow \mathcal{P}(\mathbb{V}_s)$$

produces the set of shared variables s by an atomic command $[A]^l$, which s appears in A and is not modified with last, *i.e.,* $\overleftarrow{s}$ does not occur in A.

The function

$$\text{gen}_{\text{LH}} : \mathbb{A} \rightarrow \mathcal{P}(\mathbb{V}_s)$$

produces the set of shared variables s which has at least one access with last modifier in an atomic command $[A]^l$, *i.e.,* $\overleftarrow{s}$ occurs in A.

kill and gen functions

$$\text{kill}_{\text{LH}}([A]^l) = \{s \mid s \in \text{svars}(l) \wedge \neg\text{islasted}(s, l)\}$$
$$\text{gen}_{\text{LH}}([A]^l) = \{s \mid \text{islasted}(s, l)\}$$

data flow equations: LH=

$$\text{LH}_{\text{out}}(l) = \begin{cases} \emptyset & \text{if } l \in \text{final}(S), \\ \bigcup\{\text{LH}_{\text{in}}(l') \mid (l, l', _) \in \text{flow}(S)\} & \text{otherwise} \end{cases}$$
$$\text{LH}_{\text{in}}(l) = (\text{LH}_{\text{out}}(l) \setminus \text{kill}_{\text{LH}}(A^l)) \cup \text{gen}_{\text{LH}}(A^l)$$
$$\text{where } A^l \in \text{atoms}(S)$$

Figure 4. Last holding analysis of shared variables.

$$\mathcal{H}_a(l) = \{v \mid v \notin \text{LH}_{\text{in}}(l) \wedge (v \in \text{LH}_{\text{out}}(l) \vee v \in \text{svars}(l))\}$$

$$\mathcal{H}_r(l) = \{(v, t) \mid (v \in \text{LH}_{\text{in}}(l) \vee v \in \text{svars}(l)) \wedge ((t == \bot|\mathsf{B} \wedge (l, l', t) \in \text{flow}(S) \wedge v \notin \text{LH}_{\text{out}}(l)) \vee (t == \mathsf{T}|\mathsf{F}|\mathsf{X}|\mathsf{BX} \wedge (l, l', t) \in \text{flow}(S) \wedge v \notin \text{LH}_{\text{in}}(l')))\}$$

Figure 5. Equations to compute the holding information on each atomic command.

The last holding analysis is a $backward\ analysis$, and itself is now defined by a pair of functions LH_{in} and LH_{out} mapping labels to sets of shared variables:

$$\text{LH}_{\text{in}}, \text{LH}_{\text{out}} : \mathbb{L} \rightarrow \mathcal{P}(\mathbb{V}_s),$$

which consist of the last holding information of the thread S. For each atomic command l in S, if s occurs in $\text{LH}_{\text{out}}(l)$, it means that there is at least one flow path from the exit of l containing accesses to s with last modifier; if s occurs in $\text{LH}_{\text{out}}(l)$ but not in $\text{LH}_{\text{in}}(l)$, it means s is accessed by the atomic command l without last modifier and needs to be held for its next access in one of the flow paths from l, *i.e.,* the thread needs to start a new holding of s from l.

4.1.2 Phase 2. Computing the holding information on atomic commands.

The *holding information* $\mathcal{H}_a$ and $\mathcal{H}_r$ on atomic commands will be computed according to the above *last holding information* and shared variable holding inside atomic commands. Figure 5 shows the equations to compute the *holding information* of shared variables on each atomic commands in the code of a thread.

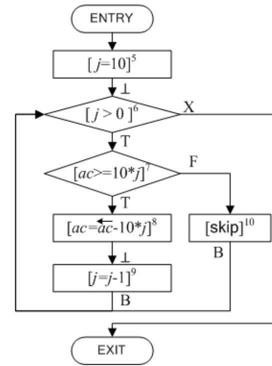
```
1    int account = 0;
2    void main(){
3      cobegin{
4      /* deposit */
5      {
6        int i;
7        i = 10;
8        while(i > 0){
9          account = account + 10 * i;
     //t1 = 10 * i; t1 = account + t1; last account = t1;
10         i = i - 1;
11       }
12     }
13     /* withdraw */
14     {
15       int j;
16       j = 10;
17       while(j > 0){
18         if(account >= 10 * j)
19         account = last account - 10;
20           j = j - 1;
21       }
22     }
23     coend
24   }
```

(a) The original source code

```
T1   /* deposit */
[i = 10]^1
while([i > 0]^2){
  [ac = ac + 10 * i]^3
  [i = i - 1]^4
}
```

```
T2   /* withdraw */
[j = 10]^5
while([j > 0]^6){
  if([ac >= 10 * j]^7)
    [ac = ←ac − 10;]^8
    [j = j − 1;]^9
  else   [skip]^10
}
```

(b) The labeled abstract code



(c) The flow graph of T_2

l	$\mathsf{kill}_{\mathsf{LH}}$	$\mathsf{gen}_{\mathsf{LH}}$	$\mathsf{LH}_{\mathsf{in}}{}^{(1)}$	$\mathsf{LH}_{\mathsf{out}}{}^{(1)}$	$\mathsf{LH}_{\mathsf{in}}$	$\mathsf{LH}_{\mathsf{out}}$	$\mathcal{H}_a$	$\mathcal{H}_r$
$5, 6, 9, 10$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
7	$\{ac\}$	$\emptyset$	$\emptyset$	$\{ac\}$	$\emptyset$	$\{ac\}$	$\{ac\}$	$\{(ac, \mathsf{F})\}$
8	$\{ac\}$	$\{ac\}$	$\{ac\}$	$\emptyset$	$\{ac\}$	$\emptyset$	$\emptyset$	$\{(ac, \perp)\}$

(d) The last holding analysis result and holding information of thread T_2

Figure 6. Example: a simple bank account program.

5. Examples

5.1 A Simple Bank Account Example

Figure **??** shows a simple bank account program with 2 threads, where *account* is a shared variable to storing the deposit account, thread T_1 executes the depositing operation and thread T_2 executes the withdrawing operation. The abstract labeled code of T_1 and T_2 is shown in Figure **??**, and Figure **??** and **??** show the flow graph and the holding analysis result of thread T_2.

6. Operational Semantics

6.1 Abstract Machine Model

Figure 7 defines the abstract machine executing the mini-SPC source code. The whole world $\mathbb{W}$ consists of a store $\mathbb{S}$, a shared variable holding set $\mathbb{HD}$ and a thread set $\mathbb{TS}$.

The store $\mathbb{S}$ is partitioned into a shared part $\mathbb{SS}$ and a thread-local part $\mathbb{LS}$, which are modeled as mappings from variables to values. We assume that all names of shared variables and thread-local variables are different from each other to simplify the memory

$$
\begin{array}{rlll}
(\mathit{World}) & \mathbb{W} & ::= & (\mathbb{S}, \mathbb{HD}, \mathbb{TS}) \\
(\mathit{ThreadSet}) & \mathbb{TS} & ::= & \emptyset \mid (T_1, \ldots, T_n) \\
(\mathit{Store}) & \mathbb{S} & ::= & \mathbb{SS} \uplus \mathbb{LS} \\
(\mathit{ThreadState}) & \mathbb{T} & := & (\mathbb{S}, \mathbb{HD}, T) \\
(\mathit{Thread}) & T_i & ::= & (\mathtt{tid}, S, \mathbb{TS}) \\
(\mathit{ThrdID}) & \mathtt{tid} & ::= & m\ (nat\ nums, and\ m > 0) \\
(\mathit{Store}) & \mathbb{SS}, \mathbb{LS} & ::= & \{x \rightsquigarrow v\}^* \\
(\mathit{HoldingSet}) & \mathbb{HD} & ::= & \{x \rightsquigarrow \mathtt{tid}\}^* \\
(\mathit{Stmt}) & S & ::= & [\mathsf{skip}]^l \mid [\widetilde{x} = E]^l \mid [(\overleftarrow{x_1}, ..., \overleftarrow{x_m})]^l \\
 & & & S_1; S_2 \mid \mathsf{if}([B]^l)\ S_1\ \mathsf{else}\ S_2 \\
 & & & \mathsf{while}([B]^l)\ S \mid S_1 || \ldots || S_n \\
(\mathit{Variable}) & x & ::= & \mathtt{id} \\
(\mathit{Value}) & v & ::= & i\ (\mathit{nat\ nums})
\end{array}
$$

Figure 7. The abstract machine.

model. The shared variable holding set $\mathbb{HD}$ is a partial mapping from shared variables to thread identifiers, *e.g.,* $s \rightsquigarrow \texttt{tid}$, which means the shared variable s is currently held by the thread $\texttt{tid}$.

The abstract machine has originally a fixed number of threads, and each thread may spawn a fixed number of sub-threads. A parent thread should wait till its all sub-threads complete. Each thread T_i contains its thread identifier $\texttt{tid}$, a piece of code *S* to be executed and a set of sub-threads $\mathbb{TS}$. *S* is any statement of mini-SPC.

6.2 Operational Semantics

The operational semantics of mini-SPC is defined in Figure 9, where auxiliary predicates mayHd, updHd, updHdT and updHdF are defined in Figure 8. Predicate mayHd is used to check whether shared variable s is not held by any other thread except thread $\texttt{tid}$. Predicates updHd, updHdT and updHdF are used to compute a new shared variable holding set at different tagged exit of an atomic command.

Figure 9 defines two kinds of transition, *i.e.,* $(\mathbb{W} \Rightarrow \mathbb{W}')$ for the whole world execution and $(\mathbb{T} \Rightarrow_{\mathsf{thrd}} \mathbb{T}')$ for the thread execution respectively. $\mathbb{S} \vdash E \triangleright v$ stands for evaluating the expression E under the store $\mathbb{S}$ and obtaining the value v. $\mathbb{S}\{x \rightsquigarrow v\}$ is used to formalize store update. $\mathsf{spawn}(S)$ stands for spawning a new sub-thread to run S. In the thread transition rules each atomic command except $[\mathsf{skip}]^l$ is executed and transformed into a special empty command skip. We merge the transition of $[\mathsf{skip}]^l$ command in the mini-SPC into rule SEQ2.

Note that if $\mathbb{T}$ does not contain any sub-threads, *i.e.,* $\mathbb{T} = (\mathbb{S}, \mathbb{HD}, (\texttt{tid}, S, \emptyset))$, or code of each sub-thread is skip, then relation $(\mathbb{T} \Rightarrow_{\mathsf{thrd}} \mathbb{T}')$ is deterministic, otherwise it is not deterministic: the state transition may be made by executing any sub-thread in $\mathbb{T}$. The semantics of the abstract machine $(\mathbb{W} \Rightarrow \mathbb{W}')$ is not deterministic either.

$$
\begin{array}{rcl}
\mathsf{mayHd}(\mathbb{HD}, \texttt{tid}, l) & \stackrel{\text{def}}{=} & \forall s \in \mathcal{H}_a(l).s \notin \mathsf{dom}(\mathbb{HD}) \\
& & \vee\, (s, \texttt{tid}) \in \mathbb{HD} \\
\mathsf{updHd}(\mathbb{HD}, \texttt{tid}, l) & \stackrel{\text{def}}{=} & \mathbb{HD} \cup \{(s, \texttt{tid}) | s \in \mathcal{H}_a(l)\} \\
& & -\, \{(s, \texttt{tid}) | (s, \bot) \in \mathcal{H}_r(l)\} \\
\mathsf{updHdT}(\mathbb{HD}, \texttt{tid}, l) & \stackrel{\text{def}}{=} & \mathbb{HD} \cup \{(s, \texttt{tid}) | s \in \mathcal{H}_a(l)\} \\
& & -\, \{(s, \texttt{tid}) | (s, \mathsf{T}) \in \mathcal{H}_r(l)\} \\
\mathsf{updHdF}(\mathbb{HD}, \texttt{tid}, l) & \stackrel{\text{def}}{=} & \mathbb{HD} \cup \{(s, \texttt{tid}) | s \in \mathcal{H}_a(l)\} \\
& & -\, \{(s, \texttt{tid}) | (s, \mathsf{F}) \in \mathcal{H}_r(l)\} \\
\mathsf{updHdX}(\mathbb{HD}, \texttt{tid}, l) & \stackrel{\text{def}}{=} & \mathbb{HD} \cup \{(s, \texttt{tid}) | s \in \mathcal{H}_a(l)\} \\
& & -\, \{(s, \texttt{tid}) | (s, \mathsf{X}) \in \mathcal{H}_r(l) \vee \\
& & \quad (s, \mathsf{BX}) \in \mathcal{H}_r(l)\}
\end{array}
$$

Figure 8. Auxiliary definitions.

$\mathbb{W} \Rightarrow \mathbb{W}'$ (*World Transition*)

$$\frac{(\mathbb{S}, \mathbb{HD}, T_i) \Rightarrow_{\mathsf{thrd}} (\mathbb{S}', \mathbb{HD}', T_i')}{(\mathbb{S}, \mathbb{HD}, (T_1, \ldots, T_i, \ldots, T_n)) \Rightarrow (\mathbb{S}', \mathbb{HD}', (T_1, \ldots, T_i', \ldots, T_n))} \text{ WORLD}$$

$\mathbb{T} \Rightarrow_{\mathsf{thrd}} \mathbb{T}'$ (*Thread Transition*)

$$\frac{(\mathbb{S}, \mathbb{HD}, (\mathtt{tid}, S_1, \emptyset)) \Rightarrow_{\mathsf{thrd}} (\mathbb{S}', \mathbb{HD}', (\mathtt{tid}, S_1', \emptyset))}{(\mathbb{S}, \mathbb{HD}, (\mathtt{tid}, S_1; S_2, \emptyset)) \Rightarrow_{\mathsf{thrd}} (\mathbb{S}', \mathbb{HD}', (\mathtt{tid}, S_1'; S_2, \emptyset))} \text{ SEQ1}$$

$$\frac{}{(\mathbb{S}, \mathbb{HD}, (\mathtt{tid}, \mathsf{skip}; S_2, \emptyset)) \Rightarrow_{\mathsf{thrd}} (\mathbb{S}, \mathbb{HD}, (\mathtt{tid}, S_2, \emptyset))} \text{ SEQ2}$$

$$\frac{\mathsf{mayHd}(\mathbb{HD}, \mathtt{tid}, l) \quad \mathbb{HD}' = \mathsf{updHd}(\mathbb{HD}, \mathtt{tid}, l)}{(\mathbb{S}, \mathbb{HD}, (\mathtt{tid}, [(\overleftarrow{x_1}, \ldots, \overleftarrow{x_m})]^l, \emptyset)) \Rightarrow_{\mathsf{thrd}} (\mathbb{S}, \mathbb{HD}', (\mathtt{tid}, \mathsf{skip}, \emptyset))} \text{ HOLD}$$

$$\frac{\mathsf{mayHd}(\mathbb{HD}, \mathtt{tid}, l) \quad \mathbb{S} \vdash E \triangleright v \quad \mathbb{HD}' = \mathsf{updHd}(\mathbb{HD}, \mathtt{tid}, l) \quad \mathbb{S}' = \mathbb{S}\{x \rightsquigarrow v\}}{(\mathbb{S}, \mathbb{HD}, (\mathtt{tid}, [\widetilde{x} = E]^l, \emptyset)) \Rightarrow_{\mathsf{thrd}} (\mathbb{S}', \mathbb{HD}', (\mathtt{tid}, \mathsf{skip}, \emptyset))} \text{ ASS}$$

$$\frac{\mathsf{mayHd}(\mathbb{HD}, \mathtt{tid}, l) \quad \mathbb{S} \vdash B \triangleright 1 \quad \mathbb{HD}' = \mathsf{updHdT}(\mathbb{HD}, \mathtt{tid}, l)}{(\mathbb{S}, \mathbb{HD}, (\mathtt{tid}, \mathsf{if}([B]^l) S_1 \text{ else } S_2, \emptyset)) \Rightarrow_{\mathsf{thrd}} (\mathbb{S}, \mathbb{HD}', (\mathtt{tid}, S_1, \emptyset))} \text{ IF1}$$

$$\frac{\mathsf{mayHd}(\mathbb{HD}, \mathtt{tid}, l) \quad \mathbb{S} \vdash B \triangleright 0 \quad \mathbb{HD}' = \mathsf{updHdF}(\mathbb{HD}, \mathtt{tid}, l)}{(\mathbb{S}, \mathbb{HD}, (\mathtt{tid}, \mathsf{if}([B]^l) S_1 \text{ else } S_2, \emptyset)) \Rightarrow_{\mathsf{thrd}} (\mathbb{S}, \mathbb{HD}', (\mathtt{tid}, S_2, \emptyset))} \text{ IF2}$$

$$\frac{\mathsf{mayHd}(\mathbb{HD}, \mathtt{tid}, l) \quad \mathbb{S} \vdash B \triangleright 1 \quad \mathbb{HD}' = \mathsf{updHdT}(\mathbb{HD}, \mathtt{tid}, l)}{(\mathbb{S}, \mathbb{HD}, (\mathtt{tid}, \mathsf{while}([B]^l) S, \emptyset)) \Rightarrow_{\mathsf{thrd}} (\mathbb{S}, \mathbb{HD}', (\mathtt{tid}, S; \mathsf{while}([B]^l) S, \emptyset))} \text{ LOOP1}$$

$$\frac{\mathsf{mayHd}(\mathbb{HD}, \mathtt{tid}, l) \quad \mathbb{S} \vdash B \triangleright 0 \quad \mathbb{HD}' = \mathsf{updHdX}(\mathbb{HD}, \mathtt{tid}, l)}{(\mathbb{S}, \mathbb{HD}, (\mathtt{tid}, \mathsf{while}([B]^l) S, \emptyset)) \Rightarrow_{\mathsf{thrd}} (\mathbb{S}, \mathbb{HD}', (\mathtt{tid}, \mathsf{skip}, \emptyset))} \text{ LOOP2}$$

$$\frac{}{(\mathbb{S}, \mathbb{HD}, (\mathtt{tid}, S_1 || \ldots || S_n; S, \emptyset)) \Rightarrow_{\mathsf{thrd}} (\mathbb{S}, \mathbb{HD}, (\mathtt{tid}, \mathsf{skip}; S, (\mathsf{spawn}(S_1), \ldots, \mathsf{spawn}(S_n)))} \text{ PAR}$$

$$\frac{(\mathbb{S}, \mathbb{HD}, T_i) \Rightarrow_{\mathsf{thrd}} (\mathbb{S}', \mathbb{HD}', T_i')}{(\mathbb{S}, \mathbb{HD}, (\mathtt{tid}, S, (T_1, \ldots, T_i, \ldots, T_k))) \Rightarrow_{\mathsf{thrd}} (\mathbb{S}', \mathbb{HD}', (\mathtt{tid}, S, (T_1, \ldots, T_i', \ldots, T_k)))} \text{ SUB-THRD1}$$

$$\frac{}{(\mathbb{S}, \mathbb{HD}, (\mathtt{tid}, S, ((\mathtt{tid}_1, \mathsf{skip}, \emptyset), \ldots, (\mathtt{tid}_k, \mathsf{skip}, \emptyset)))) \Rightarrow_{\mathsf{thrd}} (\mathbb{S}, \mathbb{HD}, (\mathtt{tid}, S, \emptyset))} \text{ SUB-THRD2}$$

Figure 9. Operational semantics of mini-SPC.