

A Parallel Language with Shared Resource Declaration for Mutable Data Structure

(Technical Report)

Yu Zhang

School of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui 230027, China
yuzhang@ustc.edu.cn

Abstract

1. The SPC-II Language

1.1 Concrete Syntax

Figure 1 shows the concrete syntax of SPC-II.

1.2 Abstract Syntax

Figure 2 shows the abstract syntax of SPC-II. A SPC-II program $\Gamma_G S$ consists of a global typing context Γ_G and a piece of code S , where Γ_G includes a set of structure type definitions and a set of variable hypotheses. In order to let programmer describe shape features of shared mutable data structures and specify sharable property of program variables, SPC-II introduces *shape declaration* into structure type definition, and *sharable tag* into variable hypotheses. So the type and effect system of SPC-II can infer the sharable effect of *lvalue* expressions and check more resource access mistakes appearing in a program, *e.g.*, ...

SPC-II includes some common statements such as assignment, control flow statements, and supports explicitly dynamic memory management. It introduces *fork-join parallel statement* for parallel programming. At the source language level each assignment or each condition test of control flow statements executes atomically and is called an *atomic command*. When executing an atomic command A , all shared resources (including declared shared variables and dynamically allocated shared locations) accessed by A should be accessed under the same *shared resource state* (*i.e.*, a mapping from all shared resources of a program to their values). Except the *atomicity* of atomic command, SPC-II requires if a definition d of a local pointer variable x assigns a shared location to x , then the shared location should not be modified or deallocated by other threads during executing statements where the definition d can reach. This requirement is called *implicit shared resource holding semantics*.

1.2.1 Types and Values

Currently SPC-II supports integer type and pointer types pointing to some structure type. The special symbol **ns*** is used to represent the type of null pointer **null**, which is compatible with an arbitrary

(Type)	$\tau ::= \mathbf{int} \mid \mathbf{s} : \gamma \prod_{f_i \in F} \langle f_i : \tau_i \rangle \mid \mathbf{s} * \mid \mathbf{ns} * \mid \mathbf{unit}$
(Shape)	$\gamma ::= \cdot \mid \mathbf{list}(f)$
(Value)	$v ::= n \mid \mathbf{null} \mid l \mid \langle f_i : v_i \rangle_{f_i \in F} \mid \mathbf{junk}$
(Loc)	$l \in \mathbf{Nat}$
(LVal)	$\ell ::= x \mid \ell \rightarrow f$
(Expr)	$E ::= v \mid \ell \mid E \mathit{bop} E \mid \mathit{uop} E$
(Bop)	$\mathit{bop} ::= + \mid - \mid * \mid / \mid \% \mid < \mid <=$ $\mid == \mid != \mid >= \mid > \mid \&\& \mid \parallel$
(Uop)	$\mathit{uop} ::= - \mid !$
(Stmt)	$S ::= [\ell := E]^l \mid [\ell := \mathbf{malloc}(s)]^l \mid [\mathbf{free}(\ell)]^l \mid [\mathbf{skip}]^l$ $\mid [\mathbf{acqv}(x)]^l \mid [\mathbf{relv}(x)]^l \mid [\mathbf{acql}(\ell)]^l \mid [\mathbf{rell}(\ell)]^l$ $\mid \mathbf{if} [E]^l S_1 S_2 \mid \mathbf{while} [E]^l S \mid S_1; S_2$ $\mid \Gamma_1 S_1 \parallel \dots \parallel \Gamma_n S_n$
(Prog)	$P ::= \Gamma_G S$
(TypeCx)	$\Gamma ::= \cdot \mid \Gamma, x : \tau \mid \delta \mid \Gamma, \mathbf{s} : \gamma \prod_{f_i \in F} \langle f_i : \tau_i \rangle$
(SharTag)	$\delta ::= \cdot \mid \mathbf{shared} \mid \mathbf{toshared} \mid \mathbf{local}$

Figure 2. Abstract Syntax of SPC-II

pointer type. **unit** stands for the type of statements. The symbol **junk**, n and l represent any uninitialized value, any integer and any memory location, respectively. The value of a memory location is a tuple $\langle f_i : v_i \rangle_{f_i \in F}$.

1.2.2 Shape Declaration

The symbol s is a meta variable which stands for any structure name. A structure definition $\mathbf{s} : \gamma \prod_{f_i \in F} \langle f_i : \tau_i \rangle$ consists of a list of fields (each has a name f_i and a type τ_i) and an optional shape declaration γ . For example,

$\mathbf{lnode} : \mathbf{list}(\mathbf{next}) \langle \mathbf{data} : \mathbf{int}, \mathbf{next} : \mathbf{lnode} * \rangle$

says that **lnode** is singly linked list node type, where the linked field is **next**.

1.2.3 Typing Context

The global typing context Γ_G of a SPC-II program consists of a set of structure type definitions and a set of variable hypotheses.

$\Gamma_1 S_1 \parallel \dots \parallel \Gamma_n S_n$ represents a fork-join parallel statement, where n sub-threads will be forked when executing it and each sub-thread $\Gamma_i S_i$ executes S_i under the thread-local typing context Γ_i , and the parallel statement ends after all sub-threads finish their

[copyright notice will appear here]

(Program)	<i>program</i>	::=	<i>globlist</i>
(Global Decl./Def. List)	<i>globlist</i>	::=	<i>global</i> <i>globalist</i> <i>global</i>
(Global Decl./Def.)	<i>global</i>	::=	<i>structdec</i> <i>vardecl</i> <i>fundef</i>
(Structure Declaration)	<i>structdec</i>	::=	struct <i>id</i> <i>shapedeclopt</i> { <i>fielddeclist</i> }
(Field Declaration List)	<i>fielddeclist</i>	::=	<i>fielddec</i> <i>fielddeclist</i> <i>fielddec</i>
(Field Declaration)	<i>fielddec</i>	::=	int <i>id</i> ; struct <i>id</i> * <i>id</i> ;
(Shape Declaration Option)	<i>shapedeclopt</i>	::=	ϵ <i>shapedecl</i>
(Shape Declaration)	<i>shapedecl</i>	::=	@ <i>id</i> (<i>idlist</i>)
(Id List)	<i>idlist</i>	::=	<i>id</i> <i>idlist</i> , <i>id</i>
(Variable Declaration List)	<i>vardeclist</i>	::=	<i>vardec</i> <i>vardeclist</i> <i>vardec</i>
(Variable Declaration)	<i>vardec</i>	::=	<i>type</i> <i>declist</i> ;
(Declarator List)	<i>declist</i>	::=	<i>dec</i> <i>declist</i> , <i>dec</i>
(Declarator)	<i>dec</i>	::=	* <i>directdec</i> <i>directdec</i> * toshared <i>directdec</i>
(Direct Declarator)	<i>directdec</i>	::=	<i>id</i> <i>shapedeclopt</i>
(Type)	<i>type</i>	::=	int struct <i>id</i>
(Function Definition)	<i>fundef</i>	::=	<i>rettype</i> <i>id</i> (<i>paramlist</i>) <i>block</i>
(Return Type)	<i>rettype</i>	::=	void int struct <i>id</i> * struct <i>id</i> * toshared
(Parameter List)	<i>paramlist</i>	::=	ϵ <i>param</i> <i>paramlist</i> , <i>param</i>
(Parameter)	<i>param</i>	::=	int <i>id</i> struct <i>id</i> * <i>id</i> struct <i>id</i> * toshared <i>id</i>
(Statement List)	<i>stmtlist</i>	::=	ϵ <i>stmtlist</i> <i>stmt</i>
(Statement)	<i>stmt</i>	::=	<i>atomstmt</i> read (<i>lval</i>); print (<i>exp</i>); <i>lval</i> = malloc (struct <i>id</i>); free (<i>lval</i>); <i>lval</i> = <i>id</i> (<i>arglist</i>); <i>id</i> (<i>arglist</i>); return <i>exp</i> ; return ; if (<i>bexp</i>) <i>stmt</i> if (<i>bexp</i>) <i>stmt</i> else <i>stmt</i> while (<i>bexp</i>) <i>stmt</i> cobegin <i>blocklist</i> coend <i>block</i>
(Atomic Statement)	<i>atomstmt</i>	::=	<i>lval</i> = <i>exp</i> ; skip <i>lvalist</i> ;
(Statement Block)	<i>block</i>	::=	{ <i>stmtlist</i> } { <i>vardeclist</i> <i>stmtlist</i> }
(Statement Block List)	<i>blocklist</i>	::=	<i>block</i> <i>blocklist</i> <i>block</i>
(Expression)	<i>exp</i>	::=	<i>lval</i> (<i>exp</i>) <i>const</i> <i>unop</i> <i>exp</i> <i>exp</i> <i>abop</i> <i>exp</i>
(Boolean Expression)	<i>bexp</i>	::=	(<i>bexp</i>) ! <i>bexp</i> <i>exp</i> <i>rbop</i> <i>exp</i> <i>bexp</i> <i>lbop</i> <i>bexp</i>
(Argument List)	<i>arglist</i>	::=	ϵ <i>explist</i>
(Expression List)	<i>explist</i>	::=	<i>exp</i> <i>explist</i> , <i>exp</i>
(L-value Expression)	<i>lval</i>	::=	<i>id</i> <i>lval</i> -> <i>id</i>
(L-valueExpression List)	<i>lvalist</i>	::=	<i>lval</i> <i>lvalist</i> , <i>lval</i>
(Constant)	<i>const</i>	::=	<i>intconst</i> null
(Unary Operator)	<i>unop</i>	::=	-
(Arithmetic Binary Operator)	<i>abop</i>	::=	+ - * / %
(Relational Binary Operator)	<i>rbop</i>	::=	== != < <= > >=
(Logical Binary Operator)	<i>lbop</i>	::=	&& == !=
(Integer Constant)	<i>intconst</i>	::=	[0-9][0-9]+
(Identifier)	<i>id</i>	::=	[_A-Za-z][0-9_A-Za-z]*

Figure 1. Concrete Syntax of SPC-II

execution. Each thread-local typing context Γ_i only includes a set of thread-local variable hypotheses.

1.2.4 Sharable Property

In a variable hypothesis $x : \tau \delta$, δ denotes the sharable tag of x . If $\delta = \mathbf{shared}$, it means x is shared and should only appear in the global typing context of a program; if $\delta = \mathbf{toshared}$, it means x is a thread-local pointer variable which may point to a shared variable or location; if δ is default or **local**, it means x is thread-local and cannot point to any shared variable or location. For convenience we assume that variable names among the global typing context and thread-local typing contexts are not equal in a SPC-II program.

According to the sharable property of pointer variables, we define *sharable effect* of a *lvalue* expression $l \rightarrow f$, that is, if l is **shared** or **toshared**, then $l \rightarrow f$ is **shared**; otherwise $l \rightarrow f$ is **local**.

The sharable effect of *lvalue* expressions affect the correctness of assignments and the behavior of **malloc** statements. For instance, an expression with **shared** or **toshared** effect cannot assign to an expression with **local** effect; and vice versa. Moreover, if the effect of l in $l := \mathbf{malloc}(E)$ statement is **local**, then the behavior is to allocate memory cell in the current thread-local heap; otherwise, is to allocate memory cell in the shared heap. Details about sharable effect can be seen in Section 1.3.5.

1.2.5 Some Notations about Typing Context

For a given typing context Γ and a structure name s , $\Gamma(s)$ says that the definition of s appears in Γ , and $\Gamma(s)(f) : \tau$ denotes that the field f of s in Γ has type τ . $\Gamma(s)[\gamma]$ denotes that the shape of s in Γ is γ , especially $\Gamma(s)[\]$ denotes s is an ordinary structure type

without shape declaration. $\Gamma(x)$ says that variable x appears in Γ and $\Gamma(x) : \tau \delta$ denotes that x in Γ has type τ and sharable tag δ .

We also introduce syntax sugar $\Gamma \setminus \delta$ to represent a typing context including all structure type definitions in Γ and all variable hypotheses which sharable tags are not δ in Γ , and $\Gamma \setminus \delta$ to represent a typing context including all structure type definitions in Γ and all variable hypotheses with δ sharable tag in Γ .

1.2.6 Holding Semantics of Shared Resources

Currently there are two kinds of shared resource holding semantics as below:

- *Atomic holding semantics*, which is introduced by the atomicity of atomic commands.
- *Implicit holding semantics*, which is introduced by the reaching definitions of **toshared** variables.

1.3 Static Semantics

Generally, a typing context Γ is assumed to be well-formed when it appears at left hand side of a judgement. The static semantics of SPC-II consists of the inductive definition of the following judgements:

$\Gamma \vdash \diamond$	typing context Γ is well-formed
$\Gamma \vdash \tau$	type τ is well-formed under Γ
$\Gamma \vdash E : \tau$	expression E is well-formed under Γ
$\Gamma \vdash S : \mathbf{unit}$	statement S is well-formed under Γ
$\emptyset_{\Gamma} \vdash P : \mathbf{unit}$	program P is well-formed

The definition of well-formed statement judgement $\Gamma \vdash S : \mathbf{unit}$ may use the following sharable effect judgement:

$$\Gamma \vdash \ell \delta \quad \textit{lvalue} \text{ expression } \ell \text{ has sharable effect } \delta \text{ under } \Gamma$$

1.3.1 Well-formed Types

The purpose of giving well-formed rules for types is that the well-formedness of structure types is context sensitive (see the rule TST), and it could not be guaranteed by the above syntactic rules.

Note: in the following definitions, **ns*** is not a well-formed type.

$$\boxed{\Gamma \vdash \tau}$$

$$\overline{\Gamma \vdash \mathbf{int}} \quad (\text{TINT})$$

$$\frac{\Gamma \vdash \mathbf{s}}{\Gamma \vdash \mathbf{s}^*} \quad (\text{TPTR})$$

$$\frac{\tau_i = \mathbf{int} \vee \tau_i = \mathbf{s}_i^* \wedge \Gamma \vdash \mathbf{s}_i \vee \tau_i = \mathbf{s}^* \quad 1 \leq i \leq |F|}{\Gamma, \mathbf{s} : \prod_{f_i \in F} \langle f_i : \tau_i \rangle \vdash \mathbf{s}} \quad (\text{TST})$$

The (TST) rule describes the well-formedness of ordinary structure types without shape declaration.

$$\frac{\begin{array}{l} i \neq k \wedge (\tau_i = \mathbf{int} \vee \tau_i = \mathbf{s}_i^* \wedge \Gamma \vdash \mathbf{s}_i) \\ i = k \wedge \tau_i = \mathbf{s}^* \quad f_k \in F \quad 1 \leq i, k \leq |F| \end{array}}{\Gamma, \mathbf{s} : \mathbf{list}(f_k) \prod_{f_i \in F} \langle f_i : \tau_i \rangle \vdash \mathbf{s}} \quad (\text{TST-LIST})$$

The (TST-LIST) rule describes the well-formedness of structure types with singly linked list shape. We also define the following predicate $\text{isShapePtr}(\Gamma, \mathbf{s}, f)$ to check whether the field f is a shape pointer or not.

$$\text{isShapePtr}(\Gamma, \mathbf{s}, f) \stackrel{\text{def}}{=} \begin{cases} \text{true} & \text{if } \Gamma(\mathbf{s})[\mathbf{list}(f)] \\ \text{false} & \text{otherwise} \end{cases}$$

1.3.2 Type Compatibility

When variables with different types τ_1 and τ_2 are allowed to be used in assignments or parameter passing, we say that τ_1 and τ_2 are compatible, denoted as $\tau_1 \cong \tau_2$. Especially, if \mathbf{s}_1 and \mathbf{s}_2 are both structure type names, $\mathbf{s}_1 \equiv \mathbf{s}_2$ means that the two strings are identical, then $\mathbf{s}_1 \cong \mathbf{s}_2$. This convention illustrates that the equivalence of structure types in SPC-II is name equivalent, but not structural equivalent.

The rules giving type compatibility are outlined below.

$$\boxed{\Gamma \vdash \tau_1 \cong \tau_2}$$

$$\overline{\Gamma \vdash \mathbf{int} \cong \mathbf{int}} \quad (\text{TCINT})$$

$$\frac{\Gamma \vdash \mathbf{s}_1 \quad \Gamma \vdash \mathbf{s}_2 \quad \mathbf{s}_1 \equiv \mathbf{s}_2}{\Gamma \vdash \mathbf{s}_1 \cong \mathbf{s}_2} \quad (\text{TCST})$$

$$\overline{\Gamma \vdash \mathbf{ns}^* \cong \mathbf{ns}^*} \quad (\text{TCNS})$$

$$\frac{\Gamma \vdash \tau_1 \cong \tau_2}{\Gamma \vdash \tau_1^* \cong \tau_2^*} \quad (\text{TCPTR1})$$

$$\frac{\Gamma \vdash \tau^*}{\Gamma \vdash \tau^* \cong \mathbf{ns}^*} \quad (\text{TCPTR2})$$

1.3.3 Well-formed Typing Context

The syntactic rules of a typing context Γ guarantee its well-formedness as a general typing context, *i.e.*, $\Gamma \vdash \diamond$. However, it is necessary to define the following two judgments for judging well-formed global typing context and well-formed thread-local typing context:

$\Gamma \vdash_G \diamond$ Γ is a well-formed global typing context

$\Gamma_G; \Gamma \vdash_T \diamond$ Γ is a well-formed thread-local typing context under the well-formed global typing context Γ_G

$$\boxed{\Gamma \vdash_G \diamond}$$

$$\overline{\emptyset_{\Gamma} \vdash_G \diamond} \quad (\text{CGEMPTY})$$

$$\frac{\Gamma \vdash_G \diamond \quad x \notin \text{dom}(\Gamma) \quad \delta = \cdot \mid \mathbf{shared} \quad \Gamma \vdash \tau}{\Gamma, x : \tau \delta \vdash_G \diamond} \quad (\text{CGVAR})$$

$$\frac{\Gamma \vdash_G \diamond \quad \mathbf{s} \notin \text{dom}(\Gamma) \quad \Gamma, \mathbf{s} : \gamma \prod_{f_i \in F} \langle f_i : \tau_i \rangle \vdash \mathbf{s}}{\Gamma, \mathbf{s} : \gamma \prod_{f_i \in F} \langle f_i : \tau_i \rangle \vdash_G \diamond} \quad (\text{CGST})$$

$$\boxed{\Gamma_G; \Gamma \vdash_T \diamond}$$

$$\frac{\Gamma_G \vdash_G \diamond}{\Gamma_G; \emptyset_{\Gamma} \vdash_T \diamond} \quad (\text{CTEMPTY})$$

$$\frac{\Gamma_G; \Gamma \vdash_T \diamond \quad x \notin \text{dom}(\Gamma_G) \uplus \text{dom}(\Gamma) \quad \Gamma_G \vdash \tau}{\Gamma_G; \Gamma, x : \tau \vdash_T \diamond} \quad (\text{CTVAR1})$$

$$\frac{\Gamma_G; \Gamma \vdash_T \diamond \quad x \notin \text{dom}(\Gamma_G) \uplus \text{dom}(\Gamma) \quad \Gamma_G \vdash \tau}{\Gamma_G; \Gamma, x : \tau^* \mathbf{toshared} \vdash_T \diamond} \quad (\text{CTVAR2})$$

1.3.4 Side Conditions

In order to guarantee language safety, SPC-II introduces side conditions into traditional typing rules. These side conditions illustrate explicit requirements posed on variables values or pointer states. In order to distinguish from ordinary typing premises, side conditions are enclosed by a pair of braces, and written at right hand side of the corresponding typing rules. For instance, in the typing rule for $\mathbf{free}(E)$

$$\frac{\Gamma \vdash E : s^*}{\Gamma \vdash \mathbf{free}(E) : \mathbf{unit}} \{E \in \mathbf{effective}\} \quad (\text{TSFREE})$$

besides the constraint that the type of E should be s^* , E should not be **null** or dangling pointer, *i.e.*, **junk** either, that is, E must be a effective pointer. This requirement is given by the side condition $E \in \mathbf{effective}$.

1.3.5 Sharable Effect

The following rules define the sharable effect of *lvalue* expressions.

$$\boxed{\Gamma \vdash \ell \delta}$$

$$\frac{\Gamma(x) : \tau \delta \quad \delta = \mathbf{shared} \mid \mathbf{toshared}}{\Gamma \vdash x \delta} \quad (\text{TE-SVAR1})$$

$$\frac{\Gamma(x) : \tau \delta \quad \delta = \cdot \mid \mathbf{local}}{\Gamma \vdash x \mathbf{local}} \quad (\text{TE-SVAR2})$$

$$\frac{\Gamma \vdash \ell \delta \quad \delta = \mathbf{shared} \mid \mathbf{toshared}}{\Gamma \vdash \ell \rightarrow f \mathbf{shared}} \quad (\text{TE-SFIELD1})$$

$$\frac{\Gamma \vdash \ell \mathbf{local}}{\Gamma \vdash \ell \rightarrow f \mathbf{local}} \quad (\text{TE-SFIELD2})$$

1.3.6 Expressions

Typing rules for expressions are mostly straightforward.

$$\boxed{\Gamma \vdash E : \tau}$$

Constant

$$\frac{}{\vdash n : \mathbf{int}} \quad (\text{TEINT})$$

$$\frac{}{\vdash \mathbf{null} : \mathbf{ns}^*} \quad (\text{TENULL})$$

$$\frac{}{\vdash \mathbf{junk} : \tau} \quad (\text{TEJUNK})$$

$$\frac{\Gamma(s)(f_i) : \tau_i \quad \Gamma \vdash v_i : \tau_i \quad 1 \leq i \leq |F|}{\Gamma \vdash \langle f_i : v_i \rangle_{f_i \in F} : s} \quad (\text{TETUPLE})$$

Lval

$$\frac{\tau = \mathbf{int} \vee \tau = s^* \wedge \Gamma \vdash s^*}{\Gamma, x : \tau \vdash x : \tau} \quad (\text{TEVAR})$$

$$\frac{\Gamma \vdash \ell : s^* \quad \Gamma \vdash \Gamma(s)(f) : \tau}{\Gamma \vdash \ell \rightarrow f : \tau} \quad (\text{TEFIELD})$$

As to the field access $\ell \rightarrow f$ in the (TEFIELD) rule, the type system can further identify whether field f is a shape pointer or not by checking predicate $\text{isShapePtr}(\Gamma, s, f)$. If

$$\text{isShapePtr}(\Gamma, s, f)$$

holds, the field access would be marked with a special tag, *i.e.*, denoted as $\ell \rightarrow_s f$.

Binary Operators

$$\frac{\Gamma \vdash E_1 : \mathbf{int} \quad \Gamma \vdash E_2 : \mathbf{int}}{\Gamma \vdash E_1 \mathit{bop} E_2 : \mathbf{int}} \quad (\text{TEBOP})$$

$$\text{where } \mathit{bop} ::= + \mid - \mid * \mid / \mid \% \mid < \mid <= \mid == \mid != \mid >= \mid > \mid \&\& \mid \parallel$$

Unary Operators

$$\frac{\Gamma \vdash E : \mathbf{int}}{\Gamma \vdash \mathit{uop} E : \mathbf{int}} \quad (\text{TEUOP})$$

where $\mathit{uop} ::= - \mid !$

1.3.7 Statements

$$\boxed{\Gamma \vdash S : \mathbf{unit}}$$

$$\frac{\Gamma \vdash \ell : \mathbf{int} \quad \Gamma \vdash E : \mathbf{int}}{\Gamma \vdash [\ell := E]^l : \mathbf{unit}} \quad (\text{TSASS1})$$

The (TSASS1) rule does not describe the assignment between pointer type values, which are given by rules (TSASS2) and (TSASS3). The side condition $\neg \text{leak}(\ell)$ requires the assignment should not lead to memory leak.

$$\frac{\Gamma \vdash \ell : s^* \quad \Gamma \vdash \ell \delta \quad \Gamma \vdash \ell' : s^* \quad \Gamma \vdash \ell' \delta' \quad \delta = \delta' \vee \delta \neq \mathbf{local} \wedge \delta' \neq \mathbf{local}}{\Gamma \vdash [\ell := \ell']^l : \mathbf{unit}} \{ \neg \text{leak}(\ell) \} \quad (\text{TSASS2})$$

$$\frac{\Gamma \vdash \ell : s^*}{\Gamma \vdash [\ell := \mathbf{null}]^l : \mathbf{unit}} \{ \neg \text{leak}(\ell) \} \quad (\text{TSASS3})$$

The **malloc** statement allocates memory according to the size of its argument, and the argument should be one of the defined structure type name. If *lvalue* expression ℓ to which the result of **malloc** is assigned has **shared** effect, *i.e.*, $\Gamma \vdash \ell \mathbf{shared}$, then the **malloc** can be marked as **malloc_s**, representing that it would allocate a new memory cell from the shared heap. If ℓ has **local** effect, *i.e.*, $\Gamma \vdash \ell \mathbf{local}$, then the **malloc** can be marked as **malloc_p**, representing that it would allocate a new memory cell from the thread-local heap. Similarly the $[\mathbf{free}(\ell)]^l$ can also be marked as $[\mathbf{free}_s(\ell)]^l$ or $[\mathbf{free}_p(\ell)]^l$ by checking whether the sharable effect of ℓ is **shared** or **local**.

$$\frac{\Gamma \vdash \ell : s^* \quad \Gamma \vdash s}{\Gamma \vdash [\ell := \mathbf{malloc}(s)]^l : \mathbf{unit}} \{ \neg \text{leak}(\ell) \} \quad (\text{TSALLOC})$$

$$\frac{\Gamma \vdash \ell : s^*}{\Gamma \vdash [\mathbf{free}(\ell)]^l : \mathbf{unit}} \{ \ell \in \mathbf{effective} \} \quad (\text{TSFREE})$$

$$\frac{\Gamma \vdash x \mathbf{shared}}{[\mathbf{acqv}(x)]^l : \mathbf{unit}} \quad (\text{TSACQ-VAR})$$

$$\frac{\Gamma \vdash x \mathbf{shared}}{[\mathbf{relv}(x)]^l : \mathbf{unit}} \quad (\text{TSREL-VAR})$$

$$\frac{\Gamma \vdash \ell : \mathbf{s}^* \quad \Gamma \vdash \ell \mathbf{shared} \vee \Gamma \vdash \ell \mathbf{toshared}}{\Gamma \vdash [\mathbf{acq}(\ell)]^l : \mathbf{unit}} \quad \{\ell \in \mathbf{effective}\} \quad (\text{TSACQ-LOC})$$

$$\frac{\Gamma \vdash \ell : \mathbf{s}^* \quad \Gamma \vdash \ell \mathbf{shared} \vee \Gamma \vdash \ell \mathbf{toshared}}{\Gamma \vdash [\mathbf{rell}(\ell)]^l : \mathbf{unit}} \quad \{\ell \in \mathbf{effective}\} \quad (\text{TSREL-LOC})$$

$$\frac{\Gamma \vdash E : \mathbf{int} \quad \Gamma \vdash S_1 : \mathbf{unit} \quad \Gamma \vdash S_2 : \mathbf{unit}}{\Gamma \vdash \mathbf{if} E S_1 S_2 : \mathbf{unit}} \quad (\text{TSIF})$$

$$\frac{}{\Gamma \vdash \mathbf{skip} : \mathbf{unit}} \quad (\text{TSSKIP})$$

$$\frac{\Gamma \vdash E : \mathbf{int} \quad \Gamma \vdash S : \mathbf{unit}}{\Gamma \vdash \mathbf{while} E S : \mathbf{unit}} \quad (\text{TSWHILE})$$

$$\frac{\Gamma \vdash S_1 : \mathbf{unit} \quad \Gamma \vdash S_2 : \mathbf{unit}}{\Gamma \vdash S_1 ; S_2 : \mathbf{unit}} \quad (\text{TSSEQ})$$

The (TSPAR) rule requires that each sub-thread can access variables with **shared** tag and structure types appearing in the surrounding typing context Γ , that is, the typing context of sub-thread i consists of $\Gamma \setminus \mathbf{shared}$ obtained from the surrounding and its thread-local one Γ_i .

$$\frac{\Gamma' = \Gamma \setminus \mathbf{shared} \quad \Gamma_1 \vdash_T \diamond \quad \dots \quad \Gamma_n \vdash_T \diamond \quad \Gamma', \Gamma_1 \vdash S_1 : \mathbf{unit} \quad \dots \quad \Gamma', \Gamma_n \vdash S_n : \mathbf{unit}}{\Gamma \vdash \Gamma_1 S_1 \parallel \dots \parallel \Gamma_n S_n : \mathbf{unit}} \quad (\text{TSPAR})$$

1.3.8 Program

The (TPROG) rule says a SPC-II program $P = \Gamma_G S$ is well-formed if Γ_G is a well-formed global typing context, and the statement S is well-formed under the typing context Γ_G .

$$\boxed{\emptyset_\Gamma \vdash P : \mathbf{unit}}$$

$$\frac{P = \Gamma_G S \quad \Gamma_G \vdash_G \diamond \quad \Gamma_G \vdash S : \mathbf{unit}}{\emptyset_\Gamma \vdash P : \mathbf{unit}} \quad (\text{TPROG})$$