Variable Permissions for Concurrency Verification*

Duy-Khanh Le, Wei-Ngan Chin, Yong-Meng Teo

Department of Computer Science, National University of Singapore
(Technical Report)

Abstract. In the multicore era, verification for concurrent programs is increasingly important. Although state-of-the-art verification systems ensure safe concurrent accesses to heap data structures, they tend to ignore program variables. This is problematic since these variables might also be accessed by concurrent threads. One solution is to apply the same permission system, designed for heap memory, to variables. However, variables have different properties than heap memory and could benefit from a simpler reasoning scheme. In this paper, we propose a new permission system to ensure safe accesses to shared variables. Given a shared variable, a thread owns either a full permission or no permission at all. This ensures data-race freedom when accessing variables. Our goal is to soundly manage the transfer of variable permissions among threads. Moreover, we present an algorithm to automatically infer variable permissions from procedure specifications. Though we propose a simpler permission scheme, we show that our scheme is sufficiently expressive to capture programming models such as POSIX threads and Cilk. We also implement this new scheme inside a tool, called VPERM, to automatically verify the correctness of concurrent programs based on given pre/post-specifications.

1 Introduction

Access permissions have recently attracted much attention for reasoning about heap-manipulating concurrent programs [2, 4, 7, 9–12]. Each heap location is associated with a permission and a thread can access a location if and only if it has the access permission for that location. Permissions can be flexibly transferred among callers and callees of the same threads or among different threads. A thread needs a certain fraction of a permission to read a location but it has to own the full permission in order to perform a write. This guarantees data-race freedom in the presence of concurrent accesses to heap locations.

Program variables[1] can also be shared among threads and are prone to data races. Therefore, one may adopt a similar scheme, designed for heap locations, to reason about variables. "Variables as resource" [3, 21] indeed uses such a permission scheme for variables. Each variable x is augmented with a predicate

* This is an extended version of the paper published in the Proceedings of 14th International Conference on Formal Engineering Methods (ICFEM), pp. 5–21, Kyoto, Japan, Nov 12–16, 2012.

[1] We mean either global variables or local variables; as distinct from heap locations.

$Own(x, \pi)$ where π denotes the permission to access x. The permission domain is either (0,1] for fractional permissions [4] or [0,∞) for counting permissions [2]. This allows variables to be treated in the same way as heap locations. However, this permission scheme is more complex and places higher burden on programmers to figure out the fraction to be associated to a variable and how to perform permission accounting properly [2]. To the best of our knowledge, we are not aware of any existing verifiers that have fully implemented the idea. SMALLFOOT [1] uses side-conditions to outlaw conflicting accesses to variables. This, however, requires subtle, global, and hard-to-check conditions that a compiler should ensure [3, 22]. Similarly, CHALICE [15, 16], a program verifier developed for concurrency verification, does not support permissions for variables in method bodies. Even VERIFAST [12, 13], a state-of-the-art verifier, still does not naturally support concurrency reasoning using variables, though it has support for variables by simulating them as heap locations. Consequently, existing verification systems narrow the programmers' choice to heap locations instead of variables for shared accesses by concurrent threads at the expense of losing the expressivity and simplicity that variables provide.

In this paper, we argue that variables with their own characteristics could be treated in a much simpler way than heap locations. Firstly, each variable is distinct; therefore, aliasing issue required for heap locations can be ignored for variables in most cases. Secondly, if several threads need to concurrently *read* a variable, the main thread holding the full permission of the variable can just give each child thread a copy of the variable through pass-by-value mechanism. If concurrent threads require *write* access to the same variable, this shared variable can be protected by a mutex lock whose invariant holds the full permission of the variable. Lastly, if only one thread requires a write access to a given variable, we can simply pass the full permission of the variable into the thread (through pass-by-reference) whose permission is only returned when the child thread joins the main thread. This scheme allows concurrent but race-free accesses to variables.

Nonetheless, there are two scenarios where the above scheme is inadequate. The first scenario occurs in languages such as C/C++ when some variables can be aliased through the use of the address-of operator &. The second scenario occurs when concurrent threads require phased accesses to shared variables, e.g. concurrent threads safely read prior to writing to shared variables. In both scenarios, we propose to automatically translate the affected variables into pseudo-heap locations where a more complex heap permission scheme is utilized.

Because of the above observations, we propose to simply assign a permission of either full or zero to a variable. We can utilize heap (or pseudo-heap) locations to complement our concurrent programming model, *where necessary*, and also readily use variables, *where sufficient*. The net result is a rich but still verifiable programming paradigm for concurrent threads. We shall show that our treatment of variable permissions is sound and expressive to capture programming models such as POSIX threads [5] and Cilk [8]. To relieve programmers from annotation efforts, we shall demonstrate an algorithm to automatically infer variable permissions by only looking at procedure specifications. We shall also

provide a translation scheme to handle the variable aliasing (that can also be used for variables requiring phased accesses) and thus complement our treatment of variable permissions.

Contributions. In this paper, we make the following contributions:

- A simpler treatment of variable permissions to ensure safe concurrent accesses to program variables, as distinct from heap locations (Section 2 and 4.1). We also demonstrate the applicability of our scheme to popular programming models such as POSIX threads and Cilk (Section 5.1).
- An algorithm to automatically infer variable permissions from procedure specifications. This helps to reduce program annotations (Section 4.2).
- A translation scheme to eliminate variable aliasing for the purpose of program verification. (Section 4.3). We present how to translate programs with pointers and address-of operator (&) into our core language (Section 3).
- A prototype system, VPERM[2], to show that our variable permission scheme is practical to be implemented and to automatically verify concurrent programs such as parallel mergesort and parallel quicksort among others. Experimental results show that our system minimizes user annotations that are typically required in verification (Section 6).

2 Motivating Example

This section illustrates our treatment of variable permissions to reason about concurrent programs. Figure 1 shows an example illustrating the widely-used task-decomposition pattern in concurrent programming. The `main` procedure invokes the `creator` procedure to create a concurrent task and later performs a join to collect its result. In this example, the `main` procedure creates two local variables `x` and `y` and passes them to the `creator`. The `creator` forks a child thread that increases `x` by 1, and itself increases `y` by 2. The identifier `tid` of the child thread is returned to the `main` procedure which will later perform a `join`.

This example shows a fairly complicated inter-procedural passing of variables between the main thread and the child thread. It poses two challenges: (i) how to describe the fact that any accesses to `x` after forking the child thread and before joining it are unsafe, and (ii) how to propagate this fact across procedure boundaries. These issues can be resolved soundly and modularly by our proposed variable permissions.

Modular reasoning is achieved by augmenting the specification of the program with variable permissions: @*full*[...] and @*value*[...]. In pre-conditions (specified after `requires` keyword), @*full*[v^*] and @*value*[v^*] denote lists of pass-by-reference and pass-by-value parameters respectively. If a variable is passed by reference, the caller transfers the full permission of that variable to the callee. If a variable is passed by value, only a copy of that variable is passed to the callee and the caller still has the full permission of that variable. In post-conditions (after `ensures` keyword), @*full*[v^*] specifies the transfer of full permissions from

[2] The tool is available for both online use and download at `http://loris-7.ddns.comp.nus.edu.sg/~project/vperm/`.

the callee back to the caller via pass-by-reference parameters. Note that callers and callees can be in a single thread in case of normal procedure calls or in different threads in case of asynchronous calls via fork/join.

In this example, the `main` procedure transfers the full permissions of `x` and `y` to the `creator` (specified in its precondition as @*full*$[x, y]$). When forking a new child thread executing the `inc` procedure, the main thread transfers the full permission of `x` to the child thread (using pass-by-reference mechanism). This effect can be seen in the post-condition of the `creator` where we have two concurrent threads separated by the **and** keyword: after giving up the full permission of `x`, the main thread retains the full permission of `y` (@*full*$[y]$) while the child thread (with identifier **thread**=*tid*) holds the full permission of `x` (@*full*$[x]$). Thus, prior to invoking a `join` to merge back the child thread, the main thread has zero permission of `x` and is not allowed to access it (neither read nor write). This ensures data-race freedom since only one thread at a time can have the full permission of `x`.

```
void inc(ref int i, int j)
requires @full[i] ∧ @value[j]
ensures @full[i] ∧ i'=i+j;
{ i = i + j; }

int creator(ref int x, ref int y)
requires @full[x, y]
ensures @full[y] ∧ y'=y+2 ∧ res=tid
  and @full[x] ∧ x'=x+1 ∧ thread=tid;
{
  int tid = fork(inc, x, 1);
  inc(y, 2);
  return tid;
}

void main()
{
  int id;
  int x = 0, y = 0;
  id = creator(x, y);
  ...
  join(id);
  assert (x' + y' = 3);
}
```

Fig. 1. A Motivating Example

In the specification, we use the reserved keyword **thread** to capture the identifier `tid` of a child thread and the keyword **res** to represent the return value of a procedure call (in case of `creator`, the return value is the thread identifier `tid` of the child thread). Additionally, we use *primed notation* to handle updates to variables. The primed version x' of a variable x denotes its latest value; the unprimed version x denotes its initial value (i.e. its value at the beginning of the procedure). Note that a variable x and its primed version x' can be related but are two different logical variables.

One may think that this treatment of variable permissions can be easily captured through parameter passing, e.g. for each reference parameter `v`, just add an @*full*$[v]$ in the main thread of both pre- and post-conditions. However, this simple assumption may not hold in the context of concurrency. The key question is which thread holds full permission of a given variable. The full permission can belong to the main thread in the pre-condition but later it is transferred to a child thread in the post-condition and vice versa. For example, in the `creator`, the main thread has @*full*$[x]$ in the pre-condition but this permission is later

transferred to the child thread in the post-condition. In summary, the goal of our scheme is to succinctly manage the transfer of variable permissions among threads in a sound and modular manner.

3 Programming and Specification Languages

P ::=	$data\_decl^*$ $global\_decl^*$ $proc\_decl^*$	Program
$data\_decl$ ::=	**data** C { $field\_decl^*$ }	Data declaration
$field\_decl$ ::=	$type\ f$;	Field declaration
$global\_decl$ ::=	**global** $type\ v$	Global variable declaration
$proc\_decl$ ::=	$ret\_type\ pn(param^*)$ $spec^*$ { e }	Procedure declaration
$spec$ ::=	**requires** Φ_{pr} **ensures** Φ_{po};	Pre/Post-conditions
$param$ ::=	$type\ v$ | **ref** $type\ v$	Parameter
$type$ ::=	**int** | **bool** | C	Type
e ::=	v | $v.f$ | k	Variable/field/constant
$stmt$ ::=	$v = \textbf{fork}(pn,v^*)$ | **join**(v) | $pn(v^*)$ | ...	Statement

Fig. 2. Programming Language with Annotations and Concurrency

Our core programming language (Figure 2) is an imperative language with fork/join concurrency for dynamic thread creation. We chose fork/join as constructs for concurrency because they are often used in concurrent programming [17]. A program consists of a list of data declarations ($data\_decl^*$), a list of global variable declarations ($global\_decl^*$), and a list of procedure declarations ($proc\_decl^*$). Each procedure $proc\_decl$ is annotated with pairs of pre/post specifications (Φ_{pr}/Φ_{po}). A parameter $param$ can be passed by value or by reference (**ref**). A **fork** receives a procedure name pn, a list of parameters v^*, and returns a unique thread identifier as an integer. A **join** requires a thread identifier to join the thread back. The semantics of other program statements is standard as can be found in well-known languages such as C/C++. Note that the core language does not include program pointers and address-of operator (&). In Section 4.3, we show how to translate those constructs into the core language.

Shape predicate	$spred$	::=	$[\texttt{self}::]c[(\texttt{f})]\langle v^*\rangle \equiv \Phi\ [\textbf{inv}\ \pi_0]$
Separation formula	Φ	::=	$\bigvee(\exists v^* \cdot \mu[(\textbf{and}\ \mu)^*])^*$
Thread formula	μ	::=	$\kappa \wedge \nu \wedge \gamma \wedge \phi$
Heap formula	κ	::=	$\textbf{emp} \mid \ell \mid \kappa_1 * \kappa_2$
Atomic heap formula	ℓ	::=	$p::c[(\texttt{f})]\langle v^*\rangle$
Vperm formula	ν	::=	$@zero[v^*] \mid @full[v^*] \mid @value[v^*] \mid \nu_1 \wedge \nu_2 \mid \nu_1 \vee \nu_2$
Thread id formula	γ	::=	$\textbf{thread} = v \mid true$
Pure formula	ϕ	::=	...

Fractional permission variable $\texttt{f} \in (0,1]$ $\quad$ $v \in$ Variables

$c \in$ Data or predicate names $\quad$ $k \in$ Integer constants

Fig. 3. Grammar for Specification Language

Figure 3 shows our rich specification language for concurrent programs manipulating variables and heap locations. For variables, we use variable permissions. For heap locations, we support user-defined predicates *spred* [18] and fractional permissions f [4]. Φ is a separation logic formula [23] in disjunctive normal form. Each disjunct in Φ consists of a thread formula μ for a main thread and a list of thread formulas (separated by the **and** keyword) to represent concurrent threads. Each thread formula μ contains four parts: a heap formula κ, a vperm formula ν, a threading formula γ, and a pure formula ϕ . A *heap formula* κ consists of multiple atomic heap formulas ℓ connected with each other via separation connectives $*$. An atomic heap formula $p{::}c[(\mathtt{f})]\langle v^* \rangle$ represents the fact that a thread has certain permission `f` to access a heap location of type `c` pointed to by `p`. Vperm formula ν describes permissions of variables (Section 4.1). A thread id formula γ specifies the identifier of a concurrent thread using the keyword **thread**; a main thread has a thread id formula of `true`. A pure formula ϕ consists of standard equality/inequality, Presburger arithmetic and set constraints.

4 Variable Permissions for Safe Concurrency

4.1 Verification Rules

Our verification system is based on entailment checking:

$$\Delta_A \vdash \Delta_C \rightsquigarrow \Delta_R$$

Intuitively, the entailment checks if the antecedent Δ_A is precise enough to imply the consequent Δ_C, and computes the residue for the next program state Δ_R.

Formalism. In order to ensure safe concurrent accesses to variables, we use two key annotations for variable permissions:

- $@full[v^*]$ specifies the full permissions of a list of variables v^*. In pre-conditions, it means that v^* is a list of pass-by-reference parameters. In post-conditions, it captures the return of permissions to caller.
- $@value[v^*]$ only appears in pre-conditions to specify a list of pass-by-value parameters v^*.

$$@full[S] \ \wedge \ v{\not\in}S \vdash @full[v] \ \rightsquigarrow \ fail \qquad \textbf{FAIL-1}$$

$$@full[S] \ \wedge \ v{\not\in}S \vdash @value[v] \ \rightsquigarrow \ fail \qquad \textbf{FAIL-2}$$

$$\frac{v \in S}{@full[S] \vdash @full[v] \ \rightsquigarrow \ @full[S{-}\{v\}]} \qquad \textbf{P-REF}$$

$$\frac{v \in S}{@full[S] \vdash @value[v] \ \rightsquigarrow \ @full[S]} \qquad \textbf{P-VAL}$$

$$@full[S_1] \wedge @full[S_2] \ \rightsquigarrow \ @full[S_1 \cup S_2] \qquad \textbf{NORM-1}$$

$$@full[S_1] \vee @full[S_2] \ \rightsquigarrow \ @full[S_1 \cap S_2] \qquad \textbf{NORM-2}$$

$$@full[S_1] \wedge @value[S_2] \ \rightsquigarrow \ @full[S_1 \cup S_2] \qquad \textbf{BEGIN}$$

Fig. 4. Entailment Rules on Variable Permissions

Variable permissions can be transferred among callers and callees of the same thread, and among distinct threads. The verification rules for variable permissions are shown in Figure 4. A main thread (or a caller) that does not have full permission of a variable cannot pass that full permission to another thread (or a callee) either by reference or by value (**FAIL−1** and **FAIL−2**). After passing a variable by reference, a main thread (or a caller) loses the full permission of that variable (**P−REF**). However, for a pass-by-value variable, it will still retain the full permission (**P−VAL**). The normalization rules **NORM−1** and **NORM−2** soundly approximate sets of full permissions. At the beginning of a procedure, a main thread has full permissions of its pass-by-reference and pass-by-value parameters (**BEGIN**). The rules presented are simple, and this is precisely how we would like the readers to feel. Simplicity has its virtue and we hope that this would encourage safer concurrent programs to be written.

In our implementation, we also support $@zero[\cdots]$ as a dual to $@full[\cdots]$ annotation. The former denotes a set of variables that may possibly have zero permission. This is useful for more concise representation since only a small fraction of variables typically lose their permissions temporarily.

Forward Verification. Forward verification is formalized using Hoare's triples of the form $\{\Phi_{pr}\}P\{\Phi_{po}\}$: given a program P beginning in a state satisfying the pre-condition Φ_{pr}, if it terminates, it will do so in a state satisfying the post-condition Φ_{po}. Our forward verification rules are presented in Figure 5. We only focus on three key statements that transfer variable permissions: procedure call, fork and join. Note that the transfer of variable permissions is done via entailments as illustrated in Figure 4. In our system, each program state $\Delta[\Delta_t^*]$ consists of the current state Δ of a main thread and a list of post-states Δ_t^* of child threads. Here post-states refer to states of child threads after they finish execution. These post-states will be merged into the state of the main thread when child threads are joined.

$$\frac{\{P\}\ pn(v^*)\ \{Q\} \quad \Delta \vdash P \rightsquigarrow \Delta_1 \quad \Delta_2 \stackrel{\Delta}{=} \Delta_1 * Q}{\{\Delta[\Delta_t^*]\}\ pn(v^*)\ \{\Delta_2[\Delta_t^*]\}} \ \textbf{CALL}$$

$$\frac{\begin{array}{c}\{P\}\ pn(v^*)\ \{Q\} \qquad \Delta \vdash P \rightsquigarrow \Delta_1 \\ \Delta_{tnew} \stackrel{\Delta}{=} Q \wedge \textbf{thread}{=}unique\_id \\ \Delta_2 \stackrel{\Delta}{=} \Delta_1 \circ_{\{v\}} v'{=}unique\_id\end{array}}{\{\Delta[\Delta_t^*]\}\ v := \textbf{fork}(pn,v^*)\ \{\Delta_2[\Delta_{tnew}::\Delta_t^*]\}} \ \textbf{FORK}$$

$$\frac{\begin{array}{c}(\Delta_1 \wedge \textbf{thread}{=}id) \in \Delta_t^* \\ \Delta \vdash v' = id \rightsquigarrow \Delta_2 \qquad \Delta_3 \stackrel{\Delta}{=} \Delta_2 * \Delta_1 \\ \Delta_{tnew}^* = \Delta_t^* - [\Delta_1 \wedge \textbf{thread}{=}id]\end{array}}{\{\Delta[\Delta_t^*]\}\ \textbf{join}(v)\ \{\Delta_3[\Delta_{tnew}^*]\}} \ \textbf{JOIN}$$

Fig. 5. Forward Verification Rules for Concurrency

In order to perform a procedure call (**CALL**), a main thread should be in a state Δ that can entail the pre-condition P of the procedure pn. For clarity of

presentation, we omit the substitutions that link actual and formal parameters of the procedure prior to the entailment. After the entailment, the main thread subsumes the post-condition Q of the procedure with the residue Δ_1 to form a new state Δ_2. The list of concurrent threads Δ_t^* remains unchanged.

Similarly, in order to fork a new child thread (FORK), a main thread should be in a state Δ that can satisfy the pre-condition P of the forked procedure *pn*. Then a new thread Δ_{tnew} with a unique identifier carrying the post-condition Q of the corresponding forked procedure is created. The new thread is then added to the list of child threads. The main thread keeps the identifier of the child thread in its new state Δ_2 via the return value v of the **fork** call.

In the opposite way, when joining a child thread with an identifier v (JOIN), the main thread checks if v is a certain identifier in any child thread, merges the post-state of the child thread Δ_1 into its residue state Δ_2 to form a new state Δ_3, and removes the thread from the list of concurrent threads (denoted by the subtraction "$-$"). The rest of verification rules used in our system only operate on the state of the main thread and are standard as discussed in [18].

Theorem 1 (Soundness of Variable Permission Scheme) *Given a program with a set of procedures P^i and their corresponding pre/post-conditions (Φ^i_{pr}/Φ^i_{po}) enhanced with variable permissions, if our verification system derives a proof for every procedure P^i, i.e. $\{\Phi^i_{pr}\}\ P^i\ \{\Phi^i_{po}\}$ is valid, then the program is free from data races.*

Proof. By proving that the scheme maintains the invariant that the full permission of each variable belongs to at most one thread at any time. More details are given in Appendix A. □

4.2 Inferring Variable Permissions

In this section, we investigate inference for variable permissions. Approaches in permission inference for variables [22] and heap locations [7, 10] require entire program code and/or its specifications for their global analysis. The simplicity of our variable permission scheme offers opportunities for automatically and modularly inferring variable permissions by only looking at procedure specifications.

Our inference is based on following key observations. Firstly, local variables of a procedure cannot escape from their lexical scope; therefore, they are not allowed to appear in post-conditions. Secondly, scopes of pass-by-value parameters are only within their procedures; therefore, @*value*[...] only exists in pre-conditions and updates to these parameters need not be specified in post-conditions. Thirdly, for each procedure with its *R-complete* pre/post-conditions, updates to its reference parameters must be specified in its post-condition via *primed notations*. Lastly, because child threads carry the post-conditions of their corresponding forked procedures, their states include information about updates to variables that were passed by reference to their forked procedures.

Definition 1 (Primed Notations and R-complete Specifications) *Primed notations represent the latest values of program variables; unprimed notations denote either logical variables or initial values of program variables. A procedure specification is R-complete if all updates to its pass-by-reference parameters are specified in the pre/post conditions using primed notations.*

Algorithm 1 Inferring variable permissions from procedure specifications

Input: Φ_{pr}, Φ_{po}: pre/post-conditions of a procedure without variable permissions
Input: V_{ref}, V_{val} : sets of pass-by-reference and pass-by-value parameters
Output: Pre/post-conditions with inferred variable permissions

1: V_{post}:=V_{ref}
2: /*Infer @*full*[...] annotations for post-condition*/
3: **for** each thread Δ in Φ_{po} **do**
4: /*Set of free variables that are updated in Δ using primed notations*/
5: V_m:=$\{v : v \in FreeVars(\Delta) \wedge isPrimed(v)\}$
6: **if** ($V_m - V_{post}$) $\neq \phi$ **then** Error
7: **else**
8: Δ:=$\Delta \wedge$ @*full*$[V_m]$
9: V_{post}:=$V_{post} - V_m$
10: **end if**
11: **end for**
12: /*excluding reference parameters not updated in post-condition*/
13: V_{pre}:=$V_{ref} - V_{post}$
14: /*Infer @*full*[...] annotations for pre-condition's child threads*/
15: /*in the same way as with those in post-condition but replace V_{post} by V_{pre}*/
16: **for** each child thread Δ_t in Φ_{pr} **do**
17: ...
18: **end for**
19: For the main thread Δ in Φ_{pr}: $\Delta := \Delta \wedge$ @*full*$[V_{pre}] \wedge$ @*value*$[V_{val}]$
20: **return** Φ_{pr},Φ_{po}

We present our inference in Algorithm 1. For each procedure, the algorithm starts inference for the post-condition first. For each thread in the post-condition (either main thread or child thread), the full permissions are inferred by computing those pass-by-reference parameters that are updated in each thread's specification via primed notations. The `if` statement in line 6 detects an error if there are some primed variables that (1) are not reference parameters or (2) belonged to other threads in the previous iterations. The subtraction in line 9 removes from the set of reference parameters V_{post} those variables whose inferred full permissions already belonged to the current thread. This ensures that only one thread in the specification holds the full permission of a variable. Because child threads in the pre-condition carry the post-conditions of their corresponding forked procedures, we infer variable permissions for these child threads in the same way as with those in the post-condition. Note that the main thread is the currently active execution thread; therefore, its state in the pre-condition does not include primed variables. The main thread of the pre-condition holds full permissions of variables whose are updated (specified in the post-condition) and do not belong to any child threads. The subtraction in line 13 is necessary

because there are certain variables that are passed by reference but their full permissions do not belong to any threads (see Section 5.1 for more discussions). Finally, permission annotation @*value*[...] of pass-by-value parameters is added into the main thread of the pre-condition. For illustration, we present a running example in Table 1.

Table 1. Inferring variable permissions for procedure `creator` in Figure 1

Input		**Intermediate values**	**Inferred**
V_{ref}:={x, y}, V_{val}:={}			
Φ_{po}:=	y'=y+2 ∧ **res**=tid	V_{post}:={y}, V_m:={y}	@*full*[y]
	and x'=x+1 ∧ **thread**=tid;	V_{post}:={x, y}, V_m:={x}	@*full*[x]
Φ_{pr}:=	`true`	V_{pre}:={x, y}	@*full*[x, y]

Corollary 2 (Soundness of Inference and Verification) *Given a procedure* `P` *with its R-complete pre/post-conditions (Φ_{pr}/Φ_{po}) without variable permissions, and our inference algorithm results in new pre/post-conditions (Φ'_{pr}/Φ'_{po}) with inferred variable permissions, if our verification system derives a proof, i.e. {Φ'_{pr}} `P` {Φ'_{po}} is valid, then the procedure* `P` *is free from data races.*

Proof. We first prove that the inferred full permission of each variable belongs to at most one thread in a procedure's *R-complete* specification. Then we prove that with the inferred variable permissions, the procedure is free from data races. Details are given in Appendix B. □

4.3 Eliminating Variable Aliasing

In this section, we investigate the problem of variable aliasing. Aliasing occurs when a data location can be accessed through different symbolic names (i.e. variable names). For example in C/C++, variables can be aliased by the use of address-of operator (&). This poses challenges to program verification in general and concurrency verification in particular. Figure 6a shows a problematic example where `p` and `x` are aliased due to the assignment p=&x. After passing `x` by reference to a child thread, although the main thread does not have permission to access `x`, it can still access the value of `x` via its alias ∗`p` and therefore incurs possible data races. Our goal is to ensure safe concurrent accesses to variables even in the presence of aliasing, e.g. to outlaw racy accesses to the value of `x`.

We propose a translation scheme to eliminating variable aliasing by unifying pointers to program variables and pointers to heap locations. The translation is automatic and transparent to programmers. We refer to each variable (or parameter) whose &`x` appears in the program as an *addressable* variable. Intuitively, for each *addressable* variable, our translation scheme transforms it into a pointer to a pseudo-heap location by the following substitution ρ=[`int` ↦ `int_ptr`, &`x` ↦ `x`, `x` ↦ `x.val`]. Our approach covers values of any type (including primitive and data types). For each type `t`, there is a corresponding

```
void inc(ref int i, int j)
requires @full[i] ∧ @value[j]
ensures @full[i] ∧ i′=i+j;
{ i = i + j; }

void main()
{
  int x = 0;
  int * p = &x;
  int id = fork(inc, x, 1);
  ...//accesses to *p are racy
  join(id);

}
```

(a) Original Program

```
void inc(int_ptr i, int j)
requires i::int_ptr⟨old_i⟩ ∧ @value[i, j]
ensures i::int_ptr⟨new_i⟩ ∧ new_i=old_i + j;
{ i.val = i.val + j; }

void main()
{
  int_ptr x = new int_ptr(0);
  int_ptr p = x;
  int id = fork(inc, x, 1);
  ...//accesses to p.val or x.val are illegal
  join(id);
  delete(x);
}
```

(b) Translated Program

Fig. 6. An Example of Eliminating Variable Aliasing

type `t_ptr` to represent the type of pointers to pseudo-heap locations holding a value of type `t`. The value located at a pseudo-heap location is accessed via its `val` field (e.g. `x.val`).

Definition 2 (Pseudo-heap Locations) *Pseudo-heap locations are heap-allocated locations used for verification purpose only. Each pseudo-heap location represents a transformed program variable and captures the original value of the variable in its* `val` *field.*

Our scheme also translates program pointers into pointers to heap-allocated locations by the following substitution ρ=[`int*` $\mapsto$ `int_ptr`, `*p` $\mapsto$ `p.val`]. For pointers that point to another pointer, our translation is also applicable, e.g. `int**` is translated into `int_ptr_ptr`. The translation scheme ensures that the semantics of the translated program is equivalent to that of the original program. By transforming addressable variables into pseudo-heap locations, reasoning about aliased variables has been translated to reasoning about aliased heap locations which is easier to handle (i.e. using separation logic [23]). For detailed formal discussions, we refer interested readers to Appendix C.

An example translation is shown in Figure 6b. The addressable variable `x` of type `int` is transformed into a pointer to a pseudo-heap location of type `int_ptr`. The program pointer `p` becomes a pointer to the location which `x` refers to. Variable `x` will then be passed to a child thread. The procedure `inc` is also translated to reflect the fact that its reference parameter `i` has been transformed. In the specification, `i::int_ptr⟨old_i⟩` represents the fact that `i` is a variable of type `int_ptr` pointing to a pseudo-heap location containing certain value `old_i`. The original value of `x` is indeed captured in the value of the pseudo-heap location. In the translated program, when the main thread passes variable `x` to the child thread, the pseudo-heap location that `x` points to is also passed to the child thread. Therefore, before the child thread joins, the main thread cannot access the pseudo-heap location (e.g. via `p.val`) because it no longer owns that location. Note that the pseudo-heap location is deleted at the end to prevent memory leak.

We propose this translation for verification purpose *only* and do not recommend it for compilation use due to performance deficiency since accessing heap-allocated locations are typically more costly than program variables. Variable aliasing may also occur via parameter-passing when two reference parameters of a procedure refer to the same actual variable. Our variable permission scheme (as presented in Section 4.1) disallows the possibility because a caller cannot have two full permissions of a variable to pass it by reference twice.

5 Discussion

5.1 Applicability of the Proposed Variable Permissions

In this section, we discuss the application of our variable permission scheme to popular concurrent programming models such as POSIX threads and Cilk.

Pthreads is considered one of the most popular concurrent programming models for C/C++ [5]. In Pthreads, when creating a new child thread, a main thread passes a pointer to a heap location to the child thread. We model this argument passing by giving a copy of that pointer to the child thread. Furthermore, Pthreads uses global variables to facilitate sharing among threads. If several threads need to concurrently read a shared global variable, the main thread holding the full permission of that variable can just give each child thread a copy of that variable through pass-by-value mechanism. If concurrent threads require write access to the same variables, these variables can be protected by mutex locks whose invariants hold full permissions of the variables. This allows concurrent but race-free accesses to shared global variables. In our system, mutable global variables are automatically converted into pseudo reference parameters for each procedure (that uses them) prior to verification. For shared global variables that are protected by mutex locks, although they are converted into pseudo reference parameters, neither of concurrent threads has the variables' full permissions. It is the locks' invariants that capture the full permissions. Permission annotations for these variables are automatically inferred as shown in Section 4.2. Note that Pthreads' mutex locks are heap-allocated and therefore require reasoning over heap locations which is beyond the scope of this paper. We refer interested readers to [9, 11, 12] for detailed discussions.

Cilk is a well-known concurrent programming model originally developed at MIT and recently adopted by Intel [8]. In Cilk, the `spawn` keyword is used to create a new thread and to return the value of the procedure call instead of a thread identifier. Before the child thread ends, any accesses to that return value are unsafe. Our `fork` can have the same effect by passing an additional variable by reference to capture the return value. This guarantees data-race freedom because only the child thread has the full permission of that variable. More importantly, compared with Pthreads, Cilk provides more flexible parameter passing when creating a child thread. Multiple variables can be passed to a child thread either by value or by reference. This flexible passing can be naturally handled by our pass-by-value and pass-by-reference scheme. To the best of our

knowledge, our scheme is the first verification methodology for expressing such a flexible parameter passing style.

5.2 Phased Accesses to Shared Variables

Our variable permission is designed as a simpler permission scheme that can be used where sufficient. For immutable variables that are shared by concurrent threads, the general guideline is to pass copies of those variables to the threads to enjoy safe accesses to those copies. Mutable variables can be shared but should be protected by mutex locks to ensure race-freedom because there are some threads mutating the variables. However, there is still a class of complex sharing patterns that cannot be directly handled by our scheme. For example, a thread holds a certain permission to read a shared variable and is guaranteed that no other threads can modify the variable (read phase). Later, it acquires additional permissions from other threads and/or lock invariants, and combines them into a full permission to modify the shared variable (write phase). This kind of *phased accesses* to shared variables cannot be verified without splitting a full permission into smaller partial permissions. In this case, the thread can hold a partial permission while the rest of permissions belong to other threads and/or lock invariants.

Under this circumstance, we propose to detect those variables that are accessed in a phased way, and transform them into pseudo-heap locations where a more complex reasoning scheme is utilized [9, 11, 12]. The translation is done in a similar way as shown in Section 4.3. As a result, our general guideline is to readily use variables in most cases where the proposed variable permission scheme is sufficient, and to automatically and uniformly transform variables into pseudo-heap locations where necessary, i.e. in complex scenarios such as aliasing and phased accesses.

6 Experimental Results

We have integrated our variable permission scheme, inference algorithm, and translation scheme into a tool called VPERM for verifying concurrent programs (see Appendix D for detailed descriptions of these programs). Our variable permission scheme is best compared with approaches in [3, 21, 22] but implementations of these approaches are not available. Therefore, we compare our system with VERIFAST, a state-of-the-art verifier, in terms of annotation overhead ($\frac{LOAnn}{LOC}$) and verification time. Note that VERIFAST does not naturally support permissions for variables but simulates shared variables as heap locations. All experiments were done on a 3.20GHz Intel Core i7-960 processor with 16GB memory running Ubuntu Linux 10.04. Table 2 shows that although slower than VERIFAST, our system is more automatic in the sense that our system requires significantly less annotation overhead. The annotation overhead does not grow with more lines of code because we only require pre/post specifications at the procedure boundary. On average, we require less than three lines of annotation per procedure. This is important to reduce programmers' efforts for annotation. VERIFAST has higher annotation overhead because beside pre/post specifications, it requires additional annotations (such as which predicate to open/close

Table 2. Annotation Overhead and Verification Time (*Procs* is the number of procedures used in a program; *LOC* stands for "lines of code"; *LOAnn* stands for "lines of annotation"; Times are in seconds)

Program	Procs	LOC	VERIFAST			VPERM		
			LOAnn	**Overhead**	**Time**	**LOAnn**	**Overhead**	**Time**
`alt_threading`	3	17	23	135%	0.03	6	35%	0.18
`threads`	8	68	34	50%	0.04	18	26%	0.44
`tree_count`	1	20	We are not aware of corresponding programs in VERIFAST distribution. They could be coded but require much annotation			2	10%	0.31
`tree_search`	1	23				3	13%	1.17
`task_decompose`	3	19				6	32%	0.21
`fibonacci`	2	30				4	13%	0.29
`quicksort`	3	78				10	13%	1.60
`mergesort`	6	104				12	12%	1.48

or which lemma to apply) for each non-trivial command, such as field-access, fork and join. Although we attempted to write annotations for those programs that are not present in VERIFAST distribution, they are by no means trivial. In many cases, writing correct annotations is difficult and time-consuming. Therefore, we believe that our system shows a decent trade-off where it takes longer verification time (machine effort) but requires considerable less manual annotation (human effort).

7 Related Work

In 1970s, Owicki-Gries [20] came up with the very first tractable proof method for concurrent programs that prevents conflicting accesses to variables using side-conditions. However, these conditions are subtle and hard for compilers to check because it involves examining the entire program [3, 22]. Recently, concurrent separation logic (CSL) [19] has been proposed to nicely reason about heap-manipulating concurrent programs but CSL still relies on side-conditions for dealing with variables. SMALLFOOT verifier [1] uses CSL as its underlying logic and therefore suffers from the same limitation. In contrast, our scheme brings variable permissions into the logic and therefore makes it easier to check for conflicting accesses to variables. "Variables as resource" [3, 21] has proposed to apply permission systems [2, 4], originally designed for heap locations, to variables. Recently, Reddy et. al. [22] reformulate the treatment of variables using the system of syntactic control of interference. They share the same idea of applying fractional permissions [4] to variables. However, these more complex permission schemes place higher burden on programmers to figure out the permission fractions used to associate to variables. To the best of our knowledge, we are not aware of any existing verifiers that have fully implemented the idea. CHALICE [15, 16] ignores the treatment of variables in method bodies while VERIFAST [12, 13] simulates variables as heap locations. Although the underlying semantics of HOLFOOT [24] formalizes "variables as resource", its automatic verification system, which is based on SMALLFOOT, does not allow sharing variables using fractional permissions. In contrast, our variable permission scheme is simpler, using either full or zero permissions, but is expressive enough to support popular programming models such as Pthreads [5] and Cilk [8]. Furthermore,

while previous approaches assume theoretical programming languages without dynamic thread creation [3, 21, 24] and procedure [22], our variable permission scheme is more practical to be incorporated into VPERM tool and to verify concurrent programs with procedures and dynamic thread creation such as parallel quicksort and mergesort. We also presented an algorithm to automatically infer variable permissions and therefore reduce programmers' efforts for annotations. There is some work on automatic inference of access permissions in the literature [7, 10] but they only address permissions for heap locations. Reddy et. al. [22] is the very first work on inferring permissions for variables. However, their approach is different from ours. Firstly, while their approach is a two-pass algorithm over entire program syntax tree and proof outline, our approach can infer variable permissions directly from procedure specifications. Secondly, their work targets programs written in a theoretical language without procedures and dynamic thread creation while our approach supports more realistic programs with procedures and fork/join concurrency. Lastly, most work on verification has often disallowed variable aliasing by using side-conditions [19, 20] or via assertions [3, 9]. Therefore, our presented translation scheme to eliminate variable aliasing is orthogonal to their work since we provide a way to transform addressable variables into pointers to pseudo-heap locations, and thus enable reasoning about their behaviors in the same way as heap locations [9, 19]. In contrast to several informal translation tools [6, 14] which attempt to translate C/C++ programs with pointers into Java, we present a translation scheme with its formal semantics. Another difference is that while they focus on language translation, we aim towards facilitating program verification.

8 Conclusion

We have proposed a new permission system to ensure data-race freedom when accessing variables. Our scheme is simple but expressive to capture programming models such as POSIX threads and Cilk. Through a simple permission scheme for variables, we have extended formal reasoning to popular concurrent programming paradigms that rely on variables. We have provided an algorithm to automatically infer variable permissions and thus reduced program annotations. We have also shown a translation scheme to eliminate variable aliasing and to facilitate verification of programs with aliases on variables. Lastly, we have implemented our scheme into a tool, called VPERM, for verifying concurrent programs including parallel quicksort and parallel mergesort.

Acknowledgement. We thank the reviewers of ICFEM 2012 for insightful feedback. This work is supported by MOE Project 2009-T2-1-063.

References

1. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *Formal Methods for Components and Objects*, pages 115–137, 2005.
2. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission Accounting in Separation Logic. In *ACM Symposium on Principles of Programming Languages*, pages 259–270, New York, NY, USA, 2005. ACM.

3. R. Bornat, C. Calcagno, and H. Yang. Variables as Resource in Separation Logic. *Electronic Notes in Theoretical Computer Science*, 155:247–276, 2006.
4. J. Boyland. Checking Interference with Fractional Permissions. In *Proceedings of the International Static Analysis Symposium*, pages 55–72, 2003.
5. David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
6. Erik D. Demaine. C to Java: Converting Pointers into References. *Concurrency - Practice and Experience*, 10(11-13):851–861, 1998.
7. P. Ferrara and P. Müller. Automatic Inference of Access Permissions. In *Proceedings of the International on Verification, Model Checking, and Abstract Interpretation*. Springer, 2012.
8. M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 212–223, New York, NY, USA, 1998.
9. A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local Reasoning for Storable Locks and Threads. In *Asian Symposium on Programming Languages And Systems*, pages 19–37, 2007.
10. S. Heule, K. R. M. Leino, P. Müller, and A. J. Summers. Fractional Permissions Without the Fractions. In *Proceedings of the International Workshop on Formal Techniques for Java-like Programs*, July 2011.
11. A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle Semantics for Concurrent Separation Logic. In *European Symposium on Programming*, pages 353–367, Berlin, Heidelberg, 2008.
12. B. Jacobs and F. Piessens. Expressive modular fine-grained concurrency specification. In *ACM Symposium on Principles of Programming Languages*, pages 271–282, New York, NY, USA, 2011.
13. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: a Powerful, Sound, Predictable, Fast Verifier for C and Java. In *Proceedings of the Third International Conference on NASA Formal Methods*, pages 41–55, Berlin, Heidelberg, 2011.
14. Chris Laffra. A C++ to Java Translator. In *Advanced Java: Idioms, Pitfalls, Styles and Programming Tips*, chapter 4. Prentice Hall Computer Books, 1996.
15. K. R. Leino and P. Müller. A Basis for Verifying Multi-threaded Programs. In *European Symposium on Programming*, pages 378–393, Berlin, Heidelberg, 2009.
16. K. R. Leino, P. Müller, and J. Smans. Verification of Concurrent Programs with Chalice. In *Foundations of Security Analysis and Design V*, pages 195–222. Springer-Verlag, Berlin, Heidelberg, 2009.
17. Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, first edition, 2004.
18. H.H. Nguyen, C. David, S.C. Qin, and W.N. Chin. Automated Verification of Shape and Size Properties via Separation Logic. In *Proceedings of the International on Verification, Model Checking, and Abstract Interpretation*, Nice, France, 2007.
19. P. W. O'Hearn. Resources, Concurrency and Local Reasoning. In *International Conference on Concurrency Theory*, pages 49–67, 2004.
20. S. Owicki and D. Gries. Verifying Properties of Parallel Programs: an Axiomatic Approach. *Communications of the ACM*, pages 279–285, 1976.
21. M. Parkinson, R. Bornat, and C. Calcagno. Variables as Resource in Hoare Logics. In *IEEE Logic In Computer Science*, pages 137–146, Washington, DC, USA, 2006.
22. U. S. Reddy and J. C. Reynolds. Syntactic Control of Interference for Separation Logic. In *ACM Symposium on Principles of Programming Languages*, pages 323–336, New York, NY, USA, 2012.

23. J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *IEEE Logic In Computer Science*, Copenhagen, Denmark, July 2002.
24. Thomas Tuerk. *A Separation Logic Framework for HOL*. PhD thesis, University of Cambridge, 2011.

A Soundness of Variable Permission Scheme

We sketch how our variable permission scheme (Section 4.1) ensures safe concurrency (data-race freedom). We prove that our scheme maintains the invariant that the full permission of each variable belongs to at most one thread at any time.

Definition 3 (Data-race Freedom) *A program is data-race free if there does not exist two concurrent threads Δ_{t1} and Δ_{t2}, and a variable* `x` *such that $\Delta_{t1} \vdash @full[x]$ and $\Delta_{t2} \vdash @full[x]$ at the same time.*

Definition 4 (Permission Invariant) *For every variable* `x`*, its full permission belongs to at most one thread at any time.*

Theorem 3 (Non-duplicable Permissions) *For every variable* `x`*, its full permission cannot be duplicated.*

Proof. By induction on entailment rules in Figure 4. □

We prove the soundness of our variable permission scheme by contradiction.

Proof.
Hypothesis: There are data races, i.e. there are two threads that have full permission of the same variable `x` at the same time.

The two threads can be: a main thread and a child thread (Case 1), or both child threads (Case 2).

Case 1: A main thread and a child thread have the full permission of the same variable.

Case 1.1: The child thread obtains the full permission after being forked by the main thread. Therefore, the variable `x` has to be passed by reference to the child thread (**P−REF** rule in Figure 4). Afterwards, the main thread loses the full permission because the permission is non-duplicable. This contradicts to the hypothesis.

Case 1.2: The child thread obtains the full permission from the lock invariant after acquiring a mutex lock. In our scheme, if a variable is protected by a mutex lock, the lock's invariant holds the full permission of the variable. Therefore, if the main thread has the full permission for `x`, it also has to acquire the full permission from the lock invariant. This leads to contradiction because two threads are not allowed to successfully acquire a lock at the same time.

Case 2: Two child threads have the full permission of the same variable.

Case 2.1: Child threads obtains the full permissions after being forked by another main thread. This is impossible because a full permission is non-duplicable.

Case 2.2: Child threads obtains the full permission from the lock invariant after acquiring a mutex lock. This is impossible because two threads are not allowed to successfully acquire a lock at the same time. □

B Soundness of Inference Algorithm

In this section, we give the soundness sketch of our inference algorithm (Section 4.2). We first prove that the inferred full permission of each variable belongs to at most one thread in a procedure's *R-complete* specification. Then we prove that with the inferred variable permissions, the procedure is free from data races.

Theorem 4 (Precise Inference) *The inferred full permission of each parameter belongs to at most one thread in a procedure's R-complete specification.*

Proof. We prove by contradiction.
HYPOTHESIS: There exists a parameter `x` whose inferred full permission belongs to more than one thread in the procedure's pre/post-conditions.
CASE 1: The parameter `x` is passed by reference.

CASE 1.1: The parameter `x` is not protected by any mutex lock. Because the specification is *R-complete*, by Definition 1, updates to `x` are specified in the specification using *primed notation*.

CASE 1.1.1: Inferring the permission of `x` in the post-condition.

Without lost of generosity, assuming that the full permission of `x` belongs to two threads in the post-condition, i.e. $@full[x]$ is in the state of the two threads. Because the algorithm iterates over each thread in a sequential manner (line 3-11), assuming that the two threads are visited in iterations `i` and `j` respectively ($i<j$). Let ${V_{post}}^i$ and ${V_m}^i$ denote the value of V_{post} and V_m after `i`-th iteration. Therefore, we have $x \in {V_m}^i$ and $x \in {V_m}^j$ with $i<j$. As a consequence, we have $x \in {V_{post}}^{j-1}$ (because $V_m - V_{post} = \phi$). By induction on the value of `j`, we have $x \in {V_{post}}^i$. This is impossible because of the subtraction in line 9.

CASE 1.1.2: Inferring the permission of `x` in the pre-condition.

Similar to CASE 1.1.1 but replace V_{post} by V_{pre}.

CASE 1.2: The parameter `x` is protected by some mutex lock.

In our scheme, if a variable `x` is protected by a mutex lock, only the lock's invariant holds the full permissions of `x`. This contradicts to the hypothesis. Note that in this case, updates to variable `x` are captured in the lock invariant. Therefore, neither threads hold the full permission of `x`. Formally, for every iteration `i`, $x \notin {V_m}^i$.

CASE 2: The parameter `x` is passed by value.

Because the main thread is the main execution thread, the permission $@value[...]$ of pass-by-value parameters is trivially added to the main thread of the precondition only (line 20). This contradicts to the hypothesis. Note that $@value[...]$ does not exist in the post-condition and updates to pass-by-value parameters are not allowed to be specified in the post-condition (to prevent them from escaping from their lexical scope). □

Corollary 5 (Soundness) *With the inferred variable permissions, the procedure is free from data races.*

Proof. This follows from the preciseness of our inference algorithm (Theorem 4) and the soundness of our underlying permission scheme (Theorem 1). □

C Translation Rules

Our translation rules are presented in Figure 7. As a part of the translation, we first transform the program to ensure that variables are of distinct names. Afterwards, we analyze the program to identify a set `V` of addressable variables that are passed by reference. Our translation starts with such a set of variables and gradually adds more addressable variables in. We use the notation $V \models e1 \hookrightarrow e2$ to indicate that given the aforementioned set `V`, the translation rules transform a program code `e1` with pointers and & operators into a new program `e2` expressible in our core language (Section 3). Most of the rules are straightforward. The most difficult part is to translate addressable variables that are passed by reference. Because scopes of reference parameters are beyond their procedures, we have to ensure that all instances of these variables are transformed into pseudo-heap locations. This is to ensure that any possible effects on the original variables can be entirely captured in the pseudo-heap locations.

$$\frac{not(isProcCall(e_1)) \quad v \in FV(e_1) \cap V \quad \rho{=}[\&v \mapsto v, v \mapsto v.val] \quad e_1'{=}\rho\ e_1 \quad V \models \{\ e_2\} \hookrightarrow \{e_2'\}}{V \models \{e_1;\ e_2\} \hookrightarrow \{e_1'; e_2'\}}\ [\textsc{trans-exp}]$$

$$\frac{\rho{=}[{}^*p \mapsto p.val] \quad e_1{=}\rho\ e}{V \models \{t^*\ p;\ e\} \hookrightarrow \{t\_ptr\ p; e_1\}}\ [\textsc{trans-pointer}]$$

$$\frac{(\&v \in e \vee v \in V) \quad V_1{=}V \cup \{v\} \quad V_1 \models e \hookrightarrow e_1}{V \models \{t\ v;\ e\} \hookrightarrow \{t\_ptr\ v = new\ t\_ptr(0); e_1; delete(v)\}}\ [\textsc{trans-var-decl}]$$

$$\frac{\&v \in e \quad p\ fresh \quad \rho{=}[v \mapsto p] \quad e_1{=}\rho\ e \quad V_1{=}\rho\ V \quad V_2{=}V_1 \cup \{p\} \quad V_2 \models e_1 \hookrightarrow e_2}{V \models t\ pn(t\ v, ...)\{e\} \hookrightarrow t\ pn(t\ v, ...)\{t\_ptr\ p = new\ t\_ptr(v); e_2; delete(p)\}}\ [\textsc{trans-param-val}]$$

$$\frac{v \in V \quad V \models e \hookrightarrow e_1 \quad (\Phi_{pr}', \Phi_{po}'){=}transSpec(v : t, \Phi_{pr}, \Phi_{po})}{\begin{array}{c}V \models t\ pn(\textbf{ref}\ t\ v, ...)\ \textbf{requires}\ \Phi_{pr}\ \textbf{ensures}\ \Phi_{po}\{e\} \\ \hookrightarrow t\ pn(t\_ptr\ v, ...)\ \textbf{requires}\ \Phi_{pr}'\ \textbf{ensures}\ \Phi_{po}'\{e_1\}\end{array}}\ [\textsc{trans-param-ref}]$$

$$\frac{\begin{array}{c}fresh\ old\_v, new\_v \quad \rho{=}[v \mapsto old\_v, v' \mapsto new\_v] \\ \Phi_{pr1}{=}\rho\ \Phi_{pr} \quad \Phi_{pr}'{=}v{::}\texttt{t\_ptr}\langle old\_v\rangle * \Phi_{pr1} \\ \Phi_{po1}{=}\rho\ \Phi_{po} \quad \Phi_{po}'{=}v{::}\texttt{t\_ptr}\langle new\_v\rangle * \Phi_{po}\end{array}}{transSpec(v : t, \Phi_{pr}, \Phi_{po}){:=}(\Phi_{pr}', \Phi_{po}')}\ [\textsc{trans-spec}]$$

$$\frac{\begin{array}{c}V \models t\ pn(..., t\ v, ..., \textbf{ref}\ \textbf{t}\ \textbf{u}, ...)\ \textbf{requires}\ \Phi_{pr}\ \textbf{ensures}\ \Phi_{po}\{e\} \hookrightarrow \\ t\ pn(..., t\ v, ..., \textbf{ref}\ \textbf{t}\ \textbf{u}, ...)\ \textbf{requires}\ \Phi_{pr}'\ \textbf{ensures}\ \Phi_{po}'\{e_1\} \\ v \in V \quad \rho{=}[\&v \mapsto v, v \mapsto v.val] \quad v'{=}\rho\ v\end{array}}{V \models pn(..., v, ..., u, ...) \hookrightarrow pn(..., v', ..., u....)}\ [\textsc{trans-call}]$$

Fig. 7. Translation Rules for Eliminating Variable Aliasing

D Experimental Programs

In this section, we briefly describe concurrent programs used in our experiments (Section 6). More examples can be found on our website[3]. These programs are challenging because of safety of variables and their full functional correctness. As mentioned in Section 3, our system supports verification of both variables and heap-manipulating data structures such as linked lists and trees. Proof obligations generated by our program verifier will be discharged by an entailment checker. Using various external provers such as Z3, Omega and Mona, our entailment checker is capable of verifying variety of constraints ranging from variable permission constraints, separation logic constraints, numerical constraints, first-order logic constraints to set constraints.

alt_threading. This program demonstrates the passing of variable permissions between a parent thread and a child thread. VERIFAST, in contrast, converts variables into heap locations before passing them.
threads. In this program, threads concurrently manipulate different parts of a tree and return the sum of factorials of all nodes in the tree. VERIFAST stores the sum in heap memory while we can naturally capture that sum in a variable.
tree_count. This program shows how to count the number of nodes in a tree in parallel. Concurrent threads update the corresponding counts into variables. Variable permissions, therefore, are used to prevent possible data races among threads.
tree_search. In this program, threads concurrently search for a node in a tree in a divide-and-conquer manner. Variables are used to keep different parts of the tree. We, therefore, ensure safety of variables using variable permissions. Besides, we also have to keep track of elements of the tree in a set and use Mona prover to discharge set constraints.
task_decompose. This is the motivating example mention in Section 2. It shows a fairly complicated inter-procedural passing of variables among concurrent threads.
fibonacci. This program shows a parallel implementation of Fibonacci program.The `para_fib` procedure creates two child threads to compute $(n-1)^{th}$ and $(n-2)^{th}$ Fibonacci numbers in parallel. In order to optimize performance, under a certain threshold ($n < 10$), the sequential algorithm (`seq_fib`) is used. We use reference parameters to capture the return values of concurrent threads. Therefore, the proposed variable permission scheme is used to prevent possible data races on these reference parameters.
mergesort, quicksort. In these programs, threads sort different parts of a linked list in parallel. Program variables are used to keep different parts of the list. Therefore, we use variable permissions to ensure safe accesses to these variables and at the same time have to maintain sorted-ness properties of the linked list. Interestingly, even for these challenging parallel programs, our system requires the same pre/post-conditions at procedure boundary as their sequential counterparts.

[3] `http://loris-7.ddns.comp.nus.edu.sg/~project/vperm/`