Verifying Optimizers for Concurrent Programs on Promising semantics

ANONYMOUS

ACM Reference Format:
Anonymous. 2018. Verifying Optimizers for Concurrent Programs on Promising semantics. *Proc. ACM Meas. Anal. Comput. Syst.* 37, 4, Article 111 (August 2018), 102 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Code optimizers are important components in compilations. Correct code optimizer requires that the target program generated preserve the semantics of the source program. Proving the correctness of code optimizers in compiler is usually difficult than proving the translation pass, which translates the program implemented in one language to another one, since the memory accesses in the source program may be modified during optimizations. In this document, we discuss the correctness proof of the optimizers for the concurrent programs under *promising semantics* [11] as shown in Fig. 1. At present, there are some works of proving the correctness of

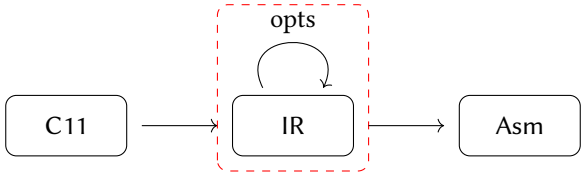


Fig. 1. What do we focus on in this work

code optimization algorithms in compilers for sequential programs [6, 17, 19], but there is not much discussion about how to prove the correctness of optimizers for concurrent programs under weak memory models.

- Jiang, et al. [7] develop CASCompCert for correct compilation of the data-race-free concurrent program. However, the concurrent program that they focused on is defined under SC memory model [10]. The behaviors of some atomic memory accesses defined in C11 [1] can not be depicted under SC memory model. For example, we can not define the behaviors of the atomic release write and the atomic acquire read on SC memory model. The following program behavior that is well-defined under C11 can not be generated under SC memory model.

$$\begin{array}{l} x_{rel} := 1; \\ r_1 := y_{acq}; \quad //0 \end{array} \;\Big\|\; \begin{array}{l} y_{rel} := 1; \\ r_1 := x_{acq}; \quad //0 \end{array}$$

Author's address: Anonymous.

2476-1249/2018/8-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

Moreover, the simulation in CASCompCert preserves the data race freedom in source programs. Thus, the simulation in CASCompCert can not prove the some optimizers, such as *loop invariant code motion* in LLVM, which may introduce read-write race during code optimization.

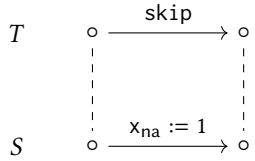
- Ševčík, et al. [15] develops CompCertTSO. However, the TSO memory model is still a strong memory model and the source programs defined on TSO memory model can not be compiled to efficient ARM or Power programs. Moreover, CompCertTSO relies on a strong simulation relation, which requires that the source and the target always generate the same memory accesses. Such restriction is too strong and many common optimizations, like eliminating redundant reads/writes and instruction reordering, can break it. For example, the following constant propagation optimization that eliminates redundant memory accesses can not be proved by the simulation relation in CompCertTSO.

$$\begin{array}{l} \mathtt{x_{na}} := 2; \\ r := \mathtt{x_{na}}; \end{array} \quad \rightsquigarrow \quad \begin{array}{l} \mathtt{x_{na}} := 2; \\ r := 2; \end{array}$$

  The constant propagation in CompCertTSO only optimizes the operations on registers.
- Most of the weak memory models for platform-independent concurrent programming language are axiomatic models [1, 3, 9]. They express the concurrency semantics in terms of global properties of complete execution. It makes them difficult to be used in the correctness proof of compilers, if we want to achieve modular reasoning.
- Promising semantics [5, 8, 11] is an operational model. The work of promising semantics defines a simulation relation to validate many code transformations for adjacent instructions. Their simulation is simple and may be applied to prove the correctness of some optimization algorithms. However, the relation between the target and source memory in their simulation is fixed. It is often too strong and causes that it is difficult to apply it to prove the correctness of the some code optimization algorithms. For example, their simulation relation restricts that each message in the target memory has a corresponding message with the same timestamp in the memory of the source program. We show that maintaining such restriction is not a trivial task in proving the correctness of some optimization algorithms. in the following. The below is an example of dead code elimination optimization.

$$\begin{array}{l} \texttt{while}(r_1 < r_2)\ \{ \\ \quad \mathtt{x_{na}} := 1; \\ \quad r_1 := r_1 + 1; \\ \} \\ \mathtt{x_{na}} := 2; \end{array} \quad \rightsquigarrow \quad \begin{array}{l} \texttt{while}(r_1 < r_2)\ \{ \\ \quad \texttt{skip}; \\ \quad r_1 := r_1 + 1; \\ \} \\ \mathtt{x_{na}} := 2; \end{array}$$

  In order to maintain such restriction that the timestamp of the message generated by the instruction "$\mathtt{x_{na}} := 2$" in the target program equals to the timestamp of the message generated by the instruction "$\mathtt{x_{na}} := 2$" in the source program, we can not establish the simulation relation between the target and source programs as the following form, since the source program will generate some messages about "$\mathtt{x_{na}} := 1$" before executing "$\mathtt{x_{na}} := 2$" and cause the timestamp of the message generated by "$\mathtt{x_{na}} := 2$" in the target program greater than the message generated by $\mathtt{x_{na}} := 2$ in the source program.



- Vafeiadis, et al. [18] discuss the validation of many standard compiler optimizations on C/C++11 memory model. Soham Chakraborty and Viktor Vafeiadis [3] validates compiler optimizations on LLVM memory model. Ševčík [4] assumes that the transformed program runs sequentially consistently and present the correctness of compiler optimizations under the data race freedom assumption. However, these work focus on whether standard compiler optimizations, such as eliminating redundant reads/write and instruction reordering, are valid under the specific memory models. They do not discuss how to use their conclusions to prove concrete code optimization algorithms. Their works are mainly used to validate, for a source program, whether the optimizer does a correct optimization on it, *e.g.,* [2, 12].
- There are many works [9, 11, 13, 14] that discuss and prove the correctness of compilation from concurrent programming languages with weak memory consistency semantics, such as promising semantics and C11 memory model, to mainstream multi-core architectures, such as x86TSO, ARM and POWER, by standard compilation schemes. However, the standard compilation schemes just map the high-level primitives to instructions of major modern architectures directly and do not include any code optimizations.

In this work, we consider how to prove the correctness of the optimizers about concurrent programs under *promising semantics*, since it is an operational memory model, whose definition does not rely on complete executions of programs, and has been proved to validate many code transformations for adjacent instructions, as well as standard compilation schemes to x86-TSO, ARM and Power. We focus on proving the the correctness of the optimization passes for two reasons.

(1) A compile usually has optimization pass and translation pass. Optimizations pass can modify the memory accesses in the source program during optimization. However, translation passes plays a role to transform a program in one language to a form in another language. It usually does not modify the memory accesses in the source program. Thus, from the perspective of memory accesses in the program compiled, the translation pass, which does identity optimizations on memory accesses in the source program, can be regarded as a special case of the optimization pass.
(2) Correctness translation from the programs in promising semantics to major modern architectures has been discussed in many previous work, *e.g.,* [14].

In this document, we will do the following contributions.

(1) We find that promising semantics can be converted to a non-preemptive semantics, which does not permit the thread switching and the promise step of the current thread after the execution of non-atomic steps as the following shown. Here, "**let** π **in** $\mathtt{f}_1 \parallel \mathtt{f}_2$" means two threads, one starting from the entry $\mathtt{f}_1$ in the code π and the other starting from the entry $\mathtt{f}_2$ in the code π, executing under promising semantics, and "**let** π **in** $\mathtt{f}_1 \mid \mathtt{f}_2$" means that such two threads execute under our non-preemptive semantics.

$$\textbf{let } \pi \textbf{ in } \mathtt{f}_1 \parallel \mathtt{f}_2 \;\approx\; \textbf{let } \pi \textbf{ in } \mathtt{f}_1 \mid \mathtt{f}_2$$

Proving the correctness of the code optimizers under the non-preemptive semantics can provide some convenience for us, since the non-preemptive semantics is simpler than promising semantics and there is no interaction with the environment after the execution non-atomic accesses in such non-preemptive semantics. In this work, we only consider the code optimizations on non-atomic memory accesses. GCC does nothing optimizations on the atomic memory accesses. As for LLVM, we only find that it does optimizations on the atomic memory accesses in *register promotion*. However, register promotion only optimizes the accesses on the memory locations, which are thread-local.
(2) We define a thread-local simulation relation under the non-preemptive semantics to prove the correctness of the compiler optimizations that we care about. Our thread-local simulation has the following advantages:
   - It is defined under the non-preemptive semantics, which is simpler than promising semantics. This can simplify the definition of our thread-local simulation and the correctness proof of some code optimizations,

such as instruction reordering, since we only need to consider the interaction between the current thread and the environment at specific program points.
- The invariant I for shared memory in our thread-local simulation is general and can be instantiated in proving specific code optimization algorithms. Thus, in proving optimizers that do not eliminate write operations, like common subexpression elimination and instruction reordering, the invariant for shared memory can be simple. The memory in target program and source program can be strictly equal and it can simplify our proof.
- Our thread-local simulation is *parallel compositional* as the following shown, if the source program is *write-write race free* on non-atomic memory accesses (shown as ww-RF($S_1 \mid S_2$)).

$$(I \vdash \pi_t(\mathtt{f}_1) \preccurlyeq \pi_s(\mathtt{f}_1) \land I \vdash \pi_t(\mathtt{f}_2) \preccurlyeq \pi_s(\mathtt{f}_2) \land \mathsf{ww\text{-}RF}(\mathbf{let}\ \pi_s\ \mathbf{in}\ \mathtt{f}_1 \mid \mathtt{f}_2)) \\ \Longrightarrow\ \mathbf{let}\ \pi_t\ \mathbf{in}\ \mathtt{f}_1 \mid \mathtt{f}_2 \preccurlyeq \mathbf{let}\ \pi_s\ \mathbf{in}\ \mathtt{f}_1 \mid \mathtt{f}_2$$

  Our work does not consider the multi-language linking.
- We show that our thread-local simulation is able to prove the correctness of common code optimization algorithms for eliminating redundant reads/writes and instruction reordering.

(3) We formulate the correctness of the optimizers for concurrent programs defined under promising semantics. We need to require the source programs does not contain *write-write race* on non-atomic memory accesses (defined by plain memory accesses in promising semantics) in formulating the correctness of the optimizers as the following form.

$$\forall \pi_t, \pi_s.\ (\mathsf{Optimizer}(\pi_s) = \pi_t \land \mathsf{ww\text{-}RF}(\mathbf{let}\ \pi_s\ \mathbf{in}\ \mathtt{f}_1 \parallel \mathtt{f}_2)) \\ \Longrightarrow\ \mathbf{let}\ \pi_t\ \mathbf{in}\ \mathtt{f}_1 \parallel \mathtt{f}_2 \subseteq \mathbf{let}\ \pi_s\ \mathbf{in}\ \mathtt{f}_1 \parallel \mathtt{f}_2$$

And we provide a verification framework, which we will give more introductions to in the following, in Fig. 2 to present how to establish the correctness of the optimizers. The reason that we permit that source programs have read-write data race, is that some optimizers, such as loop invariant code motion in LLVM, may introduce read-write data race on non-atomic memory accesses in optimization. Our thread-local simulation is able to preserve the write-write race freedom to make the correctness of optimizer *transitive*.

(4) We use our method to prove three common algorithms of compiler optimizations in our work. (1) *common subexpression elimination*, which is responsible for eliminating redundant reads in programs; (2) *dead code elimination*, a code optimization to eliminate redundant writes in programs; (3) *loop invariant code motion*, which recognizes computations in loops that produce the same value on every iteration of the loop an moves them out of the loop. The loop invariant code motion, like the algorithm in LLVM, may introduce read-write data race on non-atomic memory accesses in optimization. Thus, this explains why our work allows the source program to exit read write data race. The algorithms of common subexpression elimination and dead code elimination in our work are implemented based on the implementations of common subexpression elimination and dead code elimination in CompCert [6]. And the algorithm of loop invariant code motion is implemented based on the algorithm in Steven S.Muchnick's textbook on code optimizations [16]. However, we extend these code optimization algorithms in our work, since we consider the code optimizations accross atomic memory accesses and need to handle the atomic memory access operations in the program. Below, we give some examples to show that the code optimizations can across the atomic memory accesses. Some of those optimizations have already been observed in the LLVM.

$$\begin{array}{l} \mathtt{x}_{\mathsf{na}} := 4; \\ r := \mathtt{y}_{\mathsf{acq}}; \\ \mathtt{x}_{\mathsf{na}} := 2; \end{array} \overset{DCE}{\rightsquigarrow} \begin{array}{l} \mathtt{skip}; \\ r := \mathtt{y}_{\mathsf{acq}}; \\ \mathtt{x}_{\mathsf{na}} := 2; \end{array}$$

(Not observed in LLVM *)*

$$\begin{array}{l} r_1 := \mathtt{x}_{\mathsf{na}}; \\ \mathtt{y}_{\mathsf{rel}} := 1; \\ r_2 := \mathtt{x}_{\mathsf{na}}; \end{array} \overset{CSE}{\rightsquigarrow} \begin{array}{l} r_1 := \mathtt{x}_{\mathsf{na}}; \\ \mathtt{y}_{\mathsf{rel}} := 1; \\ r_2 := r_1; \end{array}$$

(Not observed in LLVM *)*

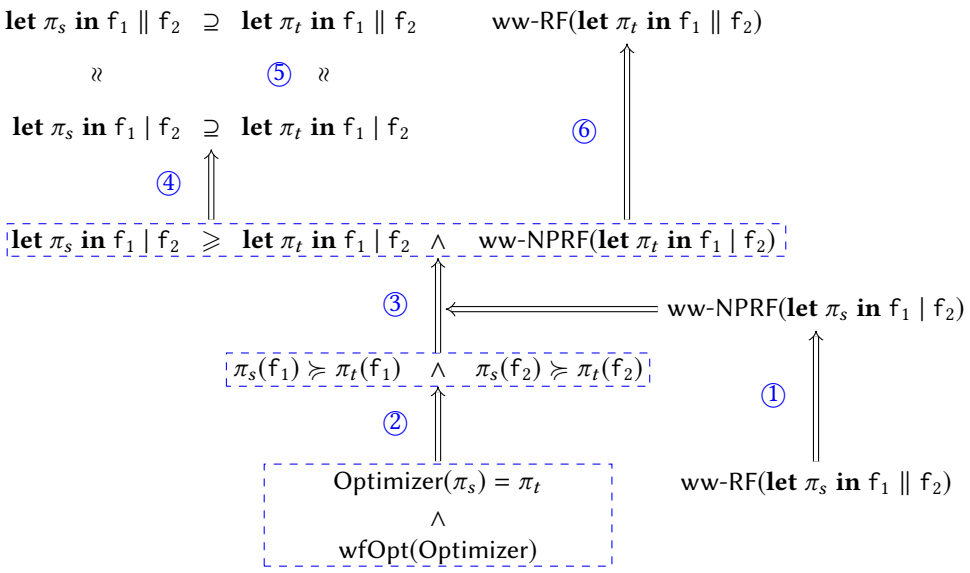


Fig. 2. Proof sketch

```
while(r < 10) {
    x_rlx := 1;
    r1 := y_na;
    r := r + 1;
}
```

LICM ⇝

```
t := y_na;
while(r < 10) {
    x_rlx := 1;
    r1 := t;
    r := r + 1;
}
```

(Observed in LLVM *)*

We also show that our thread-local simulation is able to support verifying the correctness of *instruction reordering* optimizations.

In this work, we focus on the correctness proof the code optimizations for eliminating redundant reads/writes and instruction reordering. We do not discuss the code optimizations on stack memory, such as *function inline*, *tail call*, *register promotion* and *register allocation&spilling*. The reason is that, if we do not consider the escape of the stack pointer of a thread to another thread, the stack memory is thread-local. It is plausible that we can convert the part of the thread-local memory of a thread to a form of partial mapping from memory locations to variables and let it be the local state of the thread. The interfering of the environment does not influence these code optimizations.

We establish a verification framework in Fig. 2. We define a non-preemptive semantics, which equals to promising semantics as shown in the step ⑤ in Fig. 2, and do the correctness proof of optimizers on such non-preemptive semantics. A well-defined optimizer should ensures that each source thread can establish the thread-local simulation with its corresponding target thread, shown as the step ②. Our thread-local simulation is compositional under the write-write race freedom assumption and can preserve the write-write race free property as shown in the step ③. We require that our thread-local simulation preserves the write-write race freedom, since we need to make sure that the correctness of the optimization is transitive. The whole program simulation ensures the refinement relation between the source and target programs under the non-preemptive

semantics as shown in the step ④. From the equivalence between promising semantics and the non-preemptive semantics, we establish the correctness of optimizers for concurrent programs with promising semantics.

In the following, we give more introductions to our approach to establish the correctness of optimizers for concurrent programs defined under promising semantics.

- ***Why we define the thread-local simulation on the non-preemptive semantics?*** As we have introduced, the non-preemptive semantics does not permit the thread switching and the promise step of the current thread after the execution of non-atomic steps. Such simplification can simplify our work in defining the thread-local simulation in two points.

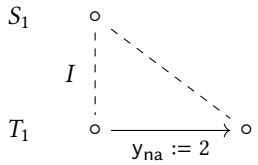
(1) The thread-local simulation should include an invariant for shared memory in target and source programs to depict the interaction with the environments. Defining the thread-local simulation on the non-preemptive semantics allows us to maintain the invariant in specific program points.

(2) Since we only consider the code optimization on non-atomic memory accesses, we can define the thread-local simulation about thread steps in such a simple form, which says that, if the target thread takes a non-atomic step, the source thread also takes some non-atomic steps to preserve the thread-local simulation; and if the target thread takes an atomic or promise step, the source thread also takes an atomic step or some promise steps to preserve the thread-local simulation.

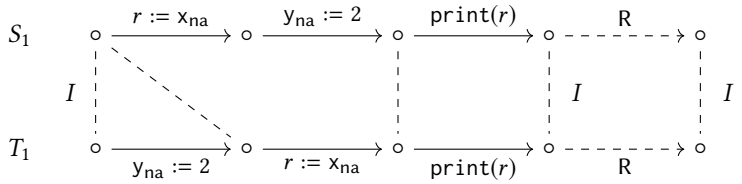
We use the following *instruction reordering* transformation to show that such two advantages can simplify the proof of some code optimizations.

$$\begin{array}{l} r := \mathsf{x}_{\mathsf{na}}; \\ \mathsf{y}_{\mathsf{na}} := 2; \\ \mathtt{print}(r); \end{array} \;\Big\|\; \mathsf{x}_{\mathsf{na}} := 3; \qquad \rightsquigarrow \qquad \begin{array}{l} \mathsf{y}_{\mathsf{na}} := 2; \\ r := \mathsf{x}_{\mathsf{na}}; \\ \mathtt{print}(r); \end{array} \;\Big\|\; \mathsf{x}_{\mathsf{na}} := 3;$$

For the source program, we call the thread on the left side S_1 and the thread on the right side S_2. For the target program, we call the thread on the left side T_1 and the thread on the right side T_2. We try to establish the thread-local upward simulation between S_1 and T_1. We define a simply invariant I, which just say that the memory in the target program and in the source program are strictly equal. Consider that T_1 executes "$\mathsf{y}_{\mathsf{na}} := 2$" first as the following shown.



If we want to take advantage (2) shown above, S_1 should not take any step. The reason is that S_1 need to execute "$r := \mathsf{x}_{\mathsf{na}}$" first, but it can not make sure which message it should read before T_1 executing "$r := \mathsf{x}_{\mathsf{na}}$". Let S_1 take zero step at this moment will break the invariant I introduced previously, since the execution of "$\mathsf{y}_{\mathsf{na}} := 2$" in T_1 will generate a new message in the target memory and cause the memory in target and source programs no longer the same. Thus, if we define the thread-local simulation on a preemptive semantics, which permits the interaction with the environments of the current thread in any program point, a problem will arise, since defining a thread-local simulation on a preemptive semantics requires us to maintain the invariant for shared memory in any program point. However, we can establish the thread-local simulation in our work, which is defined under the non-preemptive semantics, between S_1 and T_1 as the following shown.

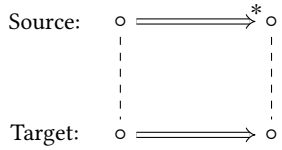


The invariant I only need to be reestablished after the execution of "print(r)", and the interactions with the environment of S_1 and T_1 do not need to be considered in proving the reordering of "$r := \mathsf{x}_{\mathsf{na}}; \mathsf{y}_{\mathsf{na}} := 2$". The interactions with the environment only need to be considered after the instruction reordering proof and after the execution of "print(r)".

- ***Why we require the write-write race freedom assumption?*** The write-write race freedom on non-atomic memory accesses assumption plays an important role in proving that our thread-local simulation is able to compose to a whole program simulation (shown as ③ in Fig. 2). The reason is that whole program step (called machine step in promising semantics) in promising semantics includes two components: (1) the current thread takes some steps; (2) after these steps, the promises of the current thread is *consistent* (or *certified*). The whole program step has the following form and we use $\mathcal{TP}$ to represent the thread pool

$$\frac{(\mathcal{TP}(\mathrm{t}), M) \longrightarrow (TS', M') \quad \mathsf{consistent}(TS', M')}{(\mathcal{TP}, \mathrm{t}, M) \Longrightarrow (\mathcal{TP}\{\mathrm{t} \rightsquigarrow TS'\}, \mathrm{t}, M')}$$

The promise consistency certification (shown as consistent(TS', M')) ensures that the current thread t is able to fulfill all its promises when executing in isolation. However, the certification does not start from the current memory M'. It starts from a capped memory [5] constructed from M'. In such construction, all timestamps intervals between existing messages are blocked by reservations. However, this will arise a problem in our work. The whole program simulation in our work has the following form, which is defined on the whole program step.



If the target program takes a whole program step, the source program can take some whole program steps and preserve the whole program simulation. From the definition of the whole program step, in the proof of the compositionality of our thread-local simulation, we need to prove that, if the current target thread can reach a thread configuration whose promises can be certified, the current source thread can also take some steps to a thread configuration whose promises can be certified. However, the problem here is that our thread-local simulation is defined from the current memory, but the promise certification starts from the capped memory. Thus, our thread-local simulation can not ensure the property which says, if the promises of the current target thread can be certified, the promises of the current source thread can also be certified.

$$\neg((\mathsf{consistent}(T) \wedge I \vdash T \preccurlyeq S) \implies \mathsf{consistent}(S))$$

Thus, we introduce the write-write race freedom assumption. And our thread-local simulation can make sure such property with the write-write race freedom assumption. The write-write race freedom forbids the following execution inserting messages using the timestamp intervals between existing messages, since

the capped memory blocks these timestamps.

$$(\mathsf{consistent}(T) \wedge I \vdash T \preccurlyeq S \wedge \mathsf{ww\text{-}RF}(S)) \implies \mathsf{consistent}(S)$$

It means that a thread from a write-write-race-free program can certify promises (for non-atomic writes) against the current memory instead of the capped memory.

Intuitively a *write-write race* means that two threads both (non-atomically) write to the same location, and neither write happens before the other. So, write-write race freedom forbids a thread t to write to a location when the memory contains a write of the same location made by another thread t′ and unobserved by t. This gives the same technical effect as the capped memory: t cannot write a message m when the memory already contains another message m' at the same location with a higher timestamp written by t′. We show that our thread-local simulation preserves the promises certification under write-write race freedom assumption and how a thread from a write-write-race-free program can certify promises (for non-atomic writes) against the current memory instead of the capped memory in details in Sec. 8.3.

- ***Why we permit read-write data race?*** The reason that we permits read-write race is that some optimizations, like *loop invariant code motion* in LLVM, may introduce read-write race in the target program during optimization. The following is an example of loop invariant code motion that will be performed in LLVM.

```
x_na := 20;          ||  r := y_acq;                  ~>   x_na := 20;          ||  r := y_acq;
y_rel := 1;          ||  if(r == 1) {                      y_rel := 1;          ||  if(r == 1) {
z_na := 5;           ||      r1 := x_na;                   z_na := 5;           ||      r1 := x_na;
                     ||      while(r1 < 10) {                                   ||      r' := z_na;
                     ||          r2 := z_na;                                    ||      while(r1 < 10) {
                     ||          r1 := r1+1;                                    ||          r2 := r';
                     ||      }                                                  ||          r1 := r1+1;
                     ||  }                                                      ||      }
                     ||                                                         ||  }
```

In the source program, the thread on the right side will not execute "$r_2 := \mathsf{z}_{\mathsf{na}}$", since the thread will never entry the loop. However, we can find that, after the loop invariant code motion optimization, there are read-write data race on accessing the variable z between two threads.

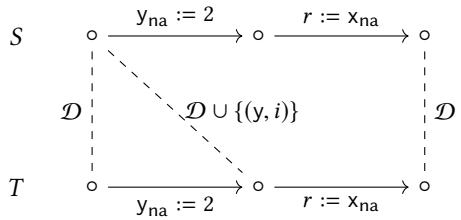
The method in CASCompCert does not support proving the correctness of loop invariant code motion in such form, since this method requires the data race freedom property of the source program preserves during compilation.

- ***How to make our thread-local simulation ensure write-write race freedom preserving?*** As shown in Fig. 2, our thread-local simulation is able to preserve the write-write race freedom. Thus, our thread-local simulation needs to restrict that the memory locations written by the execution of the target program should be the same or fewer than the execution of the source program and we introduce a delay set in the thread-local simulation to ensure such restriction. The delay set records the memory locations that has been written by the target thread but not by the source thread. Each memory location in the delay set also has an index, which restricts that the source thread has to write such memory location in finite steps. Consider the following instruction reordering transformation.

```
r := x_na;          ~>   y_na := 2;
y_na := 2;               r := x_na;
```

We establish the thread-local simulation between the target and source programs in the above transformation. Consider that the delay set is $\mathcal{D}$ before the instruction reordering proof. In the first step, the target thread executes "$\mathsf{y}_{\mathsf{na}} := 2$" and the source thread does not execution. We records the variable y in the delay

set and an index i to restrict that the source thread has to write the variable y in i steps. In the next step, the source thread executes "$\mathsf{y}_{\mathsf{na}} := 2$" and is able to remove the variable y in the delay set.



2 LANGUAGE

We define the language used to do optimizations in this section. We call the language shown in this section concur-SimpRTL. We will show its syntax and transition on the thread-local state in the following.

2.1 Syntax of concur-SimpRTL

$$
\begin{array}{rrcl}
\textit{(Fid)} & \mathtt{f} & \in & \mathbb{N} \qquad \textit{(Lab)} \quad l \in \mathbb{N} \qquad \textit{(Var)} \quad \mathtt{x}, \mathtt{y}, \mathtt{z} ::= \ldots \\
\textit{(Val)} & v & \in & \mathsf{Int32} \\
\textit{(Expr)} & e & ::= & r \mid v \mid e + e \mid e - e \mid e * e \\
\textit{(Instr)} & c & ::= & r := e \mid r := \mathtt{x}_{o_r} \mid \mathtt{x}_{o_w} := e \mid \mathtt{skip} \mid \mathtt{print}(e) \\
 & & \mid & r := \mathsf{CAS}_{o_r, o_w}(\mathtt{x}, e_r, e_w) \\
 & & \mid & \mathtt{fence\text{-}rel} \mid \mathtt{fence\text{-}acq} \mid \mathtt{fence\text{-}sc} \\
\textit{(BBlock)} & B & ::= & c, B \mid \mathtt{jmp}\ l \mid \mathtt{call}(\mathtt{f}, l_{ret}) \mid \mathtt{be}\ e, l_1, l_2 \mid \mathtt{return} \\
\textit{(Cdhp)} & C & \in & \textit{Lab} \rightharpoonup \textit{BBlock} \qquad \textit{(FunDef)} \quad \textit{Fd} ::= (C, l) \\
\textit{(Code)} & \pi & \in & \textit{Fid} \rightharpoonup \textit{FunDef} \\
\textit{(VarType)} & \iota & \in & \mathcal{P}(\textit{Var}) \\
\textit{(Prog)} & \mathbb{P} & ::= & \textbf{let}\ (\pi, \iota)\ \textbf{in}\ \mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n
\end{array}
$$

Fig. 3. Syntax of concur-SimpRTL language

We define the syntax of the language in Fig. 3. The instantiation of the code π is defined as a partial mapping from the identifier of the function to the definition of the code heap. The definition of the function *Fd* is a tuple that includes the code heap C, and the entry l of the function. The code heap C is a set of the basic block B, which is a sequence of instructions. We define a set ι to record the set of variables that can be performed atomic memory accesses. Achieving such set is not a difficult task. For example, the C programs use the keyword "`_Atomic`" to present the variables that can perform atomic memory accesses; the Java programs have some classes for atomic memory accesses, such as "`AtomicInteger`", or we can use the keyword "`volatile`"; Rust also has a number of atomic types, such as "`AtomicBool`" and "`AtomicU16`". Technically, dividing the locations into the atomic locations and the non-atomic locations also plays an important role in proving that our thread-local simulation ensures the preservation of the promise certification under the write-write race freedom assumption. We explain such point in Sec. 8.3.

We instantiate the thread local state in Fig. 4. The instantiation of the thread local state σ is a tuple includes: the register file R, the current basic block B, the current code heap C, the continuation K and the set of functions π. The continuation K records the levels of function calls. Each level of continuation K is a tuple, which includes the register r to save the return value, and the register file R and the code heap C of the caller.

2.2 Thread-local transition

We define the thread-local transition of concur-SimpRTL language in Fig. 5, which has the form of "$\sigma \xrightarrow{te} \sigma'$". Some auxiliary definitions used in defining the thread local transition are shown below. The register file in the initial state of the execution of a function has the following form.

$$R_\perp \quad ::= \quad \lambda r.0$$

$$\begin{array}{llll} \textit{(Cont)} & K & ::= & \epsilon \mid (R, B, C) :: K \\ \textit{(ThrdLocSt)} & \sigma & ::= & (R, B, C, K, \pi) \end{array} \qquad \begin{array}{llll} \textit{(RegFile)} & R & \in & \mathit{Reg} \to \mathit{Val} \end{array}$$

Fig. 4. Thread local state of concur-SimpRTL language

$$\frac{B = (r := e) :: B' \qquad R' = R\{r \rightsquigarrow [\![e]\!]_R\}}{(R, B, C, K, \pi) \xrightarrow{\tau} (R', B', C, K, \pi)} \qquad \frac{B = (r := \mathtt{x}_{o_r}) :: B' \qquad R' = R\{r \rightsquigarrow v\}}{(R, B, C, K, \pi) \xrightarrow{\mathsf{R}(o_r, \mathtt{x}, v)} (R', B', C, K, \pi)}$$

$$\frac{B = (\mathtt{x}_{o_w} := e) :: B' \qquad [\![e]\!]_R = v}{(R, B, C, K, \pi) \xrightarrow{\mathsf{W}(o_w, \mathtt{x}, v)} (R, B', C, K, \pi)} \qquad \frac{B = (\mathtt{print}(e) :: B') \qquad [\![e]\!]_R = v}{(R, B, C, K, \pi) \xrightarrow{\mathrm{out}(v)} (R, B', C, K, \pi)}$$

$$\frac{\begin{array}{c} B = (r := \mathtt{CAS}_{o_r, o_w}(\mathtt{x}, e_r, e_w)) :: B' \\ [\![e_r]\!]_R = v_r' \qquad [\![e_w]\!]_R = v_w \\ v_r = v_r' \qquad R' = R\{r \rightsquigarrow 1\} \end{array}}{(R, B, C, K, \pi) \xrightarrow{\mathsf{U}(o_r, o_w, \mathtt{x}, v_r, v_w)} (R', B', C, K, \pi)} \qquad \frac{\begin{array}{c} B = (r := \mathtt{CAS}_{o_r, o_w}(\mathtt{x}, e_r, e_w) :: B') \\ [\![e_r]\!]_R = v_r' \qquad [\![e_w]\!]_R = v_w \\ v_r \neq v_r' \qquad R' = R\{r \rightsquigarrow 0\} \end{array}}{(R, B, C, K, \pi) \xrightarrow{\mathsf{R}(o_r, \mathtt{x}, v_r)} (R', B', C, K, \pi)}$$

$$\frac{\begin{array}{c} B = \mathtt{call}(\mathtt{f}, l_{ret}) \qquad \pi(\mathtt{f}) = (C_0, l_0) \\ B_0 = C_0(l_0) \qquad B' = C(l_{ret}) \qquad K_0 = (R, B', C) :: K \end{array}}{(R, B, C, K, \pi) \xrightarrow{\tau} (R_\perp, B_0, C_0, K_0, \pi)}$$

$$\frac{B = \mathtt{return} \qquad K = (R_0, B_0, C_0) :: K_0}{(R, B, C, K, \pi) \xrightarrow{\tau} (R_0, B_0, C_0, K_0, \pi)} \qquad \frac{B = \mathtt{return}}{(R, B, C, \epsilon, \pi) \xrightarrow{\tau} \textbf{done}}$$

$$\frac{B = \mathtt{fence\text{-}rel} :: B'}{(R, B, C, K, \pi) \xrightarrow{\mathsf{F}_{\mathsf{rel}}} (R, B', C, K, \pi)} \qquad \frac{B = \mathtt{fence\text{-}acq} :: B'}{(R, B, C, K, \pi) \xrightarrow{\mathsf{F}_{\mathsf{acq}}} (R, B', C, K, \pi)}$$

$$\frac{B = \mathtt{fence\text{-}sc} :: B'}{(R, B, C, K, \pi) \xrightarrow{\mathsf{F}_{\mathsf{sc}}} (R, B', C, K, \pi)} \qquad \frac{B = \mathtt{jmp}\ l \qquad C(l) = B'}{(R, B, C, K, \pi) \xrightarrow{\tau} (R, B', C, K, \pi)}$$

$$\frac{\begin{array}{c} B = \mathtt{be}\ e, l_1, l_2 \qquad [\![e]\!]_R = v \\ (v = 0 \wedge C(l_1) = B') \vee (v \neq 0 \wedge C(l_2) = B') \end{array}}{(R, B, C, K, \pi) \xrightarrow{\tau} (R, B', C, K, \pi)}$$

Fig. 5. Thread local transition of concur-SimpRTL language

We instantiate the initialization function.

$$\mathsf{Init}(\pi, \mathtt{f}) \quad ::= \quad \begin{cases} (R_\perp, B, C, \epsilon, \pi) & \textit{if } \pi(\mathtt{f}) = (C, l) \text{ and } C(l) = B \\ \text{undef} & \textit{otherwise} \end{cases}$$

$$
\begin{array}{rrcl}
\textit{(Time)} & f, t & ::= & \mathbb{Q} \\
\textit{(TimeMap)} & T, \mathcal{S} & \in & \mathit{Var} \to \mathit{Time} \\
\textit{(View)} & V & \in & \{(T_{\mathsf{na}}, T_{\mathsf{rlx}}) \mid \forall \mathsf{x} \in \mathit{Var}.\ T_{\mathsf{na}}(\mathsf{x}) \le T_{\mathsf{rlx}}(\mathsf{x})\} \\
\textit{(Message)} & m & \in & \{\langle \mathsf{x} : v@(f, t], V\rangle \mid (f < t \vee f = t = 0) \vee V.T_{\mathsf{rlx}}(x) \le t\} \\
 & & & \quad \cup \{\langle \mathsf{x} : (f, t]\rangle \mid f < t\} \\
\textit{(Mem)} & M, P & \in & \mathcal{P}(\mathit{Message})
\end{array}
$$

Fig. 6. The memory defined as a message pool

3 PROMISING SEMANTICS

In this section, we define the semantics of concur-SimpRTL language based on *promising semantics*, which is taken from Lee, et al. [11].

We first give the definition of the memory. The memory in promising semantics is defined as a message pool. We define the memory formally in Fig. 6. The message in the memory has two types: concrete message (in the form of "$\langle \mathsf{x} : v@(f, t], V\rangle$") and reservation message (in the form of "$\langle \mathsf{x} : (f, t]\rangle$"). And We define the following notations for message.

$$
\begin{array}{lcll}
m.\mathsf{var} & ::= & x & \textit{if } m \in \{\langle \mathsf{x} : \_@(\_, \_], \_\rangle, \langle \mathsf{x} : (\_, \_]\rangle\} \\
m.\mathsf{from} & ::= & f & \textit{if } m \in \{\langle \_ : \_@(f, \_], \_\rangle, \langle \_ : (f, \_]\rangle\} \\
m.\mathsf{to} & ::= & t & \textit{if } m \in \{\langle \_ : \_@(\_, t], \_\rangle, \langle \_ : (\_, t]\rangle\} \\
m.\mathsf{val} & ::= & v & \textit{if } m = \langle \_ : v@(\_, \_], \_\rangle \\
m.\mathsf{view} & ::= & V & \textit{if } m = \langle \_ : \_@(\_, \_], V\rangle
\end{array}
$$

Two messages m_1 and m_2 are *disjoint*, denoted by $m_1 \# m_2$, if they have different locations or disjoint timestamp intervals:

$$
\begin{array}{rcl}
m_1 \# m_2 & ::= & m_1.\mathsf{var} \neq m_2.\mathsf{var} \vee \\
 & & m_1.\mathsf{to} < m_2.\mathsf{from} \ \vee \ m_2.\mathsf{to} < m_1.\mathsf{from}
\end{array}
$$

Two sets M_1 and M_2 of messages are disjoint, denoted by $M_1 \# M_2$:

$$
M_1 \# M_2 \ ::= \ \forall m_1 \in M_1, m_2 \in M_2.\ m_1 \# m_2.
$$

We write $M(\mathsf{x})$ for the sub-memory of the messages whose location is x, and $\widetilde{M}$ for the set of concrete messages in M.

$$
\begin{array}{rcl}
M(\mathsf{x}) & ::= & \{m \in M \mid m.\mathsf{var} = \mathsf{x}\} \\
\widetilde{M} & ::= & \{m \in M \mid m = \langle \_ : \_@(\_, \_], \_\rangle\}
\end{array}
$$

Given a timemap T and a memory M, we write $T \in M$ if the views of memory locations in T are in memory M.

$$
\begin{array}{rcl}
T \in M & ::= & \forall \mathsf{x} \in \mathit{Var}.\ \exists m \in \widetilde{M}.\ T(\mathsf{x}) = m.\mathsf{to} \\
V \in M & ::= & V.T_{\mathsf{na}} \in M \ \wedge \ V.T_{\mathsf{rlx}} \in M
\end{array}
$$

$$
\begin{array}{rrcl}
\textit{(ThrdView)} & \mathcal{V} & \in & \{(\textit{cur}, \textit{acq}, \textit{rel}) \mid \textit{cur}, \textit{acq} \in \textit{View} \\
 & & & \quad \wedge\, \textit{rel} \in \textit{Var} \to \textit{View} \\
 & & & \quad \wedge\, (\forall \mathsf{x} \in \textit{Var}.\ \textit{rel}(\mathsf{x}) \le \textit{cur} \le \textit{acq})\} \\
\textit{(ThrdState)} & \textit{TS} & \in & \{(\sigma, \mathcal{V}, P) \mid \forall m \in P.\ \mathcal{V}.\mathsf{cur}.T_{\mathsf{rlx}}(m.\mathsf{var}) < m.\mathsf{to}\} \\
\textit{(Tid)} & \mathsf{t} & \in & \mathbb{N} \\
\textit{(ThrdPool)} & \mathcal{TP} & \in & \textit{Tid} \to \textit{ThrdState} \\
\textit{(World)} & W & ::= & (\mathcal{TP}, \mathsf{t}, \mathcal{S}, M)^{\iota}
\end{array}
$$

$$
\begin{array}{rrcl}
\textit{(MemOrdR)} & o_r & ::= & \mathsf{na} \mid \mathsf{rlx} \mid \mathsf{acq} \qquad \textit{(MemOrdW)} \quad o_w ::= \mathsf{na} \mid \mathsf{rlx} \mid \mathsf{rel} \\
\textit{(ThrdEvt)} & te & ::= & \tau \mid \mathsf{R}(o_r, \mathsf{x}, v) \mid \mathsf{W}(o_w, \mathsf{x}, v) \mid \mathsf{U}(o_r, o_w, \mathsf{x}, v_r, v_w) \mid \mathsf{F}_{\mathsf{rel}} \mid \mathsf{F}_{\mathsf{acq}} \mid \mathsf{F}_{\mathsf{sc}} \\
 & & & \mid \mathsf{out}(v) \mid \mathsf{prm} \mid \mathsf{ccl} \mid \mathsf{rsv} \\
\textit{(ProgEvt)} & pe & ::= & \tau \mid \mathsf{out}(v) \mid \mathsf{sw}
\end{array}
$$

Fig. 7. Program state and Events

Inserting a new message into memory is defined below.

$$
M \xhookleftarrow{\mathsf{A}} m \quad ::= \quad \begin{cases} M \cup \{m\} & \textit{if } \{m\} \# M, \\ & \quad (m = \langle \mathsf{x} : \_@(f, t], \_\rangle \implies \\ & \quad\quad \neg(\exists m' \in M.\ m'.\mathsf{var} = \mathsf{x} \wedge m'.\mathsf{from} = t)) \\ \mathsf{undef} & \textit{otherwise} \end{cases}
$$

Splitting an existing message is defined below.

$$
\begin{array}{rcl}
M \xhookleftarrow{\mathsf{S}} m & ::= & \begin{cases} (M \backslash m) \cup & \\ \quad \{m, \langle \mathsf{x} : v@(t', t], V\rangle\} & \textit{if} \langle \mathsf{x} : v@(f, t], V\rangle \in M, f \le t' < t \\ \mathsf{undef} & \textit{otherwise} \end{cases} \\
\multicolumn{3}{r}{\textit{where } m = \langle \mathsf{x} : v@(f, t'], V'\rangle}
\end{array}
$$

The domain of the the memory is a set of pairs of the variable and the timestamp.

$$
\mathrm{dom}(M) \quad ::= \quad \{(\mathsf{x}, t) \mid \exists m \in M.\ m.\mathsf{var} = \mathsf{x} \wedge m.\mathsf{to} = t\}
$$

We define the program state in Fig. 7. The thread state $\textit{TS}$ is a triple, which consists a local state σ, thread view $\mathcal{V}$ and promises P. The definition of the local state σ needs to be instantiated in practice. We also define the memory model o, the thread event te and the program event pe in Fig. 7. The program state is a tuple, including the thread pool $\mathcal{TP}$, the global timestamp $\mathcal{S}$ used to depict the semantics of `fence-sc`, and the memory M. The program state should be well-defined as shown below.

$$
\begin{array}{rcl}
\mathrm{wdSt}(\mathcal{TP}, \mathcal{S}, M) & ::= & (\forall \mathsf{t} \in \mathrm{dom}(\mathcal{TP}).\ \mathcal{TP}(\mathsf{t}).P \subseteq M) \wedge \\
 & & (\forall \mathsf{t}_1, \mathsf{t}_2 \in \mathrm{dom}(\mathcal{TP}), \mathsf{t}_1 \neq \mathsf{t}_2.\ \mathcal{TP}(\mathsf{t}_1).P \# \mathcal{TP}(\mathsf{t}_2).P) \wedge \\
 & & (\forall \mathsf{t} \in \mathrm{dom}(\mathcal{TP}).\ \mathcal{TP}(\mathsf{t}).\mathcal{V} \in M) \wedge \mathcal{S} \in M \wedge (\forall m \in M.\ m.\mathsf{view} \in M)
\end{array}
$$

$$\frac{\text{for any } i \in \{1,\dots,n\}.\ \mathsf{Init}(\pi, \mathsf{f}_i) = \sigma_i \qquad TS_i = (\sigma_i, \mathcal{V}_\bot, \emptyset) \qquad \mathcal{TP} = \{1 \rightsquigarrow TS_1, \dots, n \rightsquigarrow TS_n\} \qquad \mathsf{t} \in \{1,\dots,n\} \qquad M = \{\langle \mathsf{x} : 0@(0,0], V_\bot\rangle \mid \mathsf{x} \in Var\}}{\textbf{let } (\pi, \iota) \textbf{ in } \mathsf{f}_1 \parallel \cdots \parallel \mathsf{f}_n \overset{load}{\Longrightarrow} (\mathcal{TP}, \mathsf{t}, \lambda \mathsf{x}.\, 0, M)^\iota}\ \text{(Load)}$$

$$\frac{\iota \vdash (\mathcal{TP}(\mathsf{t}), \mathcal{S}, M) \longrightarrow^+ (TS', \mathcal{S}', M') \quad \mathsf{consistent}(TS', M', \iota)}{(\mathcal{TP}, \mathsf{t}, \mathcal{S}, M)^\iota \overset{\tau}{\Longrightarrow} (\mathcal{TP}\{\mathsf{t} \rightsquigarrow TS'\}, \mathsf{t}, \mathcal{S}', M')^\iota}\ (\tau\text{-step}) \qquad \frac{\iota \vdash (\mathcal{TP}(\mathsf{t}), \mathcal{S}, M) \xrightarrow{\mathsf{out}(v)} (TS', \mathcal{S}', M')}{(\mathcal{TP}, \mathsf{t}, \mathcal{S}, M)^\iota \overset{\mathsf{out}(v)}{\Longrightarrow} (\mathcal{TP}\{\mathsf{t} \rightsquigarrow TS'\}, \mathsf{t}, \mathcal{S}', M')^\iota}\ (e\text{-step})$$

$$\frac{\mathsf{t}' \in \mathsf{dom}(\mathcal{TP})}{(\mathcal{TP}, \mathsf{t}, \mathcal{S}, M)^\iota \overset{\mathsf{sw}}{\Longrightarrow} (\mathcal{TP}, \mathsf{t}', \mathcal{S}, M)^\iota}\ (\text{sw-step})$$

$$\frac{\iota \vdash (\mathcal{TP}(\mathsf{t}), \mathcal{S}, M) \longrightarrow \textbf{done} \quad \mathsf{t}' \in \mathsf{dom}(\mathcal{TP} \backslash \{\mathsf{t}\})}{(\mathcal{TP}, \mathsf{t}, \mathcal{S}, M)^\iota \overset{\mathsf{sw}}{\Longrightarrow} (\mathcal{TP} \backslash \{\mathsf{t}\}, \mathsf{t}', \mathcal{S}, M)^\iota}\ (\text{thrd-term}) \qquad \frac{\iota \vdash (\mathcal{TP}(\mathsf{t}), \mathcal{S}, M) \longrightarrow \textbf{done} \quad \mathsf{dom}(\mathcal{TP}) = \{\mathsf{t}\}}{(\mathcal{TP}, \mathsf{t}, \mathcal{S}, M)^\iota \overset{\tau}{\Longrightarrow} \textbf{done}}\ (\text{prog-done})$$

$$\frac{\iota \vdash (\mathcal{TP}(\mathsf{t}), \mathcal{S}, M) \longrightarrow^* (TS', \mathcal{S}', M') \qquad \iota \vdash (TS', \mathcal{S}', M') \longrightarrow \textbf{abort}}{(\mathcal{TP}, \mathsf{t}, \mathcal{S}, M)^\iota \Longrightarrow \textbf{abort}}\ (\text{abort})$$

Fig. 8. Machine step

We start to present the definition of *promising semantics* taken from Lee, et al. [11]. We first give the machine step in Fig. 8. The (Load) rule presents the initialization of the program. The inital view $V_\bot$ is defined below.

$$V_\bot \quad ::= \quad (\lambda \mathsf{x}.0, \lambda \mathsf{x}.0)$$

Then, we define the thread view in the initial state.

$$\mathcal{V}_\bot \quad ::= \quad (V_\bot, V_\bot, \lambda \mathsf{x}.V_\bot)$$

$\mathsf{consistent}(TS, M, \iota)$ holds, *iff* for any $M_c \in \widehat{M}$,

$$\exists TS'.\ \iota \vdash (TS, \widehat{T}(M), M_c) \longrightarrow^* (TS', \_, \_) \ \wedge\ TS'.P = \emptyset$$

where $\widehat{M}$ is the method to construct the *capped memory*, which is proposed by Cho et al. [5]. We present the definition of constructing *capped memory* in Appendix A.

We present the thread-local step in Fig. 10. Here, we do not show the semantics of fence operations. Some auxiliary definitions used in defining the thread-local step are defined in Fig. 9. The notation ra represents that the memory order is either write release or read acquire.

$$\mathsf{ra} \ ::= \ \mathsf{rel} \mid \mathsf{acq} \qquad\qquad \mathsf{ra} \sqsupset \mathsf{rlx} \sqsupset \mathsf{na}$$

And we define the following notation to represent the thread-local step without observable events.

$$\longrightarrow \ \in \ \{\xrightarrow{te} \mid te \neq \mathsf{out}(v)\}$$

We define the abort-step in the following. A thread step is abort, if it either can not take a thread local transistion, or accesses the non-atomic location by atomic memory accesses, or accesses the atomic location by

$$\{\mathsf{x} @ t\} \quad ::= \quad V_\bot\{\mathsf{x} \rightsquigarrow t\} \qquad T_1 \sqcup T_2 \quad ::= \quad \{\mathsf{x} \rightsquigarrow t \mid t = \max(T_1(\mathsf{x}), T_2(\mathsf{x}))\}$$
$$V_1 \sqcup V_2 \quad ::= \quad (V_1.T_{\mathsf{na}} \sqcup V_2.T_{\mathsf{rlx}}, V_1.T_{\mathsf{na}} \sqcup V_2.T_{\mathsf{rlx}})$$

$$\frac{\begin{array}{c} o = \mathsf{na} \implies cur.T_{\mathsf{na}}(\mathsf{x}) \le t \\ o \in \{\mathsf{rlx}, \mathsf{ra}\} \implies cur.T_{\mathsf{rlx}}(\mathsf{x}) \le t \\ cur' = cur \sqcup V \sqcup (o \sqsupseteq \mathsf{ra}\,?\,V_r) \\ acq' = acq \sqcup V \sqcup (o \sqsupseteq \mathsf{rlx}\,?\,V_r) \\ \text{where } V = (o \sqsupseteq \mathsf{rlx}\,?\,\mathsf{x} @ t : V_\bot, \mathsf{x} @ t) \end{array}}{(cur, acq, rel) \xrightarrow{\mathsf{R}:o,\mathsf{x},t,V_r} (cur', acq', rel)}\ (\text{Rd-Helper}) \qquad \frac{\begin{array}{c} cur.T_{\mathsf{rlx}}(\mathsf{x}) \le t \\ cur' = cur \sqcup V \qquad acq' = acq \sqcup cur' \\ rel' = rel\{\mathsf{x} \rightsquigarrow (rel(\mathsf{x}) \sqcup V \sqcup (o = \mathsf{ra}\,?\,cur'))\} \\ V_w = (o \sqsupseteq \mathsf{rlx}\,?\,(rel'(\mathsf{x}) \sqcup V_r)) \\ \text{where } V = (\{\mathsf{x} @ t\}, \{\mathsf{x} @ t\}) \end{array}}{(cur, acq, rel) \xrightarrow{\mathsf{W}:o,\mathsf{x},t,V_r,V_w} (cur', acq', rel')}\ (\text{Wr-Helper})$$

$$\frac{\mathcal{S}' = acq.T_{\mathsf{rlx}} \sqcup \mathcal{S} \qquad cur' = acq' = (\mathcal{S}', \mathcal{S}') \qquad rel' = \lambda\_.\,(\mathcal{S}', \mathcal{S}')}{((cur, acq, rel), \mathcal{S}) \xrightarrow{\mathsf{F}_{\mathsf{sc}}} ((cur', acq', rel'), \mathcal{S}')}\ (\text{SCFence-Helper})$$

$$\frac{}{(P, M) \xrightarrow{m} (P, M \xleftarrow{\mathsf{A}} m)}\ (\text{New}) \qquad \frac{m \in P}{(P, M) \xrightarrow{m} (P \backslash \{m\}, M)}\ (\text{Fulfill}) \qquad \frac{P' = P \xleftarrow{\mathsf{S}} m}{(P, M) \xrightarrow{m} (P' \backslash \{m\}, M \xleftarrow{\mathsf{S}} m)}\ (\text{Split})$$

Fig. 9. Auxiliary definitions for thread local step

non-atomic memory accesses.

$$\iota \vdash ((\sigma, \mathcal{V}, P), \mathcal{S}, M) \longrightarrow \mathbf{abort} \quad ::=$$
$$\neg((\exists \sigma'.\, \sigma \longrightarrow \sigma') \vee (\sigma \longrightarrow \mathbf{done})) \vee (\exists \sigma', \mathsf{x}.\, \sigma \xrightarrow{\mathsf{U}(\_,\_,\mathsf{x},\_,\_)} \sigma' \wedge \mathsf{x} \notin \iota)$$
$$(\exists \sigma', \mathsf{x}, o, v.\, (\sigma \xrightarrow{\mathsf{R}(o,\mathsf{x},v)} \sigma' \vee \sigma \xrightarrow{\mathsf{W}(o,\mathsf{x},v)} \sigma') \wedge ((o = \mathsf{na} \wedge \mathsf{x} \in \iota) \vee (o \neq \mathsf{na} \wedge \mathsf{x} \notin \iota)))$$

We define the condition that the current thread is done.

$$\iota \vdash ((\sigma, \mathcal{V}, P), \mathcal{S}, M) \longrightarrow \mathbf{done} \quad ::= \quad \sigma \longrightarrow \mathbf{done} \wedge P = \emptyset$$

Behaviors of promising semantics. Behavior is defined as a set pf event trace.

$$(EvtTrace) \qquad \mathcal{B} \quad ::= \quad \mathbf{done} \mid \mathbf{abort} \mid \epsilon \mid \mathsf{out}(v) :: \mathcal{B}$$

$$ProgEtr(\mathbb{P}, \mathcal{B}) \ \text{ iff } \ \exists W, n.\, (\mathbb{P} \overset{load}{\Longrightarrow} W) \wedge Etr^n(W, \mathcal{B})$$

$$\frac{}{Etr^0(W, \epsilon)} \qquad \frac{W \implies \mathbf{abort}}{Etr^{n+1}(W, \mathbf{abort})} \qquad \frac{W \implies \mathbf{done}}{Etr^{n+1}(W, \mathbf{done})}$$

$$\frac{W \overset{\mathsf{out}(v)}{\Longrightarrow} W' \qquad Etr^n(W', \mathcal{B})}{Etr^{n+1}(W, \mathsf{out}(v) :: \mathcal{B})} \qquad \frac{W \overset{\tau/\mathsf{sw}}{\Longrightarrow} W' \qquad Etr^n(W', \mathcal{B})}{Etr^{n+1}(W, \mathcal{B})}$$

$$\frac{\sigma \xrightarrow{\tau} \sigma'}{\iota \vdash ((\sigma, \mathcal{V}, P), \mathcal{S}, M) \xrightarrow{\tau} ((\sigma', \mathcal{V}, P), \mathcal{S}, M)}\ \text{(Silent)} \qquad \frac{\sigma \xrightarrow{\text{out}(v)} \sigma' \quad (\mathcal{V}, \mathcal{S}) \xrightarrow{\text{F}_{\text{sc}}} (\mathcal{V}', \mathcal{S}')}{\iota \vdash ((\sigma, \mathcal{V}, \emptyset), \mathcal{S}, M) \xrightarrow{\text{out}(v)} ((\sigma', \mathcal{V}', \emptyset), \mathcal{S}', M)}\ \text{(Output)}$$

$$\frac{\begin{array}{c}\sigma \xrightarrow{\text{R}(o_r, \text{x}, v)} \sigma' \\ \langle \text{x} : v@(\_, t], V_r \rangle \in M \\ \mathcal{V} \xrightarrow{\text{R}:o_r, x, t, V_r} \mathcal{V}' \\ (o_r = \text{na} \wedge \text{x} \notin \iota) \vee (o_r \neq \text{na} \wedge \text{x} \in \iota)\end{array}}{\iota \vdash ((\sigma, \mathcal{V}, P), \mathcal{S}, M) \xrightarrow{\text{R}(o_r, \text{x}, v)} ((\sigma', \mathcal{V}', P), \mathcal{S}, M)}\ \text{(Read)} \qquad \frac{\begin{array}{c}\sigma \xrightarrow{\text{W}(o_w, \text{x}, v)} \sigma' \\ o_w = \text{ra} \implies \forall m' \in P(x).\ m'.\text{view} = V_\perp \\ m = \langle \text{x} : v@(\_, t], V_w \rangle \\ (P, M) \xrightarrow{m} (P', M') \\ \mathcal{V} \xrightarrow{\text{W}:o_w, x, t, V_\perp, V_w} \mathcal{V}' \\ (o_w = \text{na} \wedge \text{x} \notin \iota) \vee (o_w \neq \text{na} \wedge \text{x} \in \iota)\end{array}}{\iota \vdash ((\sigma, \mathcal{V}, P), \mathcal{S}, M) \xrightarrow{\text{W}(o_w, \text{x}, v)} ((\sigma', \mathcal{V}', P'), \mathcal{S}, M')}\ \text{(Write)}$$

$$\frac{\begin{array}{c}\sigma \xrightarrow{\text{U}(o_r, o_w, \text{x}, v_r, v_w)} \sigma' \\ o_r \in \{\text{rlx}, \text{acq}\} \quad o_w \in \{\text{rlx}, \text{rel}\} \quad \text{x} \in \iota \\ o_w = \text{ra} \implies \forall m' \in P(x).\ m'.\text{view} = V_\perp \\ \langle \text{x} : v_r@(\_, t_r], V_r \rangle \in M \quad m_w = \langle \text{x} : v_w@(t_r, t_w], V_w \rangle \\ (P, M) \xrightarrow{m_w} (P', M') \\ \mathcal{V} \xrightarrow{\text{R}:o_r, \text{x}, t_r, V_r} \xrightarrow{\text{W}:o_w, \text{x}, t_w, V_r, V_w} \mathcal{V}'\end{array}}{\iota \vdash ((\sigma, \mathcal{V}, P), \mathcal{S}, M) \xrightarrow{\text{U}(o_r, o_w, \text{x}, v_r, v_w)} ((\sigma', \mathcal{V}', P'), \mathcal{S}, M')}\ \text{(Update)}$$

$$\frac{\sigma \xrightarrow{\text{F}_{\text{rel}}} \sigma' \quad rel' = \lambda\_.cur \quad \forall m \in P.\ m.\text{view} = V_\perp}{\iota \vdash ((\sigma, (cur, acq, rel), P), \mathcal{S}, M) \xrightarrow{\text{F}_{\text{rel}}} ((\sigma', (cur, acq, rel'), P), \mathcal{S}, M)}\ \text{(Rel-Fence)}$$

$$\frac{\sigma \xrightarrow{\text{F}_{\text{acq}}} \sigma' \quad cur' = acq}{\iota \vdash ((\sigma, (cur, acq, rel), P), \mathcal{S}, M) \xrightarrow{\text{F}_{\text{acq}}} ((\sigma', (cur', acq, rel), P), \mathcal{S}, M)}\ \text{(Acq-Fence)}$$

$$\frac{m.\text{view} \in M' \quad (P' = P \cup \{m\} \wedge M' = M \xleftarrow{\text{A}} m) \vee (P' = P \xleftarrow{\text{S}} m \wedge M' = M \xleftarrow{\text{S}} m)}{\iota \vdash ((\sigma, \mathcal{V}, P), \mathcal{S}, M) \xrightarrow{\text{prm}} ((\sigma, \mathcal{V}, P'), \mathcal{S}, M')}\ \text{(Promise)}$$

$$\frac{m = \langle \text{x} : (f, t] \rangle \quad M' = M \xleftarrow{\text{A}} m}{\iota \vdash ((\sigma, \mathcal{V}, P), \mathcal{S}, M) \xrightarrow{\text{rsv}} ((\sigma, \mathcal{V}, P \cup \{m\}), \mathcal{S}, M')}\ \text{(Reserve)}$$

$$\frac{m = \langle \text{x} : (f, t] \rangle}{\iota \vdash ((\sigma, \mathcal{V}, P), \mathcal{S}, M) \xrightarrow{\text{ccl}} ((\sigma, \mathcal{V}, P \backslash \{m\}), \mathcal{S}, M \backslash \{m\})}\ \text{(Cancel)}$$

Fig. 10. Thread step

4 WRITE-WRITE RACE FREEDOM

In this section, we define the write-write race freedom under promising semantics in Fig. 11. A program is write-write race free if it does not contain the write-write race. We explain what is the write-write race in the

$$\frac{\mathcal{TP}(\mathrm{t}) = (\sigma, \mathcal{V}, P) \quad \sigma \xrightarrow{\mathrm{W(na,x,\_)}} \_ \quad \langle \mathrm{x} : v@(\_, t], \_\rangle \in (M \backslash P) \quad \mathcal{V}.\mathrm{cur}.T_{\mathrm{rlx}}(\mathrm{x}) < t}{(\mathcal{TP}, \mathrm{t}, \mathcal{S}, M)^{\iota} \Longmapsto \text{ww-Race}}$$

$$\frac{\mathbb{P} \xRightarrow{load} W \quad W \Longrightarrow^{*} W' \quad W' \Longmapsto \text{ww-Race}}{\mathbb{P} \Longmapsto \text{ww-Race}}$$

$$\text{ww-RF}(\mathbb{P}) \quad ::= \quad \neg(\mathbb{P} \Longmapsto \text{ww-Race})$$

Fig. 11. Write-write race freedom under promising semantics

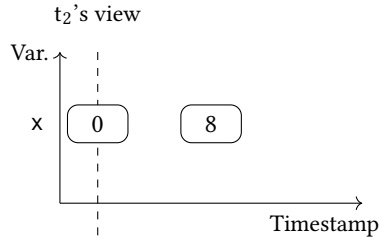


Fig. 12. The view of the thread t_2 to the location of the variable x (write-write race)

program. Consider the following program.

$$\begin{array}{l|l} \mathrm{x}_{\mathrm{na}} := 8; & r := \mathrm{y}_{\mathrm{rlx}}; \\ \mathrm{y}_{\mathrm{rel}} := 1; & \texttt{if}(r) \ \{ \\ & \quad \mathrm{x}_{\mathrm{na}} := 2; \\ & \} \end{array}$$

For this program, we call the thread on the left side t_1 and the thread on the right side t_2. This program contains write-write race, since the execution of "$\mathrm{x}_{\mathrm{na}} := 8$" in the thread t_1 and the execution of "$\mathrm{x}_{\mathrm{na}} := 2$" in the thread t_2 does not have *happen-before* relation. The happen-before relation means that, when the thread t_2 executes "$\mathrm{x}_{\mathrm{na}} := 2$", it does not know whether the execution of "$\mathrm{x}_{\mathrm{na}} := 8$" in the thread t_1 has been done or not as shown in Fig. 12. The execution of "$\mathrm{x}_{\mathrm{na}} := 2$" in the thread t_2 can insert the message before the message valued 8, which means that the execution of "$\mathrm{x}_{\mathrm{na}} := 2$" is done before the execution of "$\mathrm{x}_{\mathrm{na}} := 8$", or it can insert the message after the message valued 8, which means that the execution of "$\mathrm{x}_{\mathrm{na}} := 2$" is done after the execution of "$\mathrm{x}_{\mathrm{na}} := 8$".

5 PROOF GOAL

In this section, we formulate the correctness of optimizers, which optimize program under promising semantics. The optimizer is a tranformation from the source program to the target program.

$$\textit{(Optimizer)} \qquad \mathsf{Optimizer} \in (\mathit{Code} \times \mathit{VarType}) \rightharpoonup \mathit{Code}$$

The correctness of the optimizers is defined in Def. 5.1.

Definition 5.1 (correctness of optimizers).

$$\begin{aligned}\mathsf{Correct}(\mathsf{Optimizer}) \triangleq\ & \forall \pi_s, \pi_t, \iota.\ \mathsf{Optimizer}(\pi_s, \iota) = \pi_t \wedge \\ & \quad \mathsf{ww\text{-}RF}(\textbf{let}\ (\pi_s, \iota)\ \textbf{in}\ \mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n) \wedge \\ & \quad \mathsf{Safe}(\textbf{let}\ (\pi_s, \iota)\ \textbf{in}\ \mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n) \\ & \Longrightarrow (\textbf{let}\ (\pi_t, \iota)\ \textbf{in}\ \mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n \subseteq \textbf{let}\ (\pi_s, \iota)\ \textbf{in}\ \mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n \wedge \\ & \quad \mathsf{ww\text{-}RF}(\textbf{let}\ (\pi_t, \iota)\ \textbf{in}\ \mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n) \wedge \\ & \quad \mathsf{Safe}(\textbf{let}\ (\pi_t, \iota)\ \textbf{in}\ \mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n))\end{aligned}$$

Proving the correctness of the optimizer, which consists of multiple optimization passes, can be obtained by the transitive of the definition of the correct optimizer. We first define the composition of two optimizers below.

$$(\mathsf{Optimizer}_1 \circ \mathsf{Optimizer}_2)(\pi_s, \iota) \triangleq \begin{cases} \mathsf{Optimizer}_2(\pi_m, \iota) & \textit{if}\ \pi_m = \mathsf{Optimizer}_1(\pi_s, \iota) \\ \mathsf{undef} & \textit{otherwise} \end{cases}$$

We show the transitive of correct optimizer in Lemma. 5.2.

Lemma 5.2 (correct optimizer transitive).

$$\begin{aligned}&\forall \mathsf{Optimizer}_1, \mathsf{Optimizer}_2. \\ &\quad (\mathsf{Correct}(\mathsf{Optimizer}_1) \wedge \mathsf{Correct}(\mathsf{Optimizer}_2)) \\ &\Longrightarrow \mathsf{Correct}(\mathsf{Optimizer}_1 \circ \mathsf{Optimizer}_2)\end{aligned}$$

The definition of the data race freedom will be introduced in Sec. 4. Below, we define the program safety. The execution of a safe program will not abort.

$$\mathsf{Safe}(W) \triangleq \neg(\exists W'.\ W \Longrightarrow^* W' \wedge W' \Longrightarrow \textbf{abort})$$

$$\mathsf{Safe}(\mathbb{P}) \triangleq (\exists W.\ \mathbb{P} \xRightarrow{\mathit{load}} W) \wedge (\forall W.\ (\mathbb{P} \xRightarrow{\mathit{load}} W) \Longrightarrow \mathsf{Safe}(W))$$

For example, the following program is not safe, since we do an atomic memory access on the non-atomic location.

$$\mathtt{x}_{\mathsf{rlx}} := 3 \qquad (\textit{where}\ \ \iota(\mathtt{x}) = \mathsf{na})$$

The following program is also not safe, since the execution of the program accesses the undefined variable.

$$\mathtt{x}_{\mathsf{rlx}} := 3 \qquad (\textit{where}\ \ \mathtt{x} \notin \mathrm{dom}(\iota))$$

However, a safe program in our work does not mean that the program can always execute the current instruction successfully. Consider the following example.

$$\mathsf{CAS}_{\mathsf{rlx},\mathsf{rlx}}(\mathtt{x}, 0, 1); \quad \parallel \quad \mathtt{y}_{\mathsf{rlx}} := 1;$$

The above program satisfies our safe program definition straight-forwardly. However, it does not mean that such program can always execute its current instruction successfully. Consider the following execution.

	$\mathtt{t}_1$	$\mathtt{t}_2$
(1)		reserve $\langle \mathtt{x} : (0, 2] \rangle$
(2)	$\mathsf{CAS}_{\mathsf{rlx},\mathsf{rlx}}(\mathtt{x}, 0, 1)$?	

Since the thread t_2 may reserve the timestamps that will be used by the execution of the instruction $\mathsf{CAS}_{\mathsf{rlx},\mathsf{rlx}}(x, 0, 1)$ in the thread t_1, the thread t_1 can not execute its current instruction successfully.

We will prove that constant propagation (ConstProp), dead code elimination (DCE), common subexpression elimination (CSE) and loop invariant code motion (LICM) satisfy the definition of correct optimizer defined in Def. 5.1.

Theorem 5.3 (Correct Optimizers).

$$\mathsf{Correct}(\mathtt{ConstProp}) \land \mathsf{Correct}(\mathtt{DCE}) \land \mathsf{Correct}(\mathtt{CSE}) \land \mathsf{Correct}(\mathtt{LICM})$$

We will give the implementations of these optimizers in the following sections.

The thread local upward simulation will be defined in the following section. as the form of "$I, \iota \models \pi_t \preccurlyeq \pi_s$". We need to prove that the results of constant propagation and dead code elimination optimizations satisfies such simulation.

Definition 5.4 (Well-formed optimizer).

$$\mathsf{wfOpt}(\mathsf{Optimizer}) \triangleq \forall \pi_t, \pi_s, \iota.\ \mathsf{Optimizer}(\pi_s, \iota) = \pi_t \implies \exists I.\ I, \iota \models \pi_t \preccurlyeq \pi_s$$

The definition of the well-formed optimizer shows the correctness of the step ② in Fig. 2.

Lemma 5.5 (Well-formed optimizer implies correct optimizer).

$$\forall \mathsf{Optimizer}.\ \mathsf{wfOpt}(\mathsf{Optimizer}) \implies \mathsf{Correct}(\mathsf{Optimizer})$$

6 NON-PREEMPTIVE SEMANTICS

We try to define the non-preemptive semantics, which is equivalent to promising semantics.

In the following introduction, we use the "`atom { C }`" to represent that the execution of C will not be interrupted by other threads and the promise steps. We consider which program points are permitted to do thread switching in the following. We call the thread on the left side t_1 and the thread on the right side t_2 in the following introduction.

(1) *After the execution of the atomic memory accesses*: Permitting thread switching after the atomic (release) memory write is essential. Consider the program shown below.

```
x_na := 2;          ||  r1 := y_acq;
y_rel := 1;         ||  if(r1) {
while(true);        ||      r2 := x_na;
                    ||      while(true);
                    ||  }
```

The execution of the above program shown below demonstrates that it is essential to permit thread switching after atomic (release) write.

①	t_1 : $x_{na} := 2$
②	t_1 : $y_{rel} := 1$
③	t_2 : $r_1 := y_{rel}$ //1
④	t_2 : $r_2 := x_{na}$ //2
⑤	t_2 : `while(true)`

However, we do not find that thread switching after atomic read and relaxed read/write is essential. For simple representation, we still permit the thread switching after the execution of the atomic read.

(2) *After the execution of the promise step*: Permitting the thread switching after the execution of the promise step is also essential, since one thread will read the promise generated by the other thread. Consider the following example.

```
r1 := x_rlx;        ||  do {
y_rlx := 1;         ||      r2 := y_rlx;
                    ||  } while(r2 == 1);
```

We consider the following execution. and call the thread on the left side t_1 and the thread on the right side t_2. The thread t_1 promises "$z_{rlx} := 1$", and the thread t_2 reads such promise and loops forever. It seems that the only way to define a non-preemptive semantics is to allow thread switch after promise step.

①	t_1 : promise $\langle y : 1@(1, 2], \{y@1\}\rangle$
②	t_2 : $r_2 := y_{rlx}$ //1
③	t_2 : $r_2 := y_{rlx}$ //1
④	t_2 : $r_2 := y_{rlx}$ //1
	...

$$
\begin{array}{rrcl}
\textit{(AtomBit)} & \beta & \in & \circ \mid \bullet \\
\textit{(NPWorld)} & \hat{W} & ::= & (\mathcal{TP}, \mathrm{t}, \mathcal{S}, M, \beta)^{\iota} \\
\textit{(NPProg)} & \hat{\mathbb{P}} & ::= & \textbf{let}\ (\pi, \iota)\ \textbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n
\end{array}
$$

Fig. 13. Syntax and State of Non-preemptive semantics

(3) *After the execution of the fence operations*: We find that permit thread switching after the execution of fence operations is also essential. Consider the example shown below.

$\mathtt{x}_{\mathrm{na}} := 8;$ `fence-rel;` $r_1 := \mathtt{y}_{\mathrm{rlx}};$ $\mathtt{z}_{\mathrm{rlx}} := 1;$	$r_2 := \mathtt{z}_{\mathrm{rlx}};$ $\mathtt{y}_{\mathrm{rlx}} := r_2;$

The above program may generate the following execution.

①	$\mathrm{t}_1 : \mathtt{x}_{\mathrm{na}} := 8$
②	$\mathrm{t}_1 :$ `fence-rel`
③	$\mathrm{t}_1 :$ promise $\langle \mathtt{z} : 1@(2,3], \{\mathtt{x}@3\}\rangle$
④	$\mathrm{t}_2 : r_2 := \mathtt{z}_{\mathrm{rlx}}$ //1
⑤	$\mathrm{t}_2 : \mathtt{y}_{\mathrm{rlx}} := r_2$
⑥	$\mathrm{t}_1 : r_1 := \mathtt{y}_{\mathrm{rlx}}$ //1
⑦	$\mathrm{t}_1 : \mathtt{z}_{\mathrm{rlx}} := 1$ (* fulfill promise *)

Consider whether the example shown above is equivalent to the following program.

`atom {`
$\quad \mathtt{x}_{\mathrm{na}} := 8;$
$\quad$ `fence-rel;`
$\quad r_1 := \mathtt{y}_{\mathrm{rlx}};$
`}`
$\mathtt{z}_{\mathrm{rlx}} := 1;$
	$r_2 := \mathtt{z}_{\mathrm{rlx}};$
	$\mathtt{y}_{\mathrm{rlx}} := r_2;$

The answer is wrong. Since promising semantics does not permit that the atomic memory access after the fence release in the program order promises before the execution of the fence release, the program need to exit atomic block after the execution of the fence release.
Permitting thread switching after the execution of the acquire fence seems not essential. But for simple presentation, we still permit thread switching after the acquire fence execution.

(4) *After generating observable events and the current thread done*: Such two cases is straight-forward.

Non-preemptive semantics. We show the definition of the non-preemptive semantics in the following.
$\mathsf{consistent}_{\mathrm{NP}}(TS, M, \beta, \iota)$ holds, *iff* for any $M_c \in \widehat{M}$,

$$\exists TS'.\ \iota \vdash (TS, \widehat{T}(M), M_c, \beta) \longmapsto^{*} (TS', \_, \_, \_) \ \wedge\ TS'.P = \emptyset$$

$$
\begin{array}{lrcl}
\textit{(NA)} & \textit{na} & \in & \{\tau, \mathsf{W}(\mathsf{na}, \_, \_), \mathsf{R}(\mathsf{na}, \_, \_)\} \\
\textit{(PRC)} & \textit{prc} & \in & \{\mathsf{prm}, \mathsf{rsv}, \mathsf{ccl}\} \\
\textit{(AT)} & \textit{at} & \in & \{\textit{te} \mid \textit{te} \notin (\textit{NAtm} \cup \textit{PRC})\}
\end{array}
$$

$$
\begin{array}{l}
\iota \vdash (TS, \mathcal{S}, M, \beta) \xrightarrow{te} (TS', \mathcal{S}', M', \beta') \ ::= \\
\quad \iota \vdash (TS, \mathcal{S}, M) \xrightarrow{te} (TS', \mathcal{S}', M') \wedge \\
\quad (te \in \{\mathsf{prm}, \mathsf{rsv}\} \implies \beta = \beta' = \circ) \wedge (te = \mathsf{ccl} \implies \beta = \beta') \wedge \\
\quad (te \in NA \implies \beta' = \bullet) \wedge (te \in AT \implies \beta' = \circ)
\end{array}
$$

Fig. 14. Auxiliary definitions in auxiliary promising semantics

$$
\frac{\begin{array}{c}\text{for any } i \in \{1, \ldots, n\}.\ \mathsf{Init}(\pi, \mathsf{f}_i) = \sigma_i \qquad TS_i = (\sigma_i, \mathcal{V}_\bot, \emptyset) \\ \mathcal{TP} = \{1 \rightsquigarrow TS_1, \ldots, n \rightsquigarrow TS_n\} \qquad \mathsf{t} \in \{1, \ldots, n\} \qquad M = \{\langle \mathsf{x} : v@(0,0], V_\bot \rangle \mid \mathsf{x} \in \mathit{Var}\}\end{array}}{\textbf{let } (\pi, \iota) \textbf{ in } \mathsf{f}_1 \mid \ldots \mid \mathsf{f}_n :\xRightarrow{load} (\mathcal{TP}, \mathsf{t}, \lambda \mathsf{x}.\, 0, M, \circ)^\iota}
$$

$$
\frac{\iota \vdash (\mathcal{TP}(\mathsf{t}), \mathcal{S}, M, \beta) \longmapsto^+ (TS', \mathcal{S}', M', \beta') \quad \mathsf{consistent}_{\mathsf{NP}}(TS', M', \beta', \iota)}{(\mathcal{TP}, \mathsf{t}, \mathcal{S}, M, \beta)^\iota :\xRightarrow{\tau} (\mathcal{TP}\{\mathsf{t} \rightsquigarrow TS'\}, \mathsf{t}, \mathcal{S}', M', \beta')^\iota}
$$

$$
\frac{\iota \vdash (\mathcal{TP}(\mathsf{t}), \mathcal{S}, M, \beta) \xmapsto{\mathrm{out}(v)} (TS', \mathcal{S}', M', \circ)}{(\mathcal{TP}, \mathsf{t}, \mathcal{S}, M, \beta)^\iota :\xRightarrow{\mathrm{out}(v)} (\mathcal{TP}\{\mathsf{t} \rightsquigarrow TS'\}, \mathsf{t}, \mathcal{S}', M', \circ)^\iota}
$$

$$
\frac{\mathsf{t} \in \mathrm{dom}(\mathcal{TP})}{(\mathcal{TP}, \mathsf{t}, \mathcal{S}, M, \circ)^\iota :\xRightarrow{\mathsf{sw}} (\mathcal{TP}, \mathsf{t}', \mathcal{S}, M, \circ)^\iota}
$$

$$
\frac{\iota \vdash (\mathcal{TP}(\mathsf{t}), \mathcal{S}, M) \longrightarrow \textbf{done} \quad \mathsf{t}' \in \mathrm{dom}(\mathcal{TP} \backslash \{\mathsf{t}\})}{(\mathcal{TP}, \mathsf{t}, \mathcal{S}, M, \beta)^\iota :\xRightarrow{\mathsf{sw}} (\mathcal{TP} \backslash \{\mathsf{t}\}, \mathsf{t}', \mathcal{S}, M, \circ)^\iota}
$$

$$
\frac{\iota \vdash (\mathcal{TP}(\mathsf{t}), \mathcal{S}, M) \longrightarrow \textbf{done} \quad \mathrm{dom}(\mathcal{TP}) = \{\mathsf{t}\}}{(\mathcal{TP}, \mathsf{t}, \mathcal{S}, M, \beta)^\iota :\Longrightarrow \textbf{done}}
$$

$$
\frac{\iota \vdash (\mathcal{TP}(\mathsf{t}), \mathcal{S}, M, \beta) \longmapsto^* (TS', \mathcal{S}', M', \beta') \quad \iota \vdash (TS', \mathcal{S}', M') \longrightarrow \textbf{abort}}{(\mathcal{TP}, \mathsf{t}, \mathcal{S}, M, \beta)^\iota :\Longrightarrow \textbf{abort}}
$$

Fig. 15. Non-preemptive semantics

We permit that the thread takes a cancel step after the non-atomic step. The reason is that we need additional cancel steps to fulfill the reservations. Consider the following example.

```
x_na := 1;
while(true);
y_rlx := 2;
```

Consider the following execution of the above program under the non-preemptive semantics.

(1)	reservation $\langle y : (0, 1]\rangle$ $x_{na} := 1$
(2)	`while(true)`

We can find that, after the execution of the step (1), the atomic bit is "•". If we do not permit executing cancel steps in the promise consistency certification, the reservation "$\langle y : (0, 1]\rangle$" can not be fulfilled.

Note that we can not view the cancel step as non-atomic step, since this will cause a problem in the proof of the equvialence of the na promise-free semantics and the non-preemptive semantics under the data race freedom assumption. The problem is that reordering the execution of non-atomic steps and atomic step is incorrect when viewing the cancel step as non-atomic step. Consider the following program.

$$x_{na} := 1 \quad \Big\| \quad \begin{array}{l} z_{na} := 2; \\ \mathsf{FADD}_{acq,rlx}(y, 1); \end{array}$$

We consider the following execution of the above program under promising semantics.

	t_1	t_2
(P-1)	promise $\langle y : (0, 2]\rangle$ (* reservation *)	
(P-2)		$z_{na} := 2$
(P-3)	$x_{na} := 1$; cancel $\langle y : (0, 2]\rangle$	
(P-4)		$\mathsf{FADD}_{acq,rlx}(y, 1)$
(P-5)		**done**
(P-6)	**done**	

If we view the cancel step as the non-atomic step, we need to construct the following execution of the above program under the non-preemptive semantics.

	t_1	t_2
(NP-1)	promise $\langle y : (0, 2]\rangle$ (* reservation *)	
(NP-2)		$z_{na} := 2$ $\mathsf{FADD}_{acq,rlx}(y, 1)$;
(NP-3)		**done**
(NP-4)	$x_{na} := 1$; cancel $\langle y : (0, 2]\rangle$	
(NP-5)	**done**	

We find that the step (NP-2) can not be taken, since the timestamps required for execution "$\mathsf{FADD}_{acq,rlx}(y, 1)$" has been reserved.

Behaviors of programs under the non-preemptive semantics. We define the behavior of programs under the non-preemptive semantics shown below.

$$NPProgEtr(\hat{\mathbb{P}}, \mathcal{B}) \text{ iff } \exists \hat{W}, n.\ (\hat{\mathbb{P}} :\overset{load}{\Longrightarrow} \hat{W}) \wedge NPEtr^n(\hat{W}, \mathcal{B})$$

$$\frac{}{NPEtr^0(\hat{W}, \epsilon)} \qquad \frac{\hat{W} :\Longrightarrow \textbf{abort}}{NPEtr^{n+1}(\hat{W}, \textbf{abort})} \qquad \frac{\hat{W} :\Longrightarrow \textbf{done}}{NPEtr^{n+1}(\hat{W}, \textbf{done})}$$

$$\frac{\hat{W} :\overset{out(v)}{\Longrightarrow} \hat{W}' \quad NPEtr^n(\hat{W}', \mathcal{B})}{NPEtr^{n+1}(\hat{W}, out(v) :: \mathcal{B})} \qquad \frac{\hat{W} :\overset{\tau/sw}{\Longrightarrow} \hat{W}' \quad NPEtr^n(\hat{W}', \mathcal{B})}{NPEtr^{n+1}(\hat{W}, \mathcal{B})}$$

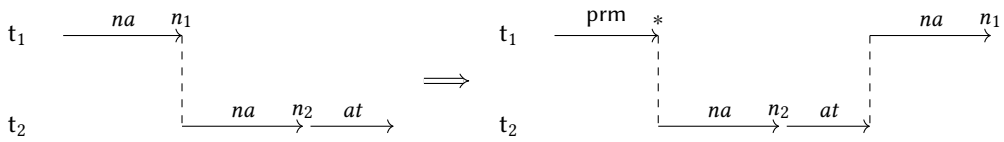
We need to prove the equivalence between the na promise-free semantics and the non-preemptive semantics under the DRF assumption as shown in Lemma. 6.1.

Lemma 6.1 (Semantics equvialence - P2NP).

$$\forall \pi, \mathtt{f}_1, \ldots, \mathtt{f}_n, \iota, \mathcal{B}.$$
$$\mathit{ProgEtr}(\textbf{let } (\pi, \iota) \textbf{ in } \mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n, \mathcal{B}) \Longleftrightarrow \mathit{NPProgEtr}(\textbf{let } (\pi, \iota) \textbf{ in } \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n, \mathcal{B})$$

Lemma. 6.1 shows the step ⑦ in Fig. 2. We show the details of the proof of Lemma. 6.1 in Appendix. B.

We give the intuition in the semantics equivalence proof. The intuition is that the write step, whose execution will generate a new message, in promising semantics can be divided into a promise step and a step to fulfill such promise.



Consider the following program.

```
x_na := 1;          ||  r1 := x_na;
r2 := y_rlx;        ||  if(r1) {
if(r2) {            ||      y_rel := 1;
    r3 := x_na;     ||      x_na := 2;
    print(r2);      ||      while(true);
}                   ||  }
```

Consider the following execution of the above program.

	t_1	t_2
(P-1)	$x_{na} := 1$	
(P-2)		$r_1 := x_{na}$ //1 $y_{rel} := 1$
(P-3)	$r_2 := y_{rlx}$ //1	
(P-4)		$x_{na} := 2$
(P-5)	$r_3 := x_{na}$ //2 print(r_3)	
(P-6)		while(true)

We can construct the execution under the non-preemptive semantics from the above execution. In construction, we need to reorder the step (P-1) and (P-2). It is impossible to reorder them directly, since the execution of "$r_1 := x_{na}$" in (P-2) needs to read the message generated by "$x_{na} := 1$" in (P-1). However, we can let "$x_{na} := 1$" generate the message valued 1 by promise step as the following shown.

$$\frac{\mathcal{TP}(\mathrm{t}) = (\sigma, \mathcal{V}, P) \quad \sigma \xrightarrow{\mathrm{W(na,x,\_)}} \_ \quad \langle \mathrm{x} : v@(\_, t], \_\rangle \in (M \backslash P) \quad \mathcal{V}.\mathrm{cur}.T_{\mathrm{rlx}}(x) < t}{(\mathcal{TP}, \mathrm{t}, \mathcal{S}, M, \beta)^{\iota} :\Longmapsto \text{ww-Race}}$$

$$\frac{\hat{\mathbb{P}} :\overset{load}{\Longmapsto} \hat{W} \quad \hat{W} :\Longmapsto^{*} \hat{W}' \quad \hat{W}' :\Longmapsto \text{ww-Race}}{\hat{\mathbb{P}} :\Longmapsto \text{ww-Race}}$$

$$\text{ww-NPRF}(\hat{\mathbb{P}}) \quad ::= \quad \neg(\hat{\mathbb{P}} :\Longmapsto \text{ww-Race})$$

Fig. 16. Write-write race freedom under the non-preemptive semantics

	t_1	t_2
(NP-1)	promise $\langle \mathrm{x} : 1@(0, 1], V_{\perp}\rangle$	
(NP-2)		$r_1 := \mathrm{x}_{\mathrm{na}}$ //1 $\mathrm{y}_{\mathrm{rel}} := 1$
(NP-3)	$\mathrm{x}_{\mathrm{na}} := 1$ *(* fulfill *)* $r_2 := \mathrm{y}_{\mathrm{rlx}}$ //1	
(NP-4)		promise $\langle \mathrm{x} : 2@(1, 2], V_{\perp}\rangle$
(NP-5)	$r_3 := \mathrm{x}_{\mathrm{na}}$ //2 `print`(r_3)	
(NP-5)		$\mathrm{x}_{\mathrm{na}} := 2$ *(* fulfill *)* `while(true)`

Data race under the non-preemptive semantics. We define the data race under the non-preemptive semantics in Fig. 16. We need to prove the equvialence between the data race freedoms under promising semantics and the non-preemptive semantics as shown in Lemma. 6.2.

Lemma 6.2 (ww-race free equvialence - P2NP).

$$\forall \pi, \mathrm{f}_1, \ldots, \mathrm{f}_n, \iota.$$
$$\text{ww-RF}(\textbf{let } (\pi, \iota) \textbf{ in } \mathrm{f}_1 \parallel \cdots \parallel \mathrm{f}_n) \Longleftrightarrow \text{ww-NPRF}(\textbf{let } (\pi, \iota) \textbf{ in } \mathrm{f}_1 \mid \ldots \mid \mathrm{f}_n)$$

Lemma. 6.2 shows the step ① and the step ⑥ in Fig. 2.

$$
\begin{array}{rcl}
\textit{(MMap)}\quad \varphi & \in & (\mathit{Var} \times \mathit{Time}) \rightharpoonup \mathit{Time} \\
\mathsf{mon}(\varphi) & \triangleq & \forall \mathsf{x}, t_1, t_2, t_1', t_2'.\ (\varphi(\mathsf{x}, t_1) = t_1' \wedge \varphi(\mathsf{x}, t_2) = t_2' \wedge t_1 < t_2) \implies t_1' < t_2' \\
\varphi(M) & \triangleq & \{(\mathsf{x}, t') \mid \langle \mathsf{x} : \_@(\_, t], \_\rangle \in M \wedge \varphi(\mathsf{x}, t) = t'\} \\
\lfloor\!\lfloor M \rfloor\!\rfloor & \triangleq & \{(\mathsf{x}, t) \mid \langle \mathsf{x} : \_@(\_, t], \_\rangle \in M\} \\
\varphi(M_t, M_s) & \triangleq & \varphi(M_t) \subseteq \lfloor\!\lfloor M_s \rfloor\!\rfloor \wedge \mathsf{dom}(\varphi) = \lfloor\!\lfloor M_t \rfloor\!\rfloor \wedge \mathsf{mon}(\varphi)
\end{array}
$$

Fig. 17. φ-related messages

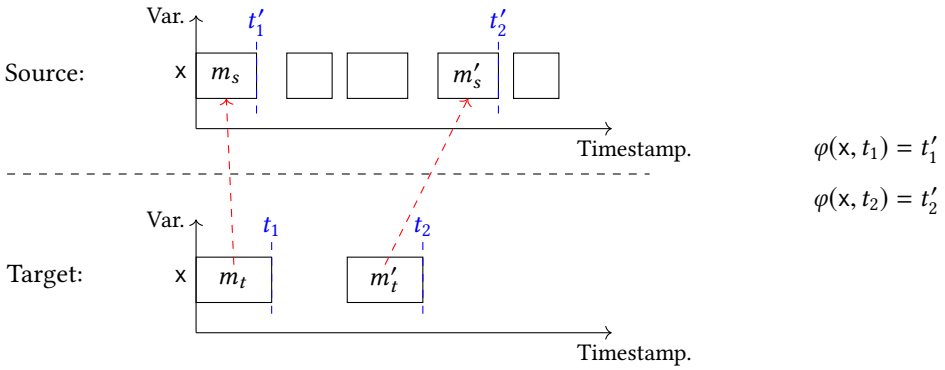


Fig. 18. An example of message injection

7 THREAD LOCAL UPWARD SIMULATION

In this section, we define a *thread-local* simulation as the formal correctness definition of optimizations. Before presenting the simulation, we first introduce several relations that relate (part of) the program configurations at the target and the source levels.

Timestamp mapping. In PS2.1, the memory is defined as a set of messages. We introduce a partial mapping φ whose type is defined in Fig. 17 to relate the "to"-timestamps of the messages in the target and source levels as shown in Fig. 18. We use $\mathsf{mon}(\varphi)$ to represent that φ is monotonic.

Invariant parameter and rely conditions. Since we allow the simulation to be established for individual threads without relying on the code of other threads, we use the invariant I and R condition to depict the behaviors of other threads for thread-local reasoning and compositionality. Our simulation is parameterized with an invariant I (in Fig. 19), which needs to hold over the shared states at *every switch point*. It can be instantiated differently when verifying different optimizations. We use the quadruple $\mathbb{S} = (\mathcal{S}_t, M_t, \mathcal{S}_s, M_s)$ to represent the global time map ($\mathcal{S}$) for SC fences and the memory (M) at the source and target levels. Users instantiating $I(\iota, \varphi, \mathbb{S})$ are expected to specify the application-dependent invariant over φ and $\mathbb{S}$ with the help of ι of the atomic variables set at the switch point. An invariant I is well-formed ($\mathsf{wf}(I)$, defined in Fig. 19), if each concrete message in the target level has a related one in the source level through a monotonic timestamp mapping φ (shown as $\varphi(M_t, M_s)$, defined in Fig. 17).

The rely condition (R) defined in Fig. 19 specifies the environment's transitions at the source and the target levels. The parameters $\mathbb{S}$ and $\mathbb{S}'$ represent the shared states at the points when the current thread switches out and back, respectively. The first item in R are guaranteed by our non-preemptive semantics as well as PS2.1

$$
\begin{array}{rcl}
\textit{(Sst)} \quad \mathbb{S} & \triangleq & (\mathcal{S}_t, M_t, \mathcal{S}_s, M_s) \\
\textit{(Inv)} \quad I & \in & \mathit{Atms} \to \mathit{MMap} \to \mathit{Sst} \to \mathit{Prop} \\
\\
T \in M & \triangleq & \forall \mathsf{x} \in \mathit{Var}.\ \exists m \in \widetilde{M}(\mathsf{x}).\ T(\mathsf{x}) = m.\mathsf{to} \\
V \in M & \triangleq & V.T_{\mathsf{na}} \in M \wedge V.T_{\mathsf{rlx}} \in M \\
\mathsf{closed}(M) & \triangleq & \forall m \in \widetilde{M}.\ m.\mathsf{view} \in M \\
[M]_\iota & \triangleq & \{m \mid m \in M \wedge m.\mathsf{var} \in \iota\} \\
M \approx M' & \triangleq & (\forall \mathsf{x}, f, t, v.\ \langle \mathsf{x} : v@(f,t], \_\rangle \in M \iff \langle \mathsf{x} : v@(f,t], \_\rangle \in M') \\
 & & \wedge\ (\forall \mathsf{x}, f, t.\ \langle \mathsf{x} : (f,t]\rangle \in M \iff \langle \mathsf{x} : (f,t]\rangle \in M') \\
T \le T' & \triangleq & \forall \mathsf{x} \in \mathit{Var}.\ T(\mathsf{x}) \le T'(\mathsf{x}) \\
\\
\mathsf{wf}(I) & \triangleq & \forall \iota, \varphi, \mathbb{S}.\ I(\iota, \varphi, \mathbb{S}) \implies \varphi(\mathbb{S}.M_t, \mathbb{S}.M_s) \\
\\
\mathsf{env}(\mathbb{S}, \mathbb{S}', P_t, P_s) & \triangleq & \\
\multicolumn{3}{l}{\quad \widetilde{M_t} \subseteq \widetilde{M_t'} \wedge \widetilde{M_s} \subseteq \widetilde{M_s'} \wedge \mathcal{S}_t \le \mathcal{S}_t' \wedge \mathcal{S}_s \le \mathcal{S}_s' \wedge} \\
\multicolumn{3}{l}{\quad \mathsf{closed}(M_t') \wedge \mathcal{S}_t' \in M_t' \wedge P_t \subseteq M_t' \wedge P_s \subseteq M_s'} \\
\multicolumn{3}{l}{\quad \textit{where}\ \ \mathbb{S} = (\mathcal{S}_t, M_t, \mathcal{S}_s, M_s)\ \textit{and}\ \mathbb{S}' = (\mathcal{S}_t', M_t', \mathcal{S}_s', M_s')} \\
\\
\mathsf{R}(\iota, (\varphi, \mathbb{S}), (\varphi', \mathbb{S}'), P_t, P_s) & \triangleq & \mathsf{env}(\mathbb{S}, \mathbb{S}', P_t, P_s) \wedge \varphi \subseteq \varphi' \wedge [\mathbb{S}'.M_t]_\iota \approx [\mathbb{S}'.M_s]_\iota
\end{array}
$$

Fig. 19. Invariant parameter and rely condition

$$
\begin{array}{rcl}
\textit{(Index)} \quad i & \in & \ldots \qquad \textit{(DlyItem)} \quad d \in (\mathit{Var} \times \mathit{Time}) \\
\textit{(Dlyset)} \quad \mathcal{D} & \in & \mathit{DlyItem} \rightharpoonup \mathit{Index} \\
\\
\mathcal{D}' < \mathcal{D} & \triangleq & \mathrm{dom}(\mathcal{D}) = \mathrm{dom}(\mathcal{D}') \wedge \forall \mathcal{D} \in \mathrm{dom}(\mathcal{D}).\ \mathcal{D}'(d) < \mathcal{D}(d) \\
\varphi(\mathcal{D}) & \triangleq & \{(\mathsf{x}, t') \mid \mathcal{D}(\mathsf{x}, t) = i \wedge \varphi(\mathsf{x}, t) = t'\} \\
\mathcal{D}[d \mapsto i] & \triangleq & \begin{cases} \mathcal{D} & \textit{if}\ d \in \mathrm{dom}(\mathcal{D}) \\ \mathcal{D}\{d \rightsquigarrow i\} & \textit{otherwise} \end{cases}
\end{array}
$$

Fig. 20. Delayed write set and corresponding definitions

(shown as env defined in Fig. 19, $\widetilde{M}$ consisting of only the concrete messages in M (see Sec. 3)). The second item indicates the increasing of the timestamp mapping, since the environment may insert additional messages. Since we do not perform optimizations on atomic accesses, the sub-memory for atomic variables in the target and source levels must be strictly equal, except for the message views (shown as $[\mathbb{S}'.M_t]_\iota \approx [\mathbb{S}'.M_s]_\iota$ in the item 3).

Delayed write set and step invariant. In our simulation, we allow the source state to be temporarily "left behind" the target. That is, when the target thread takes a write step, the source may not perform the corresponding step at present, but it must eventually do the step. We introduce the delayed write set $\mathcal{D}$ to record the set of writes which must be caught up by the source thread later. A pair of location x and timestamp t (called delayed item) in $\mathcal{D}$ means that the target thread has performed a write on x at the timestamp t, but such write has not been caught up by the source thread. $\mathcal{D}$ maps d to a well-founded index i to require the source thread to catch up a

$$\frac{(\mathsf{x},t)\in[\![(P-P')\cup(M'-M)]\!] \quad \mathcal{D}'=\mathcal{D}[(\mathsf{x},t)\mapsto i]}{(P,M),(P',M')\vdash\mathcal{D}\overset{\mathsf{W}(\mathsf{na},\mathsf{x},v)}{\rightsquigarrow}\mathcal{D}'} \qquad \frac{te\neq\mathsf{W}(\mathsf{na},\_,\_)}{(P,M),(P',M')\vdash\mathcal{D}\overset{te}{\rightsquigarrow}\mathcal{D}}$$

Fig. 21. Delayed item introduction rules

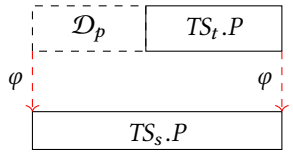
delayed write within finite steps. We also define some operations about $\mathcal{D}$ in Fig. 20, which will be used in our thread-local simulation.

The step invariant SI relates the target and source thread states and holds in *every step* of the thread-local simulation.

Definition 7.1 (Step invariant). $\mathsf{SI}(\iota,\varphi,(TS_t,M_t),(TS_s,M_s),\mathcal{D})$ iff,

(1) for any $(\mathsf{x},t)\in\mathrm{dom}(\varphi)$, if $TS_t.\mathcal{V}.\mathrm{cur}.T_{\mathsf{rlx}}(\mathsf{x})<t$, then $TS_s.\mathcal{V}.\mathrm{cur}.T_{\mathsf{rlx}}(\mathsf{x})<\varphi(\mathsf{x},t)$;
(2) there exists $\mathcal{D}_p\subseteq\mathcal{D}$, such that $(\varphi(\mathcal{D}_p)\cup\varphi(TS_t.P))=[\![TS_s.P]\!]$;
(3) for any σ', if $(TS_t.\sigma\xrightarrow{at}\sigma')$, then $\mathcal{D}=\emptyset$;
(4) $[M_t]_\iota\approx[M_s]_\iota$.

Item 1 of SI requires that if a target level message has not been observed by the target thread, its φ-related message in the source level should not be observed by the source thread either. Item 2 requires a one-to-one mapping between the source and target threads' promises as the following shown (including those in $\mathcal{D}$, which have been fulfilled by the target thread but not yet by the source).



Item 3 says that if the current instruction at the target is an atomic memory access ($TS_t.\sigma\xrightarrow{at}\sigma'$), all locations written by the target thread have also been written by the source ($\mathcal{D}=\emptyset$). Item 4 gives the same restriction as in R.

Simulation. We define the thread-local simulation between the target and source programs π_t and π_s in Def. 7.2. We use $M_0=\{\langle\mathsf{x}:0@(0,0],V_\perp\rangle\mid\mathsf{x}\in Var\}$, $\mathcal{S}_\perp=\{\mathsf{x}\rightsquigarrow 0\mid\mathsf{x}\in Var\}$ and $\varphi_0=\{(\mathsf{x},0)\rightsquigarrow 0\mid\mathsf{x}\in Var\}$ to represent the memory, the global time map for SC fence and the message mapping in the initial state respectively.

Definition 7.2 (Thread-local upward simulation). $I,\iota\models\pi_t\preccurlyeq\pi_s$ iff,

(1) $I(\iota,\varphi_0,(\mathcal{S}_\perp,M_0,\mathcal{S}_\perp,M_0))$ and $\mathsf{wf}(I)$;
(2) for any σ_t and f, if $\mathsf{Init}(\pi_t,\mathsf{f})=\sigma_t$, then there exists σ_s such that $\mathsf{Init}(\pi_s,\mathsf{f})=\sigma_s$ and

$$I,\iota\models((\sigma_t,\mathcal{V}_\perp,\emptyset),\mathcal{S}_\perp,M_0)\preccurlyeq^{\circ,\emptyset}_{\varphi_0}((\sigma_s,\mathcal{V}_\perp,\emptyset),\mathcal{S}_\perp,M_0),$$

where $I,\iota\models(TS_t,\mathcal{S}_t,M_t)\preccurlyeq^{\beta,\mathcal{D}}_{\varphi}(TS_s,\mathcal{S}_s,M_s)$ is defined in Def. 7.3.

Def. 7.2 first requires that the user-provided invariant I should hold at initial states and I should be well-formed. Second, if the execution of a target thread starts from the function f in π_t in the initial state, the source thread can also start from f in π_s and they have the simulation defined in Def. 7.3.

$$\frac{\iota \vdash (TS, \mathcal{S}, M) \xrightarrow{\mathsf{R}(\mathsf{na},\mathsf{x},v)} (TS', \mathcal{S}', M') \quad \forall (\mathsf{x}, t) \in \mathrm{dom}(\mathcal{D}).\ TS.\mathcal{V}.\mathrm{cur}.T_{\mathsf{rlx}}(\mathsf{x}) = TS'.\mathcal{V}.\mathrm{cur}.T_{\mathsf{rlx}}(\mathsf{x})}{\iota \vdash (TS, \mathcal{S}, M, \mathcal{D}) \xrightarrow{\mathsf{R}(\mathsf{na},\mathsf{x},v)} (TS', \mathcal{S}', M', \mathcal{D})}$$

$$\frac{\iota \vdash (TS, \mathcal{S}, M) \xrightarrow{\mathsf{W}(\mathsf{na},\mathsf{x},v)} (TS', \mathcal{S}', M') \quad \mathcal{D}' = \mathcal{D}\backslash(\mathsf{x}, t) \vee \mathcal{D}' = \mathcal{D}}{\iota \vdash (TS, \mathcal{S}, M, \mathcal{D}) \xrightarrow{\mathsf{W}(\mathsf{na},\mathsf{x},v)} (TS', \mathcal{S}', M', \mathcal{D}')} \qquad \frac{\iota \vdash (TS, \mathcal{S}, M) \xrightarrow{\tau} (TS', \mathcal{S}', M')}{\iota \vdash (TS, \mathcal{S}, M, \mathcal{D}) \xrightarrow{\tau} (TS', \mathcal{S}', M', \mathcal{D})}$$

Fig. 22. Delayed item elimination rules

Definition 7.3. $I, \iota \models (TS_t, \mathcal{S}_t, M_t) \preccurlyeq_{\varphi}^{\beta,\mathcal{D}} (TS_s, \mathcal{S}_s, M_s)$ is the largest relation such that:
whenever $I, \iota \models (TS_t, \mathcal{S}_t, M_t) \preccurlyeq_{\varphi}^{\beta,\mathcal{D}} (TS_s, \mathcal{S}_s, M_s)$, then
either $(\iota \vdash (TS_s, \mathcal{S}_s, M_s, \beta) \longmapsto^{+} \textbf{abort})$,
or $\mathsf{SI}(\iota, \varphi, (TS_t, M_t), (TS_s, M_s), \mathcal{D})$ and the following are true:

(1) $\forall TS_t', \mathcal{S}_t', M_t', te,$ if $\iota \vdash (TS_t, \mathcal{S}_t, M_t) \xrightarrow{te} (TS_t', \mathcal{S}_t', M_t')$, then the following hold:
  (a) if $te \in AT$, there exist $TS_s', \mathcal{S}_s', M_s'$, and φ' such that:
    - $\iota \vdash (TS_s, \mathcal{S}_s, M_s) \xrightarrow{na}{}^{*} \xrightarrow{te} (TS_s', \mathcal{S}_s', M_s')$;
    - $\varphi \subseteq \varphi'$ and $I, \iota \models (TS_t', \mathcal{S}_t', M_t') \preccurlyeq_{\varphi'}^{\circ,\emptyset} (TS_s', \mathcal{S}_s', M_s')$;
  (b) if $te \in NA$, there exist $TS_s', \mathcal{S}_s', M_s', \mathcal{D}_1$ and $\mathcal{D}_2$, such that:
    - $(TS_t.P, M_t), (TS_t'.P, M_t') \vdash \mathcal{D} \overset{te}{\rightsquigarrow} \mathcal{D}_1$;
    - $\iota \vdash (TS_s, \mathcal{S}_s, M_s, \mathcal{D}_1) \xrightarrow{na}{}^{*} (TS_s', \mathcal{S}_s', M_s', \mathcal{D}_2)$ and $\mathcal{D}_2' < \mathcal{D}_2$;
    - $\exists \mathcal{D}_2' < \mathcal{D}_2.\ I, \iota \models (TS_t', \mathcal{S}_t', M_t') \preccurlyeq_{\varphi}^{\bullet,\mathcal{D}_2'} (TS_s', \mathcal{S}_s', M_s')$;
  (c) if $te \in \{\mathsf{prm}, \mathsf{rsv}\}$ and $\beta = \circ$, there exist $TS_s', \mathcal{S}_s', M_s'$ and φ' such that:
    - $\iota \vdash (TS_s, \mathcal{S}_s, M_s) \xrightarrow{prc}{}^{*} (TS_s', \mathcal{S}_s', M_s')$;
    - $\varphi \subseteq \varphi'$ and $I, \iota \models (TS_t', \mathcal{S}_t', M_t') \preccurlyeq_{\varphi'}^{\circ,\emptyset} (TS_s', \mathcal{S}_s', M_s')$;
  (d) if $te = \mathsf{ccl}$, there exist $TS_s', \mathcal{S}_s', M_s'$ such that:
    - $\iota \vdash (TS_s, \mathcal{S}_s, M_s) \xrightarrow{\mathsf{ccl}}{}^{*} (TS_s', \mathcal{S}_s', M_s')$;
    - $I, \iota \models (TS_t', \mathcal{S}_t', M_t') \preccurlyeq_{\varphi}^{\beta,\mathcal{D}} (TS_s', \mathcal{S}_s', M_s')$;
(2) if $\beta = \circ$, then $I(\iota, \varphi, \mathbb{S})$ and $\forall \mathbb{S}', \varphi'$,
  if $\mathsf{R}(\iota, (\varphi, \mathbb{S}), (\varphi', \mathbb{S}'), TS_t.P, TS_s.P)$ and $I(\iota, \varphi', \mathbb{S}')$, then
  $I, \iota \models (TS_t, \mathcal{S}_t', M_t') \preccurlyeq_{\varphi'}^{\circ,\emptyset} (TS_s, \mathcal{S}_s', M_s')$, where $\mathbb{S} = (\mathcal{S}_t, M_t, \mathcal{S}_s, M_s)$, $\mathbb{S}' = (\mathcal{S}_t', M_t', \mathcal{S}_s', M_s')$;
(3) if $\iota \vdash (TS_t, \mathcal{S}_t, M_t) \longrightarrow \textbf{done}$, there exist $TS_s', \mathcal{S}_s', M_s'$, and φ' such that:
  - $\iota \vdash (TS_s, \mathcal{S}_s, M_s, \mathcal{D}) \xrightarrow{na}{}^{*} (TS_s', \mathcal{S}_s', M_s', \emptyset)$, $\iota \vdash (TS_s', \mathcal{S}_s', M_s') \longrightarrow \textbf{done}$;
  - $\varphi \subseteq \varphi'$ and $I(\iota, \varphi', (\mathcal{S}_t, M_t, \mathcal{S}_s', M_s'))$;
(4) if $\iota \vdash (TS_t, \mathcal{S}_t, M_t) \longrightarrow \textbf{abort}$, then $\iota \vdash (TS_s, \mathcal{S}_s, M_s, \beta) \longmapsto^{+} \textbf{abort}$.

The simulation $I, \iota \models (TS_t, \mathcal{S}_t, M_t) \preccurlyeq_{\varphi}^{\beta,\mathcal{D}} (TS_s, \mathcal{S}_s, M_s)$ carries a delayed write set $\mathcal{D}$ for the writes that the source thread has to perform later. The parameter φ records the timestamp mapping between the target and source memory at the last switch point. The atomic bit β indicates whether the thread can switch.

If the source thread aborts in finite steps (shown as $\iota \vdash (TS_s, \mathcal{S}_s, M_s, \beta) \longmapsto^{+} \textbf{abort}$), the simulation trivially holds. So the correctness of the optimization is meaningful only when the source program never aborts.

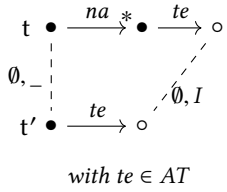
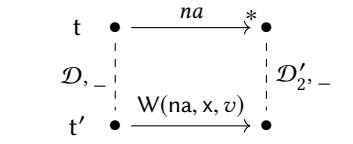
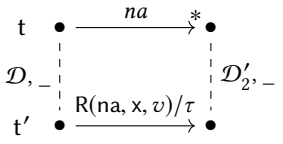


Fig. 23. Simulation diagram of atomic step (t and t′ are target and source threads respectively)



(a) Non-atomic write step case



(b) Non-atomic read step case

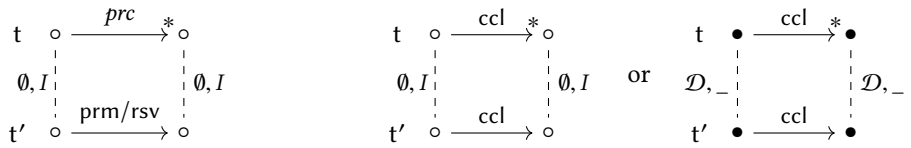
Fig. 24. Simulation diagrams of non-atomic step (t′ and t are target and source threads respectively, and $\mathcal{D}_2$ records the delayed writes that have not been caught up by the current source thread steps)

Otherwise, we require that the step invariant SI always holds, and discuss the different cases of target steps.

Case (1-a) in Def. 7.3 shows an atomic step of the target. The simulation follows the diagram in Fig. 23. The step invariant SI ensures that the delayed write set has been empty when taking an atomic step. After the step, the timestamp mapping increasing (shown as $\varphi \subseteq \varphi'$) and the invariant I needs to be reestablished, since the target and source threads reach a switch point.

Case (1-b) shows the condition that the target thread takes a non-atomic step. If the target thread takes a non-atomic write step at the timestamp t as shown in Fig. 24(a), we add a new delayed item (x, t) with an index i into the delayed write set $\mathcal{D}$ as defined in Fig. 22. The target write step may fulfill a message in its promise set ($(\mathsf{x}, t) \in [\![TS_t.P - TS'_t.P]\!]$), or generate a new message into memory ($(\mathsf{x}, t) \in [\![M'_t - M_t]\!]$). Then, the source thread will take some non-atomic steps to catch up some delayed writes according to the rules in Fig. 22. Here, we forbid the non-atomic reads in the source thread steps to read unobserved messages, whose locations are recorded in the delayed write set. Reading such unobserved message on some location x may cause the source thread to insert the message on location x at some unexpected timestamps when catching up the delayed write on x, and may cause the write-write race freedom to fail to preserve. Then, we reduce the indexes of the elements in the delayed write set (shown as $\mathcal{D}'_2 < \mathcal{D}_2$) to ensure that the source thread will eventually write to the locations in the delay set.

Case (1-c) and (1-d) in Def. 7.3 reflect the situations that the target thread takes promise/reserve step and cancel step (shown in Fig. 25). As we have introduced, a thread takes a promise step for its future write. Since we require that the locations written by the source thread should not be less than the target thread for write-write race freedom preservation, each memory write at the target level should have a corresponding memory write at the source level. Thus, if the target thread takes a promise step for a future write, the simulation requires the source thread to take a promise step for the corresponding future write in the source program. In the cancel step case (shown in Fig. 25(b)), the indexes of elements in the delayed write set do not need to be reduced, even if the

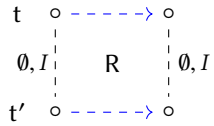


(a) Promise/reserve step case (b) Cancel step case

Fig. 25. Simulation diagrams of promise/reserve and cancel steps (t′ and t are target and source threads respectively)

target and source threads are in atomic block, since the number of the reservations in the thread's promise set is finite and the thread will never take cancel steps forever.

If the atomic bit is ∘ (case 2), we consider the interaction with other threads (following the diagram below). When switching back after environment steps satisfying R and the invariant I reestablished, the simulation must hold over the new states.



Finally, case 3 (or 4) says, if the target thread terminates (or aborts), so does the source thread. When both target and source threads terminate, we require that the invariant I holds and a larger message mapping φ' is well-formed from the restrictions of guarantee condition (G).

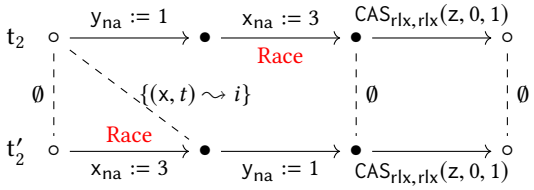
We show that why the item 3 in the step invariant defined in Def. 7.1 is essential in proving the write-write race preserving. Consider the following example. For the source program, we call the left thread t_1 and the right thread t_2. For the target program, we call the left thread t_1' and the right thread t_2'.

$$\mathsf{x_{na}} := 1; \;\Big\|\; \begin{array}{l}\mathsf{y_{na}} := 1;\\ \mathsf{x_{na}} := 3;\\ \mathsf{CAS_{rlx,rlx}}(z, 0, 1);\end{array} \quad \rightsquigarrow \quad \mathsf{x_{na}} := 1 \;\Big\|\; \begin{array}{l}\mathsf{x_{na}} := 3;\\ \mathsf{y_{na}} := 1;\\ \mathsf{CAS_{rlx,rlx}}(z, 0, 1);\end{array}$$

The execution of the target program may have the following execution to generate the write-write race.

	t_1'	t_2'
(1)	reserve $\langle z : (0, 1]\rangle$	
(2)	$\mathsf{x_{na}} := 1$	
(3)		$\mathsf{x_{na}} := 3$ //Race

Consider that we have already established the thread-local simulation between t_2' and t_2 as the following shown and we need to use such simulation to construct the write-write race in the source program.



To ensure that the source thread will eventually write the location x and generates write-write race, our thread-local simulation needs to make sure that when executing $\mathsf{CAS_{rlx,rlx}}(z, 0, 1)$ the delayed write set is empty (the item

3 in the step invariant). Note that we can not write such restriction in the atomic step case in the thread-local simulation, since the execution of $\mathsf{CAS}_{\mathsf{rlx},\mathsf{rlx}}(z, 0, 1)$ can not be done. The reason is that the timestamps that will be used by the execution of $\mathsf{CAS}_{\mathsf{rlx},\mathsf{rlx}}(z, 0, 1)$ have been reserved by t'_1.

8 WHOLE PROGRAM SIMULATION AND COMPOSITIONALITY

In this section, we define the whole program simulation in Sec. 8.1, and then show the proof sketch of the compositionality in Sec. 8.2. In Sec. 8.3, we show that our thread-local simulation preserves the promise certification. and illustrate how we prove that a certification against the current memory for non-atomic locations ensures the existence of the certification against the capped memory in detail as we have mentioned in Sec. 1.

8.1 Whole program simulation

The role of the whole program simulation in our work, as shown in Fig. 2, is to ensure that there is a refinement relation between the target and source programs. We define the whole program simulation in the following.

Definition 8.1 (Whole program simulation). $\textbf{let}\ (\pi_t, \iota)\ \textbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n \leqslant \textbf{let}\ (\pi_s, \iota)\ \textbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n$ iff, for any $\hat{W}_t$, if $\textbf{let}\ (\pi_t, \iota)\ \textbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n :\overset{load}{\Longrightarrow} \hat{W}_t$, there exists $\hat{W}_s$ such that:

- $\textbf{let}\ (\pi_s, \iota)\ \textbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n :\overset{load}{\Longrightarrow} \hat{W}_s$;
- $\hat{W}_t \leqslant \hat{W}_s$.

where $\hat{W}_t \leqslant \hat{W}_s$ is defined in Def. 8.2.

Definition 8.2. Whenever $(\mathcal{TP}_t, \mathtt{t}, \mathcal{S}_t, M_t, \beta_t)^\iota \leqslant (\mathcal{TP}_s, \mathtt{t}, \mathcal{S}_s, M_s, \beta_s)^\iota$, the following are true:

(1) for any $\mathcal{TP}_t{}'$, $\mathcal{S}_t'$, M_t' and β_t', if $(\mathcal{TP}_t, \mathtt{t}, \mathcal{S}_t, M_t, \beta_t)^\iota :\overset{\tau}{\Longrightarrow} (\mathcal{TP}_t{}', \mathtt{t}, \mathcal{S}_t', M_t', \beta_t')^\iota$, then there exist $\mathcal{TP}_s{}'$, $\mathcal{S}_s'$, M_s' and β_s' such that:
   - $(\mathcal{TP}_s, \mathtt{t}, \mathcal{S}_s, M_s, \beta_s)^\iota :\overset{\tau}{\Longrightarrow}{}^* (\mathcal{TP}_s{}', \mathtt{t}, \mathcal{S}_s', M_s', \beta_s')^\iota$;
   - $(\mathcal{TP}_t{}', \mathtt{t}, \mathcal{S}_t', M_t', \beta_t')^\iota \leqslant (\mathcal{TP}_s{}', \mathtt{t}, \mathcal{S}_s', M_s', \beta_s')^\iota$.

(2) for any $\mathcal{TP}_t{}'$, $\mathcal{S}_t'$ and M_t', if $(\mathcal{TP}_t, \mathtt{t}, \mathcal{S}_t, M_t, \beta_t)^\iota :\overset{\mathrm{out}(v)}{\Longrightarrow} (\mathcal{TP}_t{}', \mathtt{t}, \mathcal{S}_t', M_t', \circ)^\iota$, then there exist $\mathcal{TP}_s{}'$, $\mathcal{S}_s'$, M_s', β_s', $\mathcal{TP}_s{}''$, $\mathcal{S}_s''$ and M_s'' such that:
   - $(\mathcal{TP}_s, \mathtt{t}, \mathcal{S}_s, M_s, \beta_s)^\iota :\overset{\tau}{\Longrightarrow}{}^* (\mathcal{TP}_s{}', \mathtt{t}, \mathcal{S}_s', M_s', \beta_s')^\iota$ and $(\mathcal{TP}_s{}', \mathtt{t}, \mathcal{S}_s', M_s', \beta_s')^\iota :\overset{\mathrm{out}(v)}{\Longrightarrow} (\mathcal{TP}_s{}'', \mathtt{t}, \mathcal{S}_s'', M_s'', \circ)^\iota$;
   - $(\mathcal{TP}_t{}', \mathtt{t}, \mathcal{S}_t', M_t', \circ)^\iota \leqslant (\mathcal{TP}_s{}'', \mathtt{t}, \mathcal{S}_s'', M_s'', \circ)^\iota$.

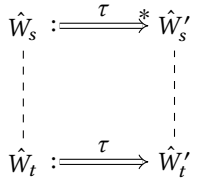
(3) for any $\mathcal{TP}_t{}'$, $\mathtt{t}'$, if $(\mathcal{TP}_t, \mathtt{t}, \mathcal{S}_t, M_t, \beta_t)^\iota :\overset{\mathrm{sw}}{\Longrightarrow} (\mathcal{TP}_t{}', \mathtt{t}', \mathcal{S}_t, M_t, \circ)^\iota$, then there exists $\mathcal{TP}_s{}'$ such that:
   - $(\mathcal{TP}_s, \mathtt{t}, \mathcal{S}_s, M_s, \beta_s)^\iota :\overset{\mathrm{sw}}{\Longrightarrow} (\mathcal{TP}_s{}', \mathtt{t}', \mathcal{S}_s, M_s, \circ)$;
   - $(\mathcal{TP}_t{}', \mathtt{t}', \mathcal{S}_t, M_t, \circ)^\iota \leqslant (\mathcal{TP}_s{}', \mathtt{t}', \mathcal{S}_s, M_s, \circ)^\iota$.

(4) if $(\mathcal{TP}_t, \mathtt{t}, \mathcal{S}_t, M_t, \beta_t)^\iota :\Longrightarrow \textbf{done}$, then there exist $\mathcal{TP}_s{}'$, $\mathcal{S}_s'$, M_s' and β_s' such that:
   - $(\mathcal{TP}_s, \mathtt{t}, \mathcal{S}_s, M_s, \beta_s)^\iota :\Longrightarrow{}^* (\mathcal{TP}_s{}', \mathtt{t}, \mathcal{S}_s', M_s', \beta_s')^\iota$ and $(\mathcal{TP}_s{}', \mathtt{t}, \mathcal{S}_s', M_s', \beta_s')^\iota :\Longrightarrow \textbf{done}$.

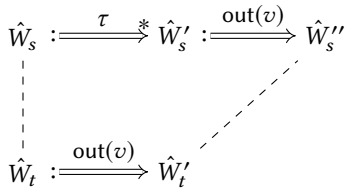
(5) if $(\mathcal{TP}_t, \mathtt{t}, \mathcal{S}_t, M_t, \beta_t)^\iota :\Longrightarrow \textbf{abort}$, then $(\mathcal{TP}_s, \mathtt{t}, \mathcal{S}_s, M_s, \beta_s)^\iota :\Longrightarrow \textbf{abort}$.

We write some figures to illustrate the whole program simulation defined in Def. 8.1 in the following.

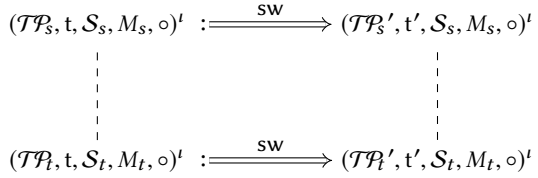
- If the target program takes a tau step, the source program is permitted to take multiple steps.



- If the target program takes an output step, the source program is restricted to generate the same output.



- If the target program does a thread switching, the source program will switch to the same thread.



8.2 Compositionality

We need to prove that the thread-local upward simulation can compose to the whole program simulation as shown in Def. 8.1.

Lemma 8.3 (Compositionality).

$$\begin{aligned}&\forall \pi_t, \pi_s, I, \iota, \mathtt{f}_1, \ldots, \mathtt{f}_n.\\ &\quad I, \iota \models \pi_t \preccurlyeq \pi_s \wedge\\ &\quad \mathsf{Safe}(\mathbf{let}\ (\pi_s, \iota)\ \mathbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n) \wedge\\ &\quad \text{ww-NPRF}(\mathbf{let}\ (\pi_s, \iota)\ \mathbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n)\\ &\Longrightarrow \mathbf{let}\ (\pi_t, \iota)\ \mathbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n \leqslant \mathbf{let}\ (\pi_s, \iota)\ \mathbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n\end{aligned}$$

Proof. From the premises, we have the following.

$$I, \iota \models \pi_t \preccurlyeq \pi_s \tag{1}$$

$$\mathsf{Safe}(\mathbf{let}\ (\pi_t, \iota)\ \mathbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n) \tag{2}$$

$$\text{ww-NPRF}(\mathbf{let}\ (\pi_t, \iota)\ \mathbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n) \tag{3}$$

We unfold (1) and have the following.

$$I(\iota, \varphi_0, (\mathcal{S}_\perp, M_0, \mathcal{S}_\perp, M_0)) \wedge \mathsf{wf}(I) \tag{4}$$

$$\begin{aligned}&\forall \sigma_t, \mathtt{f}.\ \mathsf{Init}(\pi_t, \mathtt{f}) = \sigma_t\\ &\quad \Longrightarrow \exists \sigma_s.(\mathsf{Init}(\pi_s, \mathtt{f}) = \sigma_s \wedge I, \iota \models ((\sigma_t, \mathcal{V}_\perp, \emptyset), \mathcal{S}_\perp, M_0) \preccurlyeq_\varphi^{\circ, \emptyset} ((\sigma_s, \mathcal{V}_\perp, \emptyset), \mathcal{S}_\perp, M_0))\end{aligned} \tag{5}$$

We unfold the proof goal. We have the following premise.

$$\mathbf{let}\ (\pi_t, \iota)\ \mathbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n :\overset{load}{\Longrightarrow} \hat{W}_t \tag{6}$$

And we need to prove that there exist $\hat{W}_s$ such that:

$$\mathbf{let}\ (\pi_s, \iota)\ \mathbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n :\overset{load}{\Longrightarrow} \hat{W}_s \tag{g1}$$

$$\hat{W}_t \leqslant \hat{W}_s \tag{g2}$$

Let $\hat{W}_t = (\mathcal{TP}_t, \mathrm{t}, \mathcal{S}_\perp, M_0, \circ)^\iota$. From (6) and (5), we have that there exist $\mathcal{TP}_s$ such that:

$$\mathbf{let}\ (\pi_s, \iota)\ \mathbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n :\overset{load}{\Longrightarrow} (\mathcal{TP}_s, \mathrm{t}, \mathcal{S}_\perp, M_0, \circ)^\iota \tag{7}$$

Thus, we finish the proof of (g1).

By applying Lemma. 8.5 on (2), we have the following.

$$\neg(\exists \hat{W}_s.\ (\mathcal{TP}_s, \mathsf{t}, \mathcal{S}_\perp, M_0, \circ)^\iota :\Longrightarrow^* \hat{W}_s \wedge \hat{W}_s :\Longrightarrow_{\mathsf{ax}} \textbf{abort}) \tag{8}$$

By applying Lemma. 8.8 on (3), we have the following.

$$\neg(\exists \hat{W}_s.\ (\mathcal{TP}_s, \mathsf{t}, \mathcal{S}_\perp, M_0, \circ)^\iota :\Longrightarrow^* \hat{W}_s \wedge \hat{W}_s :\Longmapsto_{\mathsf{ax}} \mathsf{ww\text{-}Race}) \tag{9}$$

By applying Lemma. 8.4 on (5), (8) and (9), we prove (g2). □

Lemma 8.4 (Compositionality - aux).

$$\begin{aligned}
&\forall \mathcal{TP}_t, , i, \mathcal{S}_t, M_t, \mathcal{TP}_s, \mathcal{S}_s, M_s, \beta, \beta_s, \iota, \mathcal{D}, \varphi, n.\\
&\quad (\forall j \in \{1, \ldots, n\} \backslash \{i\}.\ I, \iota \models (\mathcal{TP}_t(j), \mathcal{S}_t, M_t) \preccurlyeq_\varphi^{\circ, \emptyset} (\mathcal{TP}_s(j), \mathcal{S}_s, M_s)) \wedge\\
&\quad I, \iota \models (\mathcal{TP}_t(i), \mathcal{S}_t, M_t) \preccurlyeq_\varphi^{\beta, \mathcal{D}} (\mathcal{TP}_s(i), \mathcal{S}_s, M_s) \wedge\\
&\quad \neg(\exists \hat{W}_s.\ (\mathcal{TP}_s, i, \mathcal{S}_s, M_s, \beta_s)^\iota :\Longrightarrow^* \hat{W}_s \wedge \hat{W}_s :\Longrightarrow_{\mathsf{ax}} \textbf{abort}) \wedge\\
&\quad \neg(\exists \hat{W}_s.\ (\mathcal{TP}_s, i, \mathcal{S}_s, M_s, \beta_s)^\iota :\Longrightarrow^* \hat{W}_s \wedge \hat{W}_s :\Longmapsto_{\mathsf{ax}} \mathsf{ww\text{-}Race}) \wedge\\
&\quad (\beta = \circ \implies \beta_s = \circ) \wedge \mathsf{wf}(I)\\
&\implies (\mathcal{TP}_t, i, \mathcal{S}_t, M_t, \beta)^\iota \leqslant (\mathcal{TP}_s, i, \mathcal{S}_s, M_s, \beta_s)^\iota
\end{aligned}$$

Proof. By cofix. From the premises, we have the following.

$$\forall j \in \{1, \ldots, n\} \backslash \{i\}.\ I, \iota \models (\mathcal{TP}_t(j), \mathcal{S}_t, M_t) \preccurlyeq_\varphi^{\circ, \emptyset} (\mathcal{TP}_s(j), \mathcal{S}_s, M_s) \tag{1}$$

$$I, \iota \models (\mathcal{TP}_t(i), \mathcal{S}_t, M_t) \preccurlyeq_\varphi^{\beta, \mathcal{D}} (\mathcal{TP}_s(i), \mathcal{S}_s, M_s) \tag{2}$$

$$\neg(\exists \hat{W}_s.\ (\mathcal{TP}_s, i, \mathcal{S}_s, M_s, \beta_s)^\iota :\Longrightarrow^* \hat{W}_s \wedge \hat{W}_s :\Longrightarrow_{\mathsf{ax}} \textbf{abort}) \tag{3}$$

$$\neg(\exists \hat{W}_s.\ (\mathcal{TP}_s, i, \mathcal{S}_s, M_s, \beta_s)^\iota :\Longrightarrow^* \hat{W}_s \wedge \hat{W}_s :\Longmapsto_{\mathsf{ax}} \mathsf{ww\text{-}Race}) \tag{4}$$

$$(\beta = \circ \implies \beta_s = \circ) \wedge \mathsf{wf}(I) \tag{5}$$

We unfold the proof goal and we need to prove the following.

(1) If the current target thread takes a tau step, we have the following.

$$(\mathcal{TP}_t, i, \mathcal{S}_t, M_t, \beta)^\iota :\overset{\tau}{\Longrightarrow} (\mathcal{TP}_t', i, \mathcal{S}_t', M_t', \beta')^\iota \tag{6.1}$$

We unfold (6.1) and have that there exist TS_t', $\mathcal{S}_t'$, M_t' such that:

$$\iota \vdash (\mathcal{TP}_t(i), \mathcal{S}_t, M_t, \beta) \longmapsto^+ (TS_t', \mathcal{S}_t', M_t', \beta') \tag{6.11}$$

$$\mathcal{TP}_t' = \mathcal{TP}_t\{i \rightsquigarrow TS_t'\} \tag{6.12}$$

$$\mathsf{consistent}_{\mathsf{NP}}(TS_t', M_t', \beta', \iota) \tag{6.13}$$

We apply Lemma. 8.11 on (6.12) (2). We have that there exist TS_s', $\mathcal{S}_s'$, M_s' and β_s' such that:

$$\iota \vdash (\mathcal{TP}_s(i), \mathcal{S}_s, M_s, \beta_s) \longmapsto^* (TS_s', \mathcal{S}_s', M_s', \beta_s') \tag{6.14}$$

$$I, \iota \models (TS_t', \mathcal{S}_t', M_t') \preccurlyeq_{\varphi'}^{\beta', \mathcal{D}'} (TS_s', \mathcal{S}_s', M_s') \tag{6.15}$$

$$(\beta' = \circ \implies \beta_s' = \circ) \wedge \varphi \subseteq \varphi' \tag{6.16}$$

We discuss (6.14). If the current source thread takes zero step, we finish the proof of such case by co-inductive hypothesis. We focus on the case that the current source thread takes multiple steps. By applying Lemma. 8.13 on (6.13), (6.15), (3) and (4), we have the following.

$$\mathsf{consistent}_{\mathsf{NP}}(TS_s', M_s', \beta_s', \iota) \tag{6.17}$$

From (6.14), (6.17) and co-inductive hypothesis, we finish the proof.

$$\frac{\begin{array}{c}\iota \vdash (\mathcal{TP}(\mathrm{t}), \mathcal{S}, M) \longrightarrow^* (TS', \mathcal{S}', M') \\ TS'.\sigma \xrightarrow{\mathsf{W}(\mathsf{na},\mathsf{x},\_)} \_ \\ \langle \mathsf{x} : \_@(\_, t], \_\rangle \in (M' \backslash TS'.P) \qquad TS'.\mathcal{V}.\mathsf{cur}.T_{\mathsf{rlx}}(\mathsf{x}) < t \\ \iota \vdash (TS', \mathcal{S}', M') \longrightarrow^* ((\_, \_, \emptyset), \_, \_)\end{array}}{(\mathcal{TP}, \mathrm{t}, \mathcal{S}, M, \beta)^\iota :\Longrightarrow_{\mathsf{ax}} \mathsf{ww\text{-}Race}}$$

$$(\mathcal{TP}, \mathrm{t}, \mathcal{S}, M, \beta)^\iota :\Longrightarrow_{\mathsf{ax}} \mathbf{abort} \quad ::= \quad (\mathcal{TP}, \mathrm{t}, \mathcal{S}, M)^\iota \Longrightarrow \mathbf{abort}$$

Fig. 26. Auxiliary write-write race and abort step under non-preemptive semantics

(2) If the current target thread takes a done step, we have the following.

$$(\mathcal{TP}_t, \mathrm{t}, \mathcal{S}_t, M_t, \beta)^\iota :\xRightarrow{\mathsf{out}(v)} (\mathcal{TP}_t', \mathrm{t}, \mathcal{S}_t', M_t', \circ)^\iota \tag{6.2}$$

We unfold (6.1) and have that there exist TS_t', $\mathcal{S}_t'$ and M_t' such that:

$$\iota \vdash (\mathcal{TP}_t(i), \mathcal{S}_t, M_t, \beta) \xmapsto{\mathsf{out}(v)} (TS_t', \mathcal{S}_t', M_t', \beta') \tag{6.21}$$

$$\mathcal{TP}_t' = \mathcal{TP}_t\{i \rightsquigarrow TS_t'\} \tag{6.22}$$

$$\mathsf{consistent}_{\mathsf{NP}}(TS_t', M_t', \beta', \iota) \tag{6.23}$$

By applying Lemma. 8.12 on (6.21) and (2), we have that there exist TS_s', $\mathcal{S}_s'$, M_s' and φ' such that:

$$\iota \vdash (TS_s, \mathcal{S}_s, M_s, \beta) \longmapsto^* \xmapsto{\mathsf{out}(v)} (TS_s', \mathcal{S}_s', M_s', \circ) \tag{6.24}$$

$$I, \iota \models (TS_t', \mathcal{S}_t', M_t') \preccurlyeq^{\circ,\emptyset}_{\varphi'} (TS_s', \mathcal{S}_s', M_s') \tag{6.25}$$

$$\varphi \subseteq \varphi' \wedge I(\iota, \varphi', (\mathcal{S}_t', M_t', \mathcal{S}_s', M_s')) \tag{6.26}$$

From (6.24), (6.25), (6.26) and co-inductive hypothesis, we finish the proof.

(3) The correctness of the case that the target program takes a switch step is straight-forward.

(4) The correctness of the case that the target program takes a done step is straight-forward.

(5) If the current target thread takes an abort step, we have the following.

$$(\mathcal{TP}_t, \mathrm{t}, \mathcal{S}_t, M_t, \beta)^\iota :\Longrightarrow \mathbf{abort} \tag{6.5}$$

We unfold (6.5) and have the following.

$$\iota \vdash (\mathcal{TP}_t(\mathrm{t}), \mathcal{S}_t, M_t, \beta) \longmapsto^* (TS_t', \mathcal{S}_t', M_t', \beta') \tag{6.51}$$

$$\iota \vdash (TS_t', \mathcal{S}_t', M_t') \longrightarrow \mathbf{abort} \tag{6.52}$$

By applying Lemma. 8.11 on (6.51) and (2), We have that there exist TS_s', $\mathcal{S}_s'$, M_s' and β_s' such that:

$$\iota \vdash (\mathcal{TP}_s(i), \mathcal{S}_s, M_s, \beta_s) \longmapsto^* (TS_s', \mathcal{S}_s', M_s', \beta_s') \tag{6.53}$$

$$I, \iota \models (TS_t', \mathcal{S}_t', M_t') \preccurlyeq^{\beta',\mathcal{D}'}_{\varphi'} (TS_s', \mathcal{S}_s', M_s') \tag{6.54}$$

$$\beta' = \circ \implies \beta_s' = \circ \tag{6.55}$$

From (6.52) and (6.54), we construct an abort step of the source program.

□

Lemma 8.5 (Sound np-abort).

$$\begin{aligned}&\forall \mathcal{TP}, \mathrm{t}, \mathcal{S}, M, \iota.\\ &\quad \neg(\exists \hat{W}.\ (\mathcal{TP}, \mathrm{t}, \mathcal{S}, M, \circ)^{\iota} :\Longrightarrow^{*} \hat{W} \wedge \hat{W} :\Longrightarrow \mathbf{abort})\\ \Longrightarrow\ &\neg(\exists \hat{W}.\ (\mathcal{TP}, \mathrm{t}, \mathcal{S}, M, \circ)^{\iota} :\Longrightarrow^{*} \hat{W} \wedge \hat{W} :\Longrightarrow_{\mathsf{ax}} \mathbf{abort})\end{aligned}$$

Proof. We need to prove that the following.

$$\begin{aligned}&(\exists \hat{W}.\ (\mathcal{TP}, \mathrm{t}, \mathcal{S}, M, \circ)^{\iota} :\Longrightarrow^{*} \hat{W} \wedge \hat{W} :\Longrightarrow_{\mathsf{ax}} \mathbf{abort})\\ \Longrightarrow\ &(\exists \hat{W}.\ (\mathcal{TP}, \mathrm{t}, \mathcal{S}, M, \circ)^{\iota} :\Longrightarrow^{*} \hat{W} \wedge \hat{W} :\Longrightarrow \mathbf{abort})\end{aligned}$$

From the premise, we have that there exist $\hat{W}$ such that:

$$(\mathcal{TP}, \mathrm{t}, \mathcal{S}, M, \circ)^{\iota} :\Longrightarrow^{*} \hat{W} \tag{1}$$

$$\hat{W} :\Longrightarrow_{\mathsf{ax}} \mathbf{abort} \tag{2}$$

We have the following.

$$\begin{aligned}&((\mathcal{TP}, \mathrm{t}, \mathcal{S}, M, \circ)^{\iota} :\Longrightarrow_{\mathsf{ax}} \mathbf{abort}) \vee\\ &\quad \neg((\mathcal{TP}, \mathrm{t}, \mathcal{S}, M, \circ)^{\iota} :\Longrightarrow_{\mathsf{ax}} \mathbf{abort})\end{aligned} \tag{3}$$

We destruct (3) and discuss each case respectively.

- We first consider that the current thread will abort.

$$(\mathcal{TP}, \mathrm{t}, \mathcal{S}, M, \circ)^{\iota} :\Longrightarrow_{\mathsf{ax}} \mathbf{abort} \tag{3.1}$$

  We finish the proof of such case by applying Lemma. 8.6 on (3.1).
- Then, we consider that the current thread will not abort.

$$\neg((\mathcal{TP}, \mathrm{t}, \mathcal{S}, M, \circ)^{\iota} :\Longrightarrow_{\mathsf{ax}} \mathbf{abort}) \tag{3.2}$$

  By applying Lemma. 8.7 on (1), (2) and (3.2), we finish the proof.

□

Lemma 8.6 (Sound np-abort aux-1).

$$\begin{aligned}&\forall \mathcal{TP}, \mathrm{t}, \mathcal{S}, M, \iota.\\ &\quad (\mathcal{TP}, \mathrm{t}, \mathcal{S}, M, \circ)^{\iota} :\Longrightarrow_{\mathsf{ax}} \mathbf{abort}\\ \Longrightarrow\ &(\mathcal{TP}, \mathrm{t}, \mathcal{S}, M, \circ)^{\iota} :\Longrightarrow \mathbf{abort}\end{aligned}$$

Lemma 8.7 (Sound np-abort aux-2).

$$\begin{aligned}&\forall \hat{W}, \hat{W}_0, n.\\ &\quad \hat{W} :\Longrightarrow^{n} \hat{W}_0 \wedge \hat{W}_0 :\Longrightarrow_{\mathsf{ax}} \mathbf{abort} \wedge\\ &\quad \neg(\hat{W} :\Longrightarrow_{\mathsf{ax}} \mathbf{abort})\\ \Longrightarrow\ &\exists \hat{W}'.\ \hat{W} :\Longrightarrow^{*} \hat{W}' \wedge \hat{W}' :\Longrightarrow \mathbf{abort}\end{aligned}$$

Lemma 8.8 (Sound aux ww-np-race).

$$\begin{aligned}&\forall \mathcal{TP}, \mathrm{t}, \mathcal{S}, M, \iota.\\ &\quad \neg(\exists \hat{W}.\ (\mathcal{TP}, \mathrm{t}, \mathcal{S}, M, \circ)^{\iota} :\Longrightarrow^{*} \hat{W} \wedge \hat{W} :\Longmapsto \mathsf{ww\text{-}Race})\\ \Longrightarrow\ &\neg(\exists \hat{W}.\ (\mathcal{TP}, \mathrm{t}, \mathcal{S}, M, \circ)^{\iota} :\Longrightarrow^{*} \hat{W} \wedge \hat{W} :\Longmapsto_{\mathsf{ax}} \mathsf{ww\text{-}Race})\end{aligned}$$

Proof. From the premises, we have the following.

$$(\mathcal{TP}, \mathsf{t}, \mathcal{S}, M, \circ)^{\iota} :\Longrightarrow^{*} \hat{W} \tag{1}$$

$$\hat{W} :\models\!\Longrightarrow_{\mathsf{ax}} \text{ww-Race} \tag{2}$$

We need to prove the following.

$$\exists \hat{W}.\ (\mathcal{TP}, \mathsf{t}, \mathcal{S}, M, \circ)^{\iota} :\Longrightarrow^{*} \hat{W} \wedge \hat{W} :\models\!\Longrightarrow \text{ww-Race} \tag{g}$$

We have the following.

$$\begin{array}{l}((\mathcal{TP}, \mathsf{t}, \mathcal{S}, M, \circ)^{\iota} :\models\!\Longrightarrow_{\mathsf{ax}} \text{ww-Race}) \vee \\ \quad \neg((\mathcal{TP}, \mathsf{t}, \mathcal{S}, M, \circ)^{\iota} :\models\!\Longrightarrow_{\mathsf{ax}} \text{ww-Race})\end{array} \tag{3}$$

We destruct (3) and discuss each case respectively.

- We first consider that the current thread will generate data race.

$$(\mathcal{TP}, \mathsf{t}, \mathcal{S}, M, \circ)^{\iota} :\models\!\Longrightarrow_{\mathsf{ax}} \text{ww-Race} \tag{4.1}$$

  We apply Lemma. 8.9 on (4.1) and finish the proof of such case.
- Then, we consider that the current thread will not generate data race.

$$\neg((\mathcal{TP}, \mathsf{t}, \mathcal{S}, M, \circ)^{\iota} :\models\!\Longrightarrow_{\mathsf{ax}} \text{ww-Race}) \tag{5.2}$$

  We apply Lemma. 8.10 on (2), (3) and (5.2) and finish the proof of such case.

□

Lemma 8.9 (Sound aux ww-np-race aux-1).

$$\begin{array}{l}\forall \mathcal{TP}, \mathsf{t}, \mathcal{S}, M, \iota. \\ \quad (\mathcal{TP}, \mathsf{t}, \mathcal{S}, M, \circ)^{\iota} :\models\!\Longrightarrow_{\mathsf{ax}} \text{ww-Race} \\ \Longrightarrow\ \exists \hat{W}.\ (\mathcal{TP}, \mathsf{t}, \mathcal{S}, M, \circ)^{\iota} :\Longrightarrow^{*} \hat{W} \wedge \hat{W} :\models\!\Longrightarrow \text{ww-Race}\end{array}$$

Lemma 8.10 (Sound aux ww-np-race aux-2).

$$\begin{array}{l}\forall \hat{W}, \hat{W}_0, n. \\ \quad \hat{W} :\Longrightarrow^{n} \hat{W}_0 \wedge \hat{W}_0 :\models\!\Longrightarrow_{\mathsf{ax}} \text{ww-Race} \wedge \\ \quad \neg(\hat{W} :\models\!\Longrightarrow_{\mathsf{ax}} \text{ww-Race}) \\ \Longrightarrow\ \exists \hat{W}'.\ \hat{W} :\Longrightarrow^{*} \hat{W}' \wedge \hat{W}' :\models\!\Longrightarrow \text{ww-Race}\end{array}$$

Lemma 8.11 (Simulation: tau step).

$$\begin{array}{l}\forall TS_t, \mathcal{S}_t, M_t, TS'_t, \mathcal{S}'_t, M'_t, TS_s, \mathcal{S}_s, M_s, n, \beta, \beta_s, \beta', \mathcal{D}, \varphi. \\ \quad \iota \vdash (TS_t, \mathcal{S}_t, M_t, \beta) \longmapsto^{n} (TS'_t, \mathcal{S}'_t, M'_t, \beta') \wedge \\ \quad I, \iota \models (TS_t, \mathcal{S}_t, M_t) \preccurlyeq_{\varphi}^{\beta, \mathcal{D}} (TS_s, \mathcal{S}_s, M_s) \wedge (\beta = \circ \implies \beta_s = \circ) \\ \Longrightarrow\ \exists TS'_s, \mathcal{S}'_s, M'_s, \mathcal{D}', \varphi', \beta'_s. \\ \quad \iota \vdash (TS_s, \mathcal{S}_s, M_s, \beta_s) \longmapsto^{*} (TS'_s, \mathcal{S}'_s, M'_s, \beta'_s) \wedge \\ \quad I, \iota \models (TS'_t, \mathcal{S}'_t, M'_t) \preccurlyeq_{\varphi'}^{\beta', \mathcal{D}'} (TS'_s, \mathcal{S}'_s, M'_s) \wedge \varphi \subseteq \varphi' \wedge \\ \quad (\beta' = \circ \implies \beta'_s = \circ)\end{array}$$

Lemma 8.12 (Simulation: output step).

$$\begin{aligned}
&\forall TS_t, \mathcal{S}_t, M_t, TS'_t, \mathcal{S}'_t, M'_t, TS_s, \mathcal{S}_s, M_s, \beta, \beta_s, \mathcal{D}, \varphi.\\
&\quad \iota \vdash (TS_t, \mathcal{S}_t, M_t, \beta) \xmapsto{\mathrm{out}(v)} (TS'_t, \mathcal{S}'_t, M'_t, \circ) \wedge\\
&\quad I, \iota \models (TS_t, \mathcal{S}_t, M_t) \preccurlyeq_{\varphi}^{\beta, \mathcal{D}} (TS_s, \mathcal{S}_s, M_s) \wedge (\beta = \circ \implies \beta_s = \circ)\\
&\implies \exists TS'_s, \mathcal{S}'_s, M'_s, \varphi'.\\
&\quad \iota \vdash (TS_s, \mathcal{S}_s, M_s, \beta) \longmapsto^* \xmapsto{\mathrm{out}(v)} (TS'_s, \mathcal{S}'_s, M'_s, \circ) \wedge\\
&\quad I, \iota \models (TS'_t, \mathcal{S}'_t, M'_t) \preccurlyeq_{\varphi'}^{\circ, \emptyset} (TS'_s, \mathcal{S}'_s, M'_s) \wedge \varphi \subseteq \varphi'
\end{aligned}$$

8.3 Promise certification preservation

Theorem. 8.13 shows that under the assumption of the write-write race freedom, our thread-local simulation ensures the preserving of the promise certification. illustrate how we prove that a certification against the current memory for non-atomic locations ensures the existence of the certification against the capped memory in Lemma. 8.18. This lemma also shows why the locations in our work are divided into atomic locations and non-atomic locations. In the following introduction, the conditions, which say that the thread promises set is a subseteq of the memory and the thread view is closed, are omitted in the presentations of some lemmas, since these conditions are ensured by promising semantics and not the main points of our proof.

Theorem 8.13 (Promise consistency preserving).

$$\begin{aligned}
&\forall \mathcal{TP}_t, \mathrm{t}, M_t, \mathcal{TP}_s, M_s, \iota, \beta, \mathcal{D}, \varphi.\\
&\quad \mathsf{consistent}_{\mathsf{NP}}(\mathcal{TP}_t(\mathrm{t}), M_t, \beta, \iota) \wedge\\
&\quad I, \iota \models (\mathcal{TP}_t(\mathrm{t}), \mathcal{S}_t, M_t) \preccurlyeq_{\varphi}^{\beta, \mathcal{D}} (\mathcal{TP}_s(\mathrm{t}), \mathcal{S}_s, M_s) \wedge\\
&\quad \neg(\iota \vdash (\mathcal{TP}_s(\mathrm{t}), \mathcal{S}_s, M_s) \longrightarrow^* \textbf{abort}) \wedge\\
&\quad \neg((\mathcal{TP}_s, \mathrm{t}, \mathcal{S}_s, M_s, \beta)^{\iota} :\Longrightarrow_{\mathsf{ax}} \text{ww-Race})\\
&\implies \mathsf{consistent}_{\mathsf{NP}}(\mathcal{TP}_s(\mathrm{t}), M_s, \beta, \iota)
\end{aligned}$$

Proof. From the premises, we have that the following hold.

$$\mathsf{consistent}_{\mathsf{NP}}(\mathcal{TP}_t(\mathrm{t}), M_t, \beta, \iota) \tag{1}$$

$$I, \iota \models (\mathcal{TP}_t(\mathrm{t}), \mathcal{S}_t, M_t) \preccurlyeq_{\varphi}^{\beta, \mathcal{D}} (\mathcal{TP}_s(\mathrm{t}), \mathcal{S}_s, M_s) \tag{2}$$

$$\neg(\iota \vdash (\mathcal{TP}_t(\mathrm{t}), \mathcal{S}_t, M_t) \longrightarrow^* \textbf{abort}) \tag{3}$$

$$\neg((\mathcal{TP}_s, \mathrm{t}, \mathcal{S}_s, M_s, \beta)^{\iota} :\Longrightarrow_{\mathsf{ax}} \text{ww-Race}) \tag{4}$$

We unfold (1) and have the following:

$$\iota \vdash (\mathcal{TP}_t(\mathrm{t}), \widehat{T}(M_t), \widehat{M_t}, \beta) \longmapsto^* ((\_, \_, \emptyset), \_, \_, \_) \tag{5}$$

Let $M_{sc} = (\{m \in M_s \mid \iota(m.\mathrm{var}) = \mathrm{na}\} \cup \{m \in \widehat{M_s} \mid \iota(m.\mathrm{var}) = \mathtt{at}\})$. We apply Lemma. 8.14 on (5), (2) and (3) and have the following.

$$\iota \vdash (\mathcal{TP}_s(\mathrm{t}), \widehat{T}(M_s), M_{sc}, \beta) \longmapsto^* ((\_, \_, \emptyset), \_, \_, \_) \tag{6}$$

We apply Lemma. 8.17 on (4) and have the following.

$$\neg((\mathcal{TP}_s, \mathrm{t}, \widehat{T}(\widehat{M_s}), M_{sc}) :\Longrightarrow_{\mathsf{ax}} \text{ww-Race} \tag{7}$$

By applying Lemma. 8.18 on (6) and (7), we finish the proof. □

Lemma 8.14 (LSim ensures promise fulfilling - capped).

$$\begin{aligned}
&\forall \iota, TS_t, \mathcal{S}_t, M_t, TS_s, \mathcal{S}_s, M_s, \varphi, I, \iota, M_{sc}, \beta, n.\\
&\quad \iota \vdash (TS_t, \widehat{T}(M_t), \widehat{M_t}, \beta) \longmapsto^n ((_, _, \emptyset), _, _, _) \wedge\\
&\quad I, \iota \models (TS_t, \mathcal{S}_t, M_t) \preccurlyeq_{\varphi}^{\beta, \mathcal{D}} (TS_s, \mathcal{S}_s, M_s) \wedge\\
&\quad M_{sc} = (\{m \in M_s \mid \iota(m.\mathsf{var}) = \mathsf{na}\} \cup \{m \in \widehat{M_s} \mid \iota(m.\mathsf{var}) = \mathsf{at}\}) \wedge\\
&\quad \neg(\iota \vdash (TS_t, \mathcal{S}_t, M_t) \longrightarrow^* \textbf{abort})\\
&\implies \iota \vdash (TS_s, \widehat{T}(\widehat{M_s}), M_{sc}, \beta) \xrightarrow{\mathsf{pf}}{}^* ((_, _, \emptyset), _, _, _)
\end{aligned}$$

Proof. Prove by applying Lemma. 8.15. □

Lemma 8.15 (LSim ensures promise fulfilling - capped aux).

$$\begin{aligned}
&\forall TS_t, \mathcal{S}_{tc}, M_{tc}, \mathcal{S}_t, M_t, TS_s, \mathcal{S}_{sc}, M_{sc}, \mathcal{S}_s, M_s, \beta, \mathcal{D}, \varphi.\\
&\quad \iota \vdash (TS_t, \mathcal{S}_{tc}, M_{tc}, \beta) \longmapsto^n ((_, _, \emptyset), _, _, _) \wedge\\
&\quad M_t \subseteq M_{tc} \wedge (\forall m \in (M_{tc} - M_t).\ m = \langle _ : (_, _] \rangle) \wedge\\
&\quad I, \iota \models (TS_t, \mathcal{S}_t, M_t) \preccurlyeq_{\varphi}^{\beta, \mathcal{D}} (TS_s, \mathcal{S}_s, M_s) \wedge\\
&\quad (\forall \mathsf{x} \notin \iota.\ M_s(\mathsf{x}) = M_{sc}(\mathsf{x})) \wedge ([M_{tc}]_\iota \approx [M_{sc}]_\iota) \wedge\\
&\quad \neg(\iota \vdash (TS_t, \mathcal{S}_{tc}, M_{tc}) \longrightarrow^* \textbf{abort})\\
&\implies \iota \vdash (TS_s, \mathcal{S}_{sc}, M_{sc}, \beta) \xrightarrow{\mathsf{pf}}{}^* ((_, _, \emptyset), _, _, _)
\end{aligned}$$

Proof. Prove by induction on n. If n is zero, we prove by applying Lemma. 8.16. And if n is greater than zero, we prove by applying inductive hypothesis. □

Lemma 8.16 (LSim ensures promise fulfilling - capped with target prm empty).

$$\begin{aligned}
&\forall TS_t, \mathcal{S}_t, M_t, TS_s, \mathcal{S}_{sc}, M_{sc}, \mathcal{S}_s, M_s, \beta, \mathcal{D}, \varphi.\\
&\quad TS_t.P = \emptyset \wedge M_t \subseteq M_{tc} \wedge (\forall m \in (M_{tc} - M_t).\ m = \langle _ : (_, _] \rangle) \wedge\\
&\quad I, \iota \models (TS_t, \mathcal{S}_t, M_t) \preccurlyeq_{\varphi}^{\beta, \mathcal{D}} (TS_s, \mathcal{S}_s, M_s) \wedge\\
&\quad (\forall \mathsf{x} \notin \iota.\ M_s(\mathsf{x}) = M_{sc}(\mathsf{x})) \wedge ([M_{tc}]_\iota \approx [M_{sc}]_\iota) \wedge\\
&\quad \neg(\iota \vdash (TS_t, \mathcal{S}_t, M_t) \longrightarrow^* \textbf{abort})\\
&\implies \iota \vdash (TS_s, \mathcal{S}_{sc}, M_{sc}) \xrightarrow{\mathsf{pf}}{}^* ((_, _, \emptyset), _, _)
\end{aligned}$$

Proof. Prove by induction on the well-ordered delayed write set $\mathcal{D}$. The order of the delayed write set is $\mathcal{D} \ll \mathcal{D}' \triangleq \exists \mathcal{D}_0.\ \mathcal{D} \subseteq \mathcal{D}_0 \wedge \mathcal{D}_0 < \mathcal{D}'$. □

Lemma 8.17 (Race-free implies capped race-free).

$$\begin{aligned}
&\forall \mathcal{TP}, \mathsf{t}, \mathcal{S}, M, \beta, \iota, \mathcal{S}_c, M_c.\\
&\quad \neg((\mathcal{TP}, \mathsf{t}, \mathcal{S}, M, \beta)^\iota :\Longrightarrow_{\mathsf{ax}} \mathsf{ww\text{-}Race}) \wedge\\
&\quad M \subseteq M_c \wedge (\forall m \in (M_c - M).\ m = \langle _ : (_, _] \rangle) \wedge \mathcal{S} \leq \mathcal{S}_c\\
&\implies \neg((\mathcal{TP}, \mathsf{t}, \mathcal{S}_c, M_c, \beta)^\iota :\Longrightarrow_{\mathsf{ax}} \mathsf{Race})
\end{aligned}$$

Lemma. 8.18 shows that a thread from a write-write race free program can certify promises (for non-atomic writes) against the current memory instead of the capped memory. The intuition of the correctness of such lemma includes two points: (1) write-write race freedom forbids a thread t to write to a location when the memory contains a write of the same location made by another thread t′ and unobserved by t; (2) There is no atomic update operation performing on the non-atomic location.

$$
\begin{array}{rcl}
T \sim_\varphi T' & \triangleq & \forall \mathsf{x}.\ \varphi(\mathsf{x}, T(\mathsf{x})) = T'(\mathsf{x}) \\
V \sim_\varphi V' & \triangleq & V.T_{\mathsf{na}} \sim_\varphi V'.T_{\mathsf{na}} \wedge V.T_{\mathsf{rlx}} \sim_\varphi V'.T_{\mathsf{rlx}} \\
\mathcal{V} \sim_\varphi \mathcal{V}' & \triangleq & \mathcal{V}.cur \sim_\varphi \mathcal{V}'.cur \wedge \mathcal{V}.acq \sim_\varphi \mathcal{V}'.acq \wedge (\forall \mathsf{x}.\ \mathcal{V}.rel(\mathsf{x}) \sim_\varphi \mathcal{V}.rel(\mathsf{x})) \\
M \sim_\varphi M' & \triangleq & \varphi(M_t, M_s) \wedge \\
& & (\forall m_t \in M_t.\ \exists m_s \in M_s.\ \varphi(m_t.\mathsf{var}, m_t.\mathsf{to}) = m_s.\mathsf{to} \wedge m_t.\mathsf{var} = m_s.\mathsf{var} \wedge \\
& & \qquad m_t.\mathsf{val} = m_s.\mathsf{val} \wedge m_t.\mathcal{V} \sim_\varphi m_s.\mathcal{V})
\end{array}
$$

Fig. 27. Auxiliary definitions in promise certification preservation

Lemma 8.18 (Fulfilled under race-free implies promise consistent).

$$
\begin{array}{l}
\forall \mathcal{TP}, \mathsf{t}, \mathcal{S}, M, \iota, M_{sc}, n, \beta. \\
\quad \iota \vdash (\mathcal{TP}(\mathsf{t}), \widehat{T}(M), M_{sc}, \beta) \longmapsto^n ((\_, \_, \emptyset), \_, \_, \_) \wedge \\
\quad \neg((\mathcal{TP}, \mathsf{t}, \widehat{T}(\widehat{M}), M_{sc}, \beta)^\iota :\Longrightarrow_{\mathsf{ax}} \text{ww-Race}) \wedge \\
\quad M_{sc} = (\{m \in M \mid \iota(m.\mathsf{var}) = \mathsf{na}\} \cup \{m \in \widehat{M} \mid \iota(m.\mathsf{var}) = \mathtt{at}\}) \\
\Longrightarrow \ \mathsf{consistent}_{\mathsf{NP}}(\mathcal{TP}(\mathsf{t}), M, \beta, \iota)
\end{array}
$$

Proof. By unfolding the definitions of $\mathsf{consistent}_{\mathsf{NP}}$, we need to prove that for any $\mathcal{TP}, \mathsf{t}, \mathcal{S}, M, \iota, M_{sc}, n, \beta$, if

$$\iota \vdash (\mathcal{TP}(\mathsf{t}), \widehat{T}(M), M_{sc}, \beta) \longmapsto^n ((\_, \_, \emptyset), \_, \_, \_) \tag{1}$$

$$\neg((\mathcal{TP}, \mathsf{t}, \widehat{T}(\widehat{M}), M_{sc}, \beta)^\iota :\Longrightarrow_{\mathsf{ax}} \text{ww-Race}) \tag{2}$$

$$M_{sc} = (\{m \in M \mid \iota(m.\mathsf{var}) = \mathsf{na}\} \cup \{m \in \widehat{M} \mid \iota(m.\mathsf{var}) = \mathtt{at}\}) \tag{3}$$

then

$$\iota \vdash (\mathcal{TP}(\mathsf{t}), \widehat{T}(M), \widehat{M}, \beta) \longmapsto^* ((\_, \_, \emptyset), \_, \_, \_) \tag{g}$$

By applying lemma. 8.19 (in the proof of this lemma, we illustrate the main idea that a thread from a write-write race free program can certify promises (for non-atomic writes) against current memory instead of the capped memory and why we divide locations into non-atomic locations and atomic locations) on (g), we let $\varphi = \{(\mathsf{x}, t) \rightsquigarrow t \mid (\mathsf{x}, t) \in \lfloor\!\lfloor M_{sc} \rfloor\!\rfloor\}$ and need to prove the following.

$$\iota \vdash (\mathcal{TP}(\mathsf{t}), \widehat{T}(M), M_{sc}, \beta) \longmapsto^n ((\_, \_, \emptyset), \_, \_, \_) \tag{g1}$$

$$\mathcal{V} \sim_\varphi \mathcal{V} \tag{g2}$$

$$P \approx P \tag{g3}$$

$$M_{sc} \sim_\varphi \widehat{M} \tag{g4}$$

$$[M]_\iota \approx [\widehat{M}]_\iota \tag{g5}$$

$$M_{sc} \subseteq \widehat{M} \wedge (\forall m \in (\widehat{M} \backslash M_{sc}).\ m = \langle \_ : (\_, \_] \rangle) \tag{g6}$$

$$\neg((\mathcal{TP}, \mathsf{t}, \widehat{T}(\widehat{M}), M_{sc}, \beta)^\iota :\Longrightarrow_{\mathsf{ax}} \text{ww-Race}) \tag{g7}$$

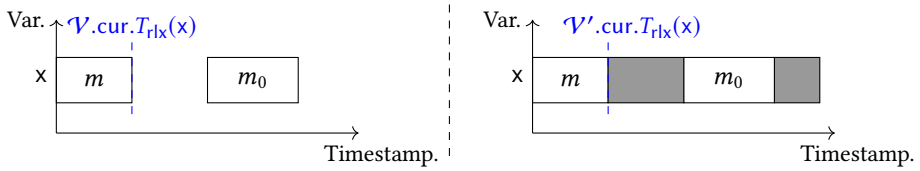
We prove (g1) by applying (1).
(g2) and (g3) can be proved according to the definitions in Fig. 27 directly.
(g4), (g5) and (g6) can be proved from (3).
We prove (g7) from (2).

□

Lemma 8.19 (Promise certification from current memory to capped memory).

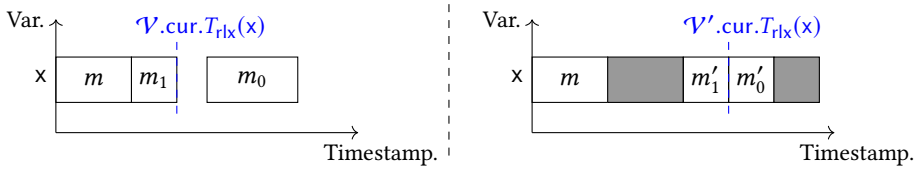
$$\begin{aligned}
&\forall n, \sigma, \mathcal{V}, P, \mathcal{S}, M, \beta, \mathcal{V}', P', M', \mathcal{TP}, \mathrm{t}, .\\
&\quad \iota \vdash ((\sigma, \mathcal{V}, P), \mathcal{S}, M, \beta) \longmapsto^{n} ((_, _, \emptyset), _, _, _) \wedge\\
&\quad \mathcal{V} \sim_{\varphi} \mathcal{V}' \wedge P \approx P' \wedge M \sim_{\varphi} M' \wedge [M]_{\iota} \approx [M']_{\iota}\\
&\quad M \subseteq M' \wedge (\forall m \in (M' \backslash M).\ m = \langle _ : (_, _] \rangle) \wedge\\
&\quad \neg((\mathcal{TP}, \mathrm{t}, \mathcal{S}, M)^{\iota} \Longrightarrow_{\mathsf{ax}} \text{ww-Race}) \wedge \mathcal{TP}(\mathrm{t}) = (\sigma, \mathcal{V}, P)\\
\Longrightarrow\ &\iota \vdash ((\sigma, \mathcal{V}', P'), \mathcal{S}, M', \beta) \longmapsto^{*} ((_, _, \emptyset), _, _, _).
\end{aligned}$$

Proof. We illustrate the main idea of the proof of such lemma. The atomic locations in M_{sc} and $\widehat{M}$ are the same. Thus, writing to the atomic locations in M_{sc} and $\widehat{M}$ has no difference. We focus on the non-atomic locations. Consider the following situation, the left side is the current memory on location x and the right side is the capped version of the current memory on location x. We assume that m and m_0 are all concrete messages. Here, the location x is a non-atomic location.

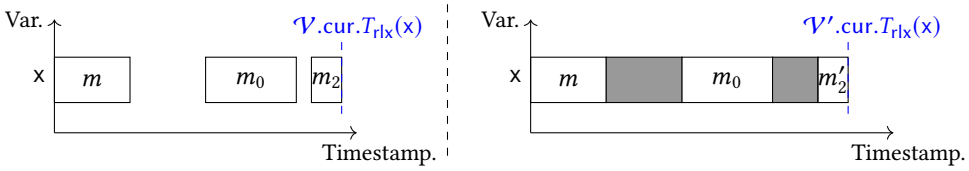


Consider that the thread does a memory write from the current state.

- If it inserts a new message m_1 between m and m_0, m_0 must be a promise of the thread. Otherwise, a write-write race arises. The corresponding memory write on the capped memory will split m_0 and insert m_1', which is the corresponding message of m_1. Note, one important thing here is that m_1 is not generated by the atomic update operation (e.g. CAS), since we prohibit the atomic update operation performed on the non-atomic location. If m_1 is generated by an atomic update operation, the "from"-timestamp of m_0 must equal to the "to"-timestamp of m, which is impossible to achieve on the capped memory.



- If it inserts a new message m_2, which has a timestamp larger than m_0, the corresponding memory write (generating m_2', which is the corresponding message of m_2) on the capped memory insert a message, which has a larger timestamp than the capped message. We show such condition in the following figure.



□

9 PROOF OF WRITE-WRITE RACE FREEDOM PRESERVING

We show the correctness proof of write-write race freedom preserving in the following.

Lemma 9.1 (ww-rf preserving).

$$\begin{aligned}&\forall \pi_t, \pi_s, I, \iota, \mathsf{f}_1, \ldots, \mathsf{f}_n.\\ &\quad \text{ww-NPRF}(\textbf{let } (\pi_s, \iota) \textbf{ in } \mathsf{f}_1 \mid \ldots \mid \mathsf{f}_n) \wedge\\ &\quad I, \iota \models \pi_t \preccurlyeq \pi_s \wedge\\ &\quad \text{Safe}(\textbf{let } (\pi_s, \iota) \textbf{ in } \mathsf{f}_1 \mid \ldots \mid \mathsf{f}_n)\\ &\Longrightarrow \text{ww-NPRF}(\textbf{let } (\pi_t, \iota) \textbf{ in } \mathsf{f}_1 \mid \ldots \mid \mathsf{f}_n)\end{aligned}$$

Proof. From the premises, we have the following.

$$\text{ww-NPRF}(\textbf{let } (\pi_s, \iota) \textbf{ in } \mathsf{f}_1 \mid \ldots \mid \mathsf{f}_n) \tag{1}$$
$$I, \iota \models \pi_t \preccurlyeq \pi_s \tag{2}$$
$$\text{Safe}(\textbf{let } (\pi_s, \iota) \textbf{ in } \mathsf{f}_1 \mid \ldots \mid \mathsf{f}_n) \tag{3}$$

We need to prove the following.

$$\text{ww-NPRF}(\textbf{let } (\pi_t, \iota) \textbf{ in } \mathsf{f}_1 \mid \ldots \mid \mathsf{f}_n) \tag{g}$$

We unfold ww-race freedom defined under the non-preemptive semantics and have the following.

$$\textbf{let } (\pi_t, \iota) \textbf{ in } \mathsf{f}_1 \mid \ldots \mid \mathsf{f}_n :\Longrightarrow \text{ww-Race} \tag{4}$$

And we need to prove the following.

$$\textbf{let } (\pi_s, \iota) \textbf{ in } \mathsf{f}_1 \mid \ldots \mid \mathsf{f}_n :\Longrightarrow \text{ww-Race} \tag{g1}$$

We unfold (4) and have that there exist $\mathcal{TP}_t$, t and $\hat{W}_t$ such that:

$$\textbf{let } (\pi_t, \iota) \textbf{ in } \mathsf{f}_1 \mid \ldots \mid \mathsf{f}_n :\overset{load}{\Longrightarrow} (\mathcal{TP}_t, \mathsf{t}, \mathcal{S}_\perp, M_0, \circ)^\iota \tag{4.1}$$
$$(\mathcal{TP}_t, \mathsf{t}, \mathcal{S}_\perp, M_0, \circ)^\iota :\Longrightarrow^* \hat{W}_t \tag{4.2}$$
$$\hat{W}_t :\Longrightarrow \text{ww-NPRF} \tag{4.3}$$

We unfold (2) and have the following.

$$I(\varphi_0, \iota, (\mathcal{S}_\perp, M_0, \mathcal{S}_\perp, M_0)) \wedge \mathsf{wf}(I) \tag{2.1}$$
$$\begin{aligned}&\forall \sigma_t.\ \mathsf{Init}(\pi_t, \mathsf{f}) = \sigma_t\\ &\quad\Longrightarrow \exists \sigma_s.\ (\mathsf{Init}(\pi_s, \mathsf{f}) = \sigma_s \wedge I, \iota \models ((\sigma_t, \mathcal{V}_\perp, \emptyset), \mathcal{S}_\perp, M_0) \preccurlyeq^{\circ, \emptyset}_{\varphi} ((\sigma_s, \mathcal{V}_\perp, \emptyset), \mathcal{S}_\perp, M_0))\end{aligned} \tag{2.2}$$

From (2.1), (2.2) and (4.1), we have that there exists $\mathcal{TP}_s$ such that:

$$\textbf{let } (\pi_s, \iota) \textbf{ in } \mathsf{f}_1 \mid \ldots \mid \mathsf{f}_n :\overset{load}{\Longrightarrow} (\mathcal{TP}_s, \mathsf{t}, \mathcal{S}_\perp, M_0, \circ)^\iota \tag{5}$$

We unfold (g1). From (5), we need to prove that there exists $\hat{W}_s$ such that.

$$(\mathcal{TP}_s, \mathsf{t}, \mathcal{S}_\perp, M_0, \circ)^\iota :\Longrightarrow^* \hat{W}_s \tag{g2.1}$$
$$\hat{W}_s :\Longrightarrow \text{ww-Race} \tag{g2.2}$$

We finish the proof from Lemma 9.2. □

Lemma 9.2 (ww-rf preserving - aux).

$$\begin{aligned}
&\forall \mathcal{TP}_t, i, \mathcal{S}_t, M_t, \hat{W}_t, \mathcal{TP}_s, \mathcal{S}_s, M_s, n, m, \varphi.\\
&\quad (\mathcal{TP}_t, i, \mathcal{S}_t, M_t, \circ)^\iota :\Longrightarrow^n \hat{W}_t \wedge \hat{W}_t :\models\!\Longrightarrow \text{ww-Race} \wedge\\
&\quad (\forall j \in \{1, \ldots, m\}.\ \text{consistent}_{\mathsf{NP}}(\mathcal{TP}_t(j), M_t, \circ, \iota)) \wedge\\
&\quad (\forall j \in \{1, \ldots, m\}.\ I, \iota \models (\mathcal{TP}_t(j), \mathcal{S}_t, M_t) \preccurlyeq_\varphi^{\circ, \emptyset} (\mathcal{TP}_s(j), \mathcal{S}_s, M_s)) \wedge\\
&\quad \mathcal{I}(\iota, \varphi, (\mathcal{S}_t, M_t, \mathcal{S}_s, M_s)) \wedge \text{wf}(I) \wedge\\
&\quad \neg(\exists \hat{W}_s.\ (\mathcal{TP}_s, i, \mathcal{S}_s, M_s, \circ)^\iota :\Longrightarrow^* \hat{W}_s \wedge \hat{W}_s :\Longrightarrow_{\text{ax}} \textbf{abort})\\
&\Longrightarrow\ \exists \hat{W}_s.\ (\mathcal{TP}_s, i, \mathcal{S}_s, M_s, \circ)^\iota :\Longrightarrow^* \hat{W}_s \wedge \hat{W}_s :\models\!\Longrightarrow \text{ww-Race}
\end{aligned}$$

Proof. From the premises, we have the following.

$$(\mathcal{TP}_t, i, \mathcal{S}_t, M_t, \circ)^\iota :\Longrightarrow^n \hat{W}_t \tag{1}$$

$$\hat{W}_t :\models\!\Longrightarrow \text{ww-Race} \tag{2}$$

$$(\forall j \in \{1, \ldots, m\}.\ \text{consistent}_{\mathsf{NP}}(\mathcal{TP}_t(j), M_t, \circ, \iota)) \tag{3}$$

$$(\forall j \in \{1, \ldots, m\}.\ I, \iota \models (\mathcal{TP}_t(j), \mathcal{S}_t, M_t) \preccurlyeq_\varphi^{\circ, \emptyset} (\mathcal{TP}_s(j), \mathcal{S}_s, M_s)) \tag{4}$$

$$\mathcal{I}(\iota, \varphi, (\mathcal{S}_t, M_t, \mathcal{S}_s, M_s)) \wedge \text{wf}(I) \tag{5}$$

$$\neg(\exists \hat{W}_s.\ (\mathcal{TP}_s, i, \mathcal{S}_s, M_s, \circ)^\iota :\Longrightarrow^* \hat{W}_s \wedge \hat{W}_s :\Longrightarrow_{\text{ax}} \textbf{abort}) \tag{6}$$

We have the following.

$$\begin{aligned}
&((\mathcal{TP}_t, i, \mathcal{S}_t, M_t, \circ)^\iota :\models\!\Longrightarrow_{\text{ax}} \text{ww-Race}) \vee\\
&\quad \neg((\mathcal{TP}_t, i, \mathcal{S}_t, M_t, \circ)^\iota :\models\!\Longrightarrow_{\text{ax}} \text{ww-Race})
\end{aligned} \tag{7}$$

We destruct (7) and discuss each case respectively.

- We first consider that the current target thread will generate data race.

  $$(\mathcal{TP}_t, i, \mathcal{S}_t, M_t, \circ)^\iota :\models\!\Longrightarrow_{\text{ax}} \text{ww-Race} \tag{7.1}$$

  By applying Lemma. 9.3 on (7.1), (4) and (5), we have the following.

  $$(\mathcal{TP}_s, \mathsf{t}, \mathcal{S}_s, M_s, \circ)^\iota :\models\!\Longrightarrow_{\text{ax}} \text{ww-Race} \tag{8}$$

  We finish the proof by applying Lemma. 8.8 on (7).
- Then, we consider that the current target thread will not generate data race.

  $$\neg((\mathcal{TP}_t, i, \mathcal{S}_t, M_t, \circ)^\iota :\models\!\Longrightarrow_{\text{ax}} \text{ww-Race}) \tag{7.2}$$

  We finish the prove by applying Lemma. 9.7 on (1), (2), (7.2), (3), (4) and (5).

□

Lemma 9.3 (ww-rf preserving - aux current race).

$$\begin{aligned}
&\forall \mathcal{TP}_t, \mathsf{t}, \mathcal{S}_t, M_t, \mathcal{TP}_s, \mathcal{S}_s, M_s, \iota, \varphi.\\
&\quad (\mathcal{TP}_t, \mathsf{t}, \mathcal{S}_t, M_t, \circ)^\iota :\models\!\Longrightarrow_{\text{ax}} \text{ww-Race} \wedge\\
&\quad I, \iota \models (\mathcal{TP}_t(\mathsf{t}), \mathcal{S}_t, M_t) \preccurlyeq_\varphi^{\circ, \emptyset} (\mathcal{TP}_s(\mathsf{t}), \mathcal{S}_s, M_s) \wedge\\
&\quad \mathcal{I}(\iota, \varphi, (\mathcal{S}_t, M_t, \mathcal{S}_s, M_s)) \wedge \text{wf}(I) \wedge\\
&\quad \neg((\mathcal{TP}_s(\mathsf{t}), \mathcal{S}_s, M_s) \longrightarrow^* \textbf{abort})\\
&\Longrightarrow\ (\mathcal{TP}_s, \mathsf{t}, \mathcal{S}_s, M_s, \circ)^\iota :\models\!\Longrightarrow_{\text{ax}} \text{ww-Race}
\end{aligned}$$

Proof. From the premises, we have the following.

$$(\mathcal{TP}_t, \mathsf{t}, \mathcal{S}_t, M_t, \circ)^\iota :\Longrightarrow_{\mathsf{ax}} \mathsf{ww\text{-}Race} \tag{1}$$

$$I, \iota \models (\mathcal{TP}_t(\mathsf{t}), \mathcal{S}_t, M_t) \preccurlyeq_{\varphi}^{\circ,\emptyset} (\mathcal{TP}_s(\mathsf{t}), \mathcal{S}_s, M_s) \tag{2}$$

$$I(\iota, \varphi, (\mathcal{S}_t, M_t, \mathcal{S}_s, M_s)) \wedge \mathsf{wf}(I) \tag{3}$$

$$\neg((\mathcal{TP}_s(\mathsf{t}), \mathcal{S}_s, M_s) \longrightarrow^* \textbf{abort}) \tag{4}$$

We unfold (1) and have the following.

$$\iota \vdash (\mathcal{TP}_t(\mathsf{t}), \mathcal{S}_t, M_t) \longrightarrow^* (TS'_t, \mathcal{S}'_t, M'_t) \tag{1.1}$$

$$TS'_t.\sigma \xrightarrow{\mathsf{W(na,x,\_)}} \_ \tag{1.2}$$

$$\langle \mathsf{x} : \_@(\_, t], \_\rangle \in (M'_t \backslash TS'_t.P) \tag{1.3}$$

$$TS'_t.\mathcal{V}.\mathsf{cur}.T_{\mathsf{rlx}}(\mathsf{x}) < t \tag{1.4}$$

$$\iota \vdash (TS'_t, \mathcal{S}'_t, M'_t) \xrightarrow{\mathsf{pf}}{}^* ((\_, \_, \emptyset), \_, \_) \tag{1.5}$$

From (1.1), we have that there exist β' such that:

$$\iota \vdash (\mathcal{TP}_t(\mathsf{t}), \mathcal{S}_t, M_t, \circ) \longmapsto^* (TS'_t, \mathcal{S}'_t, M'_t, \beta') \tag{5}$$

According to the thread-local simulation, we have that there exist TS'_s, $\mathcal{S}'_s$, M'_s, φ' and β'_s such that:

$$\iota \vdash (\mathcal{TP}_s(\mathsf{t}), \mathcal{S}_s, M_s, \circ) \longmapsto^* (TS'_s, \mathcal{S}'_s, M'_s, \beta'_s) \tag{6}$$

$$I, \iota \models (TS'_t, \mathcal{S}'_t, M'_t) \preccurlyeq_{\varphi'}^{\beta',\mathcal{D}'} (TS'_s, \mathcal{S}'_s, M'_s) \tag{7}$$

$$\varphi \subseteq \varphi' \tag{8}$$

By applying Lemma. 9.4 in (1.1) and (1.3) and (1.4), we have the following.

$$\langle \mathsf{x} : \_@(\_, t], \_\rangle \in (M_t \backslash TS_t.P) \tag{9}$$

From (3), we get that there exist and injection relation between the target memory and source memory. Thus, we have that there exists t' such that:

$$\varphi(\mathsf{x}, t) = t' \tag{10}$$

$$\langle \mathsf{x} : \_@(\_, t'], \_\rangle \in (M_s \backslash TS_s.P) \tag{11}$$

We apply Lemma. 9.5 on (11) and (5). And we have the following.

$$\langle \mathsf{x} : \_@(\_, t'], \_\rangle \in (M'_s \backslash TS'_s.P) \tag{12}$$

From (7), (8), (10) and (1.4), we have the following.

$$TS'_s.\mathcal{V}.\mathsf{cur}.T_{\mathsf{rlx}}(\mathsf{x}) < t' \tag{13}$$

By applying Lemma. 9.6 on (1.2), (1.5), (7), (12), (13) and (4), we construct a write-write race under the source execution. □

Lemma 9.4 (race message in starting memory).

$$\begin{aligned}
&\forall TS, \mathcal{S}, M, TS', \mathcal{S}', M', \iota, n.\\
&\quad \iota \vdash (TS, \mathcal{S}, M) \longrightarrow^n (TS', \mathcal{S}', M') \wedge\\
&\quad \langle \mathsf{x} : _@(_, t], _\rangle \in (M' \backslash TS'.P) \wedge TS'.\mathcal{V}.\mathsf{cur}.T_{\mathsf{rlx}}(\mathsf{x}) < t\\
&\implies \langle \mathsf{x} : _@(_, t], _\rangle \in (M \backslash TS.P)
\end{aligned}$$

Lemma 9.5 (non-promise message preserving).

$$\begin{aligned}
&\forall TS, \mathcal{S}, M, TS', \mathcal{S}', M', \mathsf{x}, t, n, \iota.\\
&\quad \langle \mathsf{x} : _@(_, t], _\rangle \in (M \backslash TS.P) \wedge\\
&\quad \iota \vdash (TS, \mathcal{S}, M) \longrightarrow^{n} (TS', \mathcal{S}', M')\\
&\implies \langle \mathsf{x} : _@(_, t], _\rangle \in (M' \backslash TS'.P)
\end{aligned}$$

Lemma 9.6 (source write-write race construction).

$$\begin{aligned}
&\forall TS_t, \mathcal{S}_t, M_t, TS_s, \mathcal{S}_s, M_s, \iota, \mathsf{x}, t', \beta, \mathcal{D}, \varphi, n.\\
&\quad TS_t.\sigma \xrightarrow{\mathsf{W(na,x,_)}} _ \wedge\\
&\quad \iota \vdash (TS_t, \mathcal{S}_t, M_t) \xrightarrow{\mathsf{pf}}{}^{n} ((_, _, \emptyset), _, _) \wedge\\
&\quad I, \iota \models (TS_t, \mathcal{S}_t, M_t) \preccurlyeq_{\varphi}^{\beta, \mathcal{D}} (TS_s, \mathcal{S}_s, M_s) \wedge\\
&\quad \langle \mathsf{x} : _@(_, t'], _\rangle \in (M_s \backslash TS_s.P) \wedge TS_s.\mathcal{V}.\mathsf{cur}.T_{\mathsf{rlx}}(\mathsf{x}) < t' \wedge\\
&\quad \neg(\iota \vdash (TS_s, \mathcal{S}_s, M_s) \longrightarrow^{*} \textbf{abort})\\
&\implies \exists TS_{s0}, \mathcal{S}_{s0}, M_{s0}.\\
&\qquad \iota \vdash (TS_s, \mathcal{S}_s, M_s) \longrightarrow^{*} (TS_{s0}, \mathcal{S}_{s0}, M_{s0}) \wedge\\
&\qquad \langle \mathsf{x} : _@(_, t'], _\rangle \in (M'_{s0} \backslash TS_{s0}.P) \wedge TS_{s0}.\mathcal{V}.\mathsf{cur}.T_{\mathsf{rlx}}(\mathsf{x}) < t' \wedge\\
&\qquad \iota \vdash (TS_{s0}, \mathcal{S}_{s0}, M_{s0}) \longrightarrow^{*} ((_, _, \emptyset), _, _)
\end{aligned}$$

Lemma 9.7 (ww-rf preserving - aux current not race).

$$\begin{aligned}
&\forall \mathcal{TP}_t, i, \mathcal{S}_t, M_t, \hat{W}_t, \mathcal{TP}_s, \mathcal{S}_s, M_s, \beta, \beta_s, \mathcal{D}, n, m.\\
&\quad (\mathcal{TP}_t, i, \mathcal{S}_t, M_t, \beta)^{\iota} :\Longrightarrow^{n} \hat{W}_t \wedge \hat{W}_t :\Longrightarrow \mathsf{ww\text{-}Race} \wedge\\
&\quad \neg((\mathcal{TP}_t, i, \mathcal{S}_t, M_t, \beta)^{\iota} :\Longrightarrow_{\mathsf{ax}} \mathsf{ww\text{-}Race}) \wedge\\
&\quad (\forall j \in \{1, \ldots, m\}.\ \mathsf{consistent}_{\mathsf{NP}}(\mathcal{TP}_t(j), M_t, \circ, \iota)) \wedge\\
&\quad (\forall j \in \{1, \ldots, m\} \backslash \{i\}.\ I, \iota \models (\mathcal{TP}_t(j), \mathcal{S}_t, M_t) \preccurlyeq_{\varphi}^{\circ, \emptyset} (\mathcal{TP}_s(j), \mathcal{S}_s, M_s)) \wedge\\
&\quad I, \iota \models (\mathcal{TP}_t(i), \mathcal{S}_t, M_t) \preccurlyeq_{\varphi}^{\beta, \mathcal{D}} (\mathcal{TP}_s(i), \mathcal{S}_s, M_s)) \wedge\\
&\quad (\beta = \circ \implies \beta_s = \circ) \wedge \mathsf{wf}(I) \wedge\\
&\quad \neg(\exists \hat{W}_s.\ (\mathcal{TP}_s, i, \mathcal{S}_s, M_s, \beta_s)^{\iota} :\Longrightarrow^{*} \hat{W}_s \wedge \hat{W}_s :\Longrightarrow_{\mathsf{ax}} \textbf{abort})\\
&\implies \exists \hat{W}_s.\ (\mathcal{TP}_s, i, \mathcal{S}_s, M_s, \beta_s)^{\iota} :\Longrightarrow^{*} \hat{W}_s \wedge \hat{W}_s :\Longrightarrow \mathsf{ww\text{-}Race}
\end{aligned}$$

Proof. Prove by induction on n.

0: From the premises, we have the following.

$$(\mathcal{TP}_t, i, \mathcal{S}_t, M_t, \beta)^{\iota} :\Longrightarrow \mathsf{ww\text{-}Race} \tag{1}$$

$$\neg((\mathcal{TP}_t, i, \mathcal{S}_t, M_t, \beta)^{\iota} :\Longrightarrow_{\mathsf{ax}} \mathsf{ww\text{-}Race}) \tag{2}$$

$$\mathsf{consistent}_{\mathsf{NP}}(\mathcal{TP}_t(i), M_t, \circ, \iota) \tag{3}$$

From (1) and (3), we have the following.

$$(\mathcal{TP}_t, i, \mathcal{S}_t, M_t, \beta)^{\iota} :\Longrightarrow_{\mathsf{ax}} \mathsf{ww\text{-}Race} \tag{4}$$

Thus, we construct a contradiction.

$n+1$: From the premises, we have the following.

$$(\mathcal{TP}_t, i, \mathcal{S}_t, M_t, \beta)^\iota :\Longrightarrow^{n+1} \hat{W}_t \tag{5}$$

$$\hat{W}_t :\Longrightarrow \text{ww-Race} \tag{6}$$

$$\neg((\mathcal{TP}_t, i, \mathcal{S}_t, M_t, \beta)^\iota :\Longrightarrow_{\mathsf{ax}} \text{ww-Race}) \tag{7}$$

$$(\forall j \in \{1, \ldots, m\}.\ \mathsf{consistent}_{\mathsf{NP}}(\mathcal{TP}_t(j), M_t, \circ, \iota)) \tag{8}$$

$$(\forall j \in \{1, \ldots, m\} \backslash \{i\}.\ I, \iota \models (\mathcal{TP}_t(j), \mathcal{S}_t, M_t) \preccurlyeq_{\varphi}^{\circ, \emptyset} (\mathcal{TP}_s(j), \mathcal{S}_s, M_s)) \tag{9}$$

$$I, \iota \models (\mathcal{TP}_t(i), \mathcal{S}_t, M_t) \preccurlyeq_{\varphi}^{\beta, \mathcal{D}} (\mathcal{TP}_s(i), \mathcal{S}_s, M_s)) \tag{10}$$

$$(\beta = \circ \implies \beta_s = \circ) \wedge \mathsf{wf}(I) \tag{11}$$

$$\neg(\exists \hat{W}_s.\ (\mathcal{TP}_s, i, \mathcal{S}_s, M_s, \beta)^\iota :\Longrightarrow^* \hat{W}_s \wedge \hat{W}_s :\Longrightarrow_{\mathsf{ax}} \textbf{abort}) \tag{12}$$

We unfold (5) and have that there exists $\hat{W}_t'$ such that:

$$(\mathcal{TP}_t, i, \mathcal{S}_t, M_t, \beta)^\iota :\Longrightarrow \hat{W}_t' \tag{5.1}$$

$$\hat{W}_t' :\Longrightarrow^n \hat{W}_t \tag{5.2}$$

We unfold (5.1) and discuss each case respectively. We let $\hat{W}_t' = (\mathcal{TP}_t', i', \mathcal{S}_t', M_t', \beta')^\iota$.

- The current target thread does not take an output step. We have that there exists TS_t' such that:

$$\iota \vdash (\mathcal{TP}_t(i), \mathcal{S}_t, M_t, \beta) \longmapsto^+ (TS_t', \mathcal{S}_t', M_t', \beta') \tag{6.1.1}$$

$$\mathsf{consistent}_{\mathsf{NP}}(TS_t', M_t', \beta', \iota) \tag{6.1.2}$$

$$\mathcal{TP}_t' = \mathcal{TP}_t\{i \rightsquigarrow TS_t'\} \tag{6.1.3}$$

  We apply Lemma. 8.11 on (6.1.1) and (10) and have that there exist TS_s', $\mathcal{S}_s'$, M_s', $\mathcal{D}'$, φ' and β_s' such that:

$$\iota \vdash (\mathcal{TP}_s(i), \mathcal{S}_s, M_s, \beta_s) \longmapsto^* (TS_s', \mathcal{S}_s', M_s', \beta_s') \tag{7.1.1}$$

$$I, \iota \models (\mathcal{TP}_t(i), \mathcal{S}_t', M_t') \preccurlyeq_{\varphi'}^{\beta', \mathcal{D}'} (\mathcal{TP}_s(i), \mathcal{S}_s', M_s') \tag{7.1.2}$$

$$\mathcal{TP}_s' = \mathcal{TP}_s\{i \rightsquigarrow TS_s'\} \tag{7.1.3}$$

$$(\beta' = \circ \implies \beta_s' = \circ) \wedge \mathsf{wf}(I) \tag{7.1.4}$$

  By applying Lemma. 8.13 on (6.1.2), (7.1.2), (12) and (7), we have the following.

$$\mathsf{consistent}_{\mathsf{NP}}(TS_s', M_s', \beta_s', \iota) \tag{13}$$

  From (7.1.1) and (13), we construct a source program transition. We can finish the proof of such case from inductive hypothesis.
- The proof of the case that the current target thread takes an output step is similar with the previous one.
- We consider that the target program takes a switch step. We have $\beta' = \circ$ and we discuss whether the new target thread will generate write-write race.

$$\begin{array}{l}((\mathcal{TP}_t', i', \mathcal{S}_t', M_t', \circ)^\iota :\Longrightarrow_{\mathsf{ax}} \text{ww-Race}) \vee \\ \quad \neg((\mathcal{TP}_t', i', \mathcal{S}_t', M_t', \circ)^\iota :\Longrightarrow_{\mathsf{ax}} \text{ww-Race})\end{array} \tag{14}$$

  We destruct (14) and discuss each case respectively.
  – We first consider that the new target thread will generate write-write race.

$$(\mathcal{TP}_t', i', \mathcal{S}_t', M_t', \circ)^\iota :\Longrightarrow_{\mathsf{ax}} \text{ww-Race} \tag{13.1}$$

    We finish the proof from Lemma. 9.3.

– Then, we consider that the new target thread will not generate write-write race.

$$\neg((\mathcal{TP}_t', i', \mathcal{S}_t', M_t', \circ)^\iota :\Longrightarrow_{\mathsf{ax}} \text{ww-Race}) \tag{14.2}$$

We finish the proof from inductive hypothesis.

□

In the proof of write-write race freedom preserving, we require that the thread-local transition is *deterministic*.

Definition 9.8 (deterministic thread local transistion).

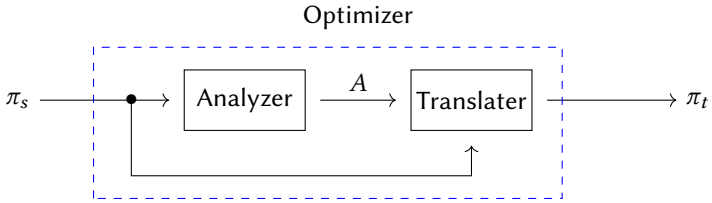
$$\begin{aligned}
&\forall \sigma, \sigma_1, te_1, \sigma_2, te_2. \\
&\quad (\sigma \xrightarrow{te_1} \sigma_1 \wedge \sigma \xrightarrow{te_2} \sigma_2) \\
&\implies (te_1 = te_2 \wedge \sigma_1 = \sigma_2) \vee (\exists x, o.\ te_1, te_2 \in \{\mathsf{R}(o, \mathsf{x}, _)\}) \vee \\
&\qquad (te_1 = \mathsf{U}(_, _, _, _, _) \vee te_2 = \mathsf{U}(_, _, _, _, _))
\end{aligned}$$

$$
\begin{array}{rlll}
\textit{(AI)} & L & \triangleq & \ldots \\
\textit{(AIB)} & LB & \triangleq & \epsilon \mid L :: LB \\
\textit{(AIF)} & \mathbb{L} & \triangleq & \{l_1 \rightsquigarrow LB_1, \ldots, l_n \rightsquigarrow LB_n\} \\
\textit{(AResP)} & A & \triangleq & \{\mathtt{f}_1 \rightsquigarrow \mathbb{L}_1, \ldots, \mathtt{f}_n \rightsquigarrow \mathbb{L}_n\}
\end{array}
$$

Fig. 28. Definition of analysis result

10 DEFINITION OF OPTIMIZERS

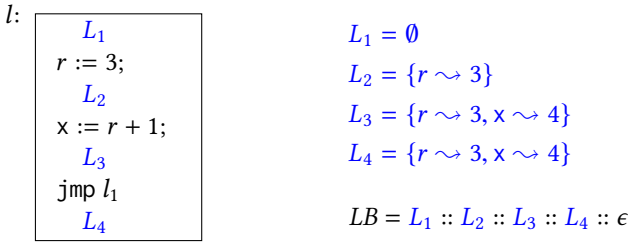
In this work, we focus on the correctness proof of optimizers. Many optimizers are implemented based on the program analysis, such as: the *constant propagation*, the *dead code elimination* and the *common subexpression elimination*. The optimizer may have the following form.



The optimizer is composed of the analyzer and the translater. The analyzer analyzes the source code π_s and gets the result of the code analysis A. Then, the translater optimizes the source code π_s according to A. In this section, we will define the form of the program analysis result in Subsec. 10.1. In Subsec. 10.2, we will give the definition of the value analysis. We will define the constant propagation optimization based on the value analysis.

10.1 The result of program analysis

In this subsection, we focus on defining the result of the program analysis. We show the definition of the analysis result in Fig. 28. Here, we use L to represent the abstract interpretation at each program point, LB that is a sequence of L to represent the abstract interpretation of each basic and $\mathbb{L}$ to represent the abstract interpretation of a code heap, which is a partial mapping from the label to the corresponding LB. The analysis result of the whole program is shown as A, which is a collection of the result of code analysis of each code heap. We show more details about their meaning using the following figure.



The above figure shows the result of the abstract interpretation of a basic code block in the value analysis. We give some auxiliary definitions on LB in Fig. 29.

$$\mathsf{IN}[LB] \triangleq \begin{cases} L & \textit{if } LB = L :: LB' \\ \text{undef} & \textit{otherwise} \end{cases} \qquad \mathsf{OUT}[LB] \triangleq \begin{cases} L & \textit{if } LB = LB' \cdot L \\ \text{undef} & \textit{otherwise} \end{cases}$$

$$\mathsf{succ}(B) \triangleq \begin{cases} \{l\} & \textit{if } B = B' \cdot (\texttt{jmp } l) \textit{ or } B = B' \cdot (\texttt{call f}, l_{ret}) \\ \{l_1, l_2\} & \textit{if } B = B' \cdot (\texttt{be } e, l_1, l_2) \\ \emptyset & \textit{otherwise} \end{cases}$$

$$\mathsf{pred}(C, l) \triangleq \{l_p \mid l \in \mathsf{succ}(C(l_p))\}$$

$$B[i \ldots] \triangleq B_2 \quad \textit{where } B = (B_1 \cdot B_2) \textit{ and } |B_1| = i$$

$$LB[i \ldots] \triangleq LB_2 \quad \textit{where } LB = (LB_1 \cdot LB_2) \textit{ and } |LB_1| = i$$

$$LB(i) \triangleq L \quad \textit{where } LB = (LB_1 \cdot L \cdot LB_2) \textit{ and } |LB_1| = i$$

Fig. 29. Auxiliary definitions on the abstract interpretation of code block

10.2 Value analysis

The set L_v records the abstract interpretation of the values of variables and registers in the current state.

$$L_v \in \mathcal{P}((Var \cup Reg) \rightharpoonup Val) \cup \{\top\}$$

The Implementation of the value analysis is shown below (n is a very large constant).

$$\mathsf{Val\_Analyzer}(C, l_0) \triangleq \mathsf{Val\_Analyzer}'(C, \mathbb{L}_0, \mathrm{dom}(C), n)$$
$$\textit{where} \quad \mathbb{L}_0 = \{l \rightsquigarrow (\top :: \epsilon) \mid l \in \mathrm{dom}(C)\}\{l_0 \rightsquigarrow (\emptyset :: \epsilon)\}$$

$$\mathsf{Val\_Analyzer}'(C, \mathbb{L}, W, n) \triangleq$$
$$\begin{cases} \mathsf{Val\_Analyzer}'(C, \mathbb{L}\{l \rightsquigarrow L_v\}, W', n-1) & \textit{if } l \in W, L_v = \bigcap\limits_{l_p \in \mathsf{pred}(C,l)} \mathsf{OUT}[\mathbb{L}(l_p)], \\ & \quad L'_v = \mathsf{TF}_v(L_v, C(l)), \\ & \quad (L'_v \neq \mathsf{OUT}[\mathbb{L}(l)] \implies W' = ((W \backslash \{l\}) \cup \mathsf{succ}(l))), \\ & \quad (L'_v = \mathsf{OUT}[\mathbb{L}(l)] \implies W' = (W \backslash \{l\})), \\ \mathbb{L} & \textit{if } W = \emptyset \\ \text{undef} & \textit{otherwise} \end{cases}$$

The value analysis on the whole program is defined as the following form. In this work, we only consider the intraprocedural analysis.

$$\mathsf{PVal\_Analyzer}(\pi) \triangleq \{\mathtt{f} \rightsquigarrow \mathsf{Val\_Analyzer}(C) \mid \pi(\mathtt{f}) = (C, l)\}$$

The transfer function TF_v for basic code blocks in the value analysis is defined below.

$$\mathsf{TF}_v(L_v, B) \triangleq \begin{cases} \top :: \epsilon & \textit{if } L_v = \top \\ L_v :: \mathsf{TF}_v(L'_v, B') & \textit{elif } B = c :: B' \wedge L'_v = f_v(c, L_v) \\ L_v :: f_v(B, L_v) :: \epsilon & \textit{elif } B \in \{\texttt{return}, \texttt{call}(\mathtt{f}, l_{ret}), \texttt{jmp } l, \texttt{be } e, l_1, l_2\} \\ \text{undef} & \textit{otherwise} \end{cases}$$

$$[\![e]\!]_{L_v} \triangleq \begin{cases} L(r) & \text{if } e = r \\ v & \text{if } e = v \\ v_1 + v_2 & \text{if } e = e_1 + e_2 \wedge [\![e_1]\!]_{L_v} = v_1 \wedge [\![e_2]\!]_{L_v} = v_2 \\ v_1 - v_2 & \text{if } e = e_1 - e_2 \wedge [\![e_1]\!]_{L_v} = v_1 \wedge [\![e_2]\!]_{L_v} = v_2 \\ v_1 * v_2 & \text{if } e = e_1 * e_2 \wedge [\![e_1]\!]_{L_v} = v_1 \wedge [\![e_2]\!]_{L_v} = v_2 \\ \text{undef} & \text{otherwise} \end{cases}$$

Fig. 30. Auxiliary definitions in value analysis

We show the transfer function for each instruction in the following. Some auxiliary definitions used in defining the transfer function for each instruction are shown in Fig. 30.

- Assignment operation

$$f_v(r := e, L_v) \triangleq \begin{cases} L_v\{r \rightsquigarrow v\} & \text{if } [\![e]\!]_{L_v} = v \\ L_v \backslash \{r\} & \text{otherwise} \end{cases}$$

- Memory store operation

$$f_v(\mathsf{x}_{o_w} := e, L_v) \triangleq \begin{cases} L_v\{\mathsf{x} \rightsquigarrow v\} & \text{if } o_w = \mathsf{na} \text{ and } [\![e_2]\!]_{L_v} = v \\ L_v \backslash \{\mathsf{x}\} & \text{otherwise} \end{cases}$$

We focus on the case that the store operation is an atomic write (where $o_w \in \{\mathsf{rlx}, \mathsf{rel}\}$). Here, we do not simply view the atomic write as an external function call, which may modify memory arbitrarily. Consider the following example.

$$\begin{array}{l} \{\mathsf{x} \rightsquigarrow 3\} \\ \quad \mathsf{y}_{\mathsf{rel}} := 3 \\ \{\mathsf{x} \rightsquigarrow 3\} \end{array}$$

Before the execution of the instruction "$\mathsf{y}_{\mathsf{rel}} := 3$", the abstract interpretation of the program state "$\{\mathsf{x} \rightsquigarrow 3\}$" means that the last message on x that current thread can read has value 3. We can find that, after the execution of the instruction "$\mathsf{y}_{\mathsf{rel}} := 3$", the current thread can still read the value 3 from the variable x. Thus, we still have "$\{\mathsf{x} \rightsquigarrow 3\}$". Since we do not optimize the atomic memory access, we assign $\top$, which represents any value, to the variable y. We can do constant propagation across the atomic memory access as shown below.

$$\begin{array}{l} \{\} \\ \quad \mathsf{x}_{\mathsf{na}} := 2; \\ \{\mathsf{x} \rightsquigarrow 2\} \\ \quad \mathsf{y}_{\mathsf{rel}} := 1; \\ \{\mathsf{x} \rightsquigarrow 2\} \\ \quad r := \mathsf{x}_{\mathsf{na}} \end{array} \xrightarrow{\textit{ConstProp}} \begin{array}{l} \mathsf{x}_{\mathsf{na}} := 2; \\ \mathsf{y}_{\mathsf{rel}} := 1; \\ r := 2; \end{array}$$

The soundness of the above optimization can also be shown by "roach-motal reordering". The soundness transformation to achieve the above optimization by "roach-motal reordering" is shown below.

$$\begin{array}{l} \mathsf{x_{na}} := 2; \\ \mathsf{y_{rel}} := 1; \\ r := \mathsf{x_{na}}; \end{array} \rightsquigarrow \begin{array}{l} \mathsf{x_{na}} := 2; \\ r := \mathsf{x_{na}}; \\ \mathsf{y_{rel}} := 1; \end{array} \rightsquigarrow \begin{array}{l} \mathsf{x_{na}} := 2; \\ r := 2; \\ \mathsf{y_{rel}} := 1; \end{array} \rightsquigarrow \begin{array}{l} \mathsf{x_{na}} := 2; \\ \mathsf{y_{rel}} := 1; \\ r := 2; \end{array}$$

- Memory load operation

$$f_v(r := \mathsf{x_{na}}, L_v) \triangleq \begin{cases} L_v\{r \rightsquigarrow v\} & \textit{if } L_v(\mathsf{x}) = v \\ L_v \backslash \{r\} & \textit{otherwise} \end{cases}$$

$$f_v(r := \mathsf{x_{rlx}}, L_v) \triangleq L_v \backslash \{r\}$$

$$f_v(r := \mathsf{x_{acq}}, L_v) \triangleq \{r' \rightsquigarrow v \mid L_{nl}(r') = v \wedge r' \neq r\}$$

Code optimizations across relaxed atomic read is sound, since the execution of the relaxed atomic read does not achieve synchronization between threads. Thus, the value of a location, which can be read before the execution of the relaxed atomic read, can still be read after the execution of the relaxed atomic read.

$$\begin{array}{l} \{\} \\ \quad \mathsf{x_{na}} := 2; \\ \{\mathsf{x} \rightsquigarrow 2\} \\ \quad r_1 := \mathsf{y_{rlx}}; \\ \{\mathsf{x} \rightsquigarrow 2\} \\ \quad r_2 := \mathsf{x_{na}}; \end{array} \xRightarrow{\textit{ConstProp}} \begin{array}{l} \mathsf{x_{na}} := 2; \\ r_1 := \mathsf{y_{rlx}}; \\ r_2 := 2; \end{array}$$

The above optimization can be achieved according to the soundness code transformation as shown below.

$$\begin{array}{l} \mathsf{x_{na}} := 2; \\ r_1 := \mathsf{y_{rlx}}; \\ r_2 := \mathsf{x_{na}}; \end{array} \rightsquigarrow \begin{array}{l} \mathsf{x_{na}} := 2; \\ r_2 := \mathsf{x_{na}}; \\ r_1 := \mathsf{y_{rlx}}; \end{array} \rightsquigarrow \begin{array}{l} \mathsf{x_{na}} := 2; \\ r_2 := 2; \\ r_1 := \mathsf{y_{rlx}}; \end{array} \rightsquigarrow \begin{array}{l} \mathsf{x_{na}} := 2; \\ r_1 := \mathsf{y_{rlx}}; \\ r_2 := 2; \end{array}$$

Doing constant propagation accross acquire atomic read is not sound, since the execution of the acquire atomic read may implement synchronization between two threads.

- Compare and set operation

$$f_v(r := \mathsf{CAS_{rlx},}_{o_w}(\mathsf{x}, e_r, e_w), L_v) \triangleq L_v \backslash \{r, \mathsf{x}\}$$

$$f_v(r := \mathsf{CAS_{acq},}_{o_w}(\mathsf{x}, e_r, e_w), L_v) \triangleq L_v \backslash (Var \cup \{r\})$$

The transfer function for the CAS operation can be viewed as a composition of the transfer functions for the memory load operation and the memory store operation. If memory order for memory load in CAS is relaxed, the transfer function for CAS is defined as a composition of the relaxed atomic read and the atomic write. If the memory order for memory load in CAS is acquired, the transfer function for CAS is defined as a composition of the acquire atomic read and the atomic write. The following constant propagation optimization is correct.

$$\begin{array}{l}\{\} \\ \quad \mathsf{x}_{\mathsf{na}} := 2; \\ \{x \rightsquigarrow 2\} \\ \quad r_1 := \mathsf{CAS}_{\mathsf{rlx},\mathsf{rel}}(\mathsf{y}, 0, 1); \\ \{\mathsf{x} \rightsquigarrow 2\} \\ \quad r_2 := \mathsf{x}_{\mathsf{na}};\end{array} \xrightarrow{\textit{ConstProp}} \begin{array}{l}\mathsf{x}_{\mathsf{na}} := 2; \\ r_1 := \mathsf{CAS}_{\mathsf{rlx},\mathsf{rel}}(y, 0, 1); \\ r_2 := 2;\end{array}$$

- Release and acquire fence operations

$$\begin{aligned} f_v(\texttt{fence-rel}, L_v) &\triangleq L_v \\ f_v(\texttt{fence-acq}, L_v) &\triangleq L_v \backslash \mathit{Var} \end{aligned}$$

The transfer function for the release fence operation is similar with the transfer function for release atomic write. And the transfer function for the acquire fence operation is similar with the transfer function for acquire atomic read.

$$\begin{array}{l}\{\} \\ \quad \mathsf{x}_{\mathsf{na}} := 1; \\ \{\mathsf{x} \rightsquigarrow 1\} \\ \quad \texttt{fence-rel}; \\ \{\mathsf{x} \rightsquigarrow 1\} \\ \quad r := \mathsf{x}_{\mathsf{na}};\end{array} \xrightarrow{\textit{ConstProp}} \begin{array}{l}\mathsf{x}_{\mathsf{na}} := 1; \\ \texttt{fence-rel}; \\ r := 1;\end{array}$$

The correctness of the above optimization can be achieved by applying soundness code transformation as shown below.

$$\begin{array}{l}\mathsf{x}_{\mathsf{na}} := 1; \\ \texttt{fence-rel}; \\ r := \mathsf{x}_{\mathsf{na}};\end{array} \rightsquigarrow \begin{array}{l}\mathsf{x}_{\mathsf{na}} := 1; \\ r := \mathsf{x}_{\mathsf{na}}; \\ \texttt{fence-rel};\end{array} \rightsquigarrow \begin{array}{l}\mathsf{x}_{\mathsf{na}} := 1; \\ r := 1; \\ \texttt{fence-rel};\end{array} \rightsquigarrow \begin{array}{l}\mathsf{x}_{\mathsf{na}} := 1; \\ \texttt{fence-rel}; \\ r := 1;\end{array}$$

Doing constant propagation across `fence-acq` is not sound, since the execution of the acquire fence operation will update the view to each location of the current thread.

- Unconditional and conditional branch

$$\begin{aligned} f_v(\texttt{jmp}\ l, L_v) &\triangleq L_v \\ f_v(\texttt{be}\ e, l_1, l_2, L_v) &\triangleq L_v \end{aligned}$$

- Function call, return, system call and SC fence

$$\begin{aligned} f_v(\texttt{call}(l, l_{ret}), L_v) &\triangleq L_v \backslash \mathit{Addr} \\ f_v(\texttt{print}(e), L_v) &\triangleq L_v \backslash \mathit{Addr} \\ f_v(\texttt{fence-sc}, L_v) &\triangleq L_v \backslash \mathit{Addr} \end{aligned}$$

Since we consider intra-procedural analysis in this work, the callee may modify memory state arbitrarily. The definition of the system call is the same as `fence-sc`, thus their transfer functions are same.

$$f_v(\texttt{return}, L_v) \triangleq \emptyset$$

$$\mathsf{fv}(e) \triangleq \begin{cases} \{r\} & \textit{if } e = r \\ \emptyset & \textit{if } e = v \\ \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) & \textit{if } (e = e_1 + e_2) \vee (e = e_1 - e_2) \vee (e = e_1 * e_2) \end{cases}$$

Fig. 31. Auxiliary definitions in liveness analysis

10.3 Liveness analysis

The set L_{nl} records the set of registers and memory locations that will not be read before the next assignment.

$$L_{nl} \in \mathcal{P}(Var \cup Reg)$$

The Implementation of the liveness analysis is shown below. It relies on the result of the value analysis.

$$\mathsf{Lv\_Analyzer}(C) \triangleq \mathsf{Lv\_Analyzer}'(C, \mathbb{L}_0, \mathrm{dom}(C), n) \\ \textit{where} \quad \mathbb{L}_0 = \{l \rightsquigarrow ((Var \cup Addr) :: \epsilon) \mid l \in \mathrm{dom}(C)\}\{l' \rightsquigarrow (\emptyset :: \epsilon) \mid C(l') = \_; \texttt{return}\}$$

$$\mathsf{Lv\_Analyzer}'(C, \mathbb{L}, W, n) \triangleq \\ \begin{cases} \mathsf{Lv\_Analyzer}'(C, \mathbb{L}\{l \rightsquigarrow L_{nl}\}, W', n-1) & \textit{if } l \in W, L_{nl} = \bigcap\limits_{l_s \in \mathrm{succ}(C(l))} \mathsf{IN}[\mathbb{L}_l(l_s)], \\ & \quad L'_{nl} = \mathsf{TF}_l(L_{nl}, C(l)), \\ & \quad (L'_{nl} \neq \mathsf{IN}[\mathbb{L}(l)] \implies (W' = (W \backslash \{l\}) \cup \mathsf{pred}(l))), \\ & \quad (L'_{nl} = \mathsf{IN}[\mathbb{L}(l)] \implies W' = (W \backslash \{l\})) \\ \mathbb{L} & \textit{if } W = \emptyset \\ \text{undef} & \textit{otherwise} \end{cases}$$

The liveness analysis on the whole program is defined as the following form. In this work, we only consider the intraprocedural analysis.

$$\mathsf{PLv\_Analyzer}(\pi) \triangleq \{\mathsf{f} \rightsquigarrow \mathsf{Lv\_Analyzer}(C) \mid \pi(\mathsf{f}) = (C, l)\}$$

The transfer function TF_l for basic code blocks in liveness analysis is shown below.

$$\mathsf{TF}_l(L_{nl}, B) \triangleq \begin{cases} f_L(c, L'_{nl}) :: \mathsf{TF}_l(L_{nl}, B') & \textit{if } B = c :: B' \wedge \mathsf{TF}_l(L_{nl}, B') = L'_{nl} :: \_ \\ f_L(B, L_l) :: L_{nl} :: \epsilon & \textit{otherwise} \end{cases}$$

The merging of two abstract interpretations in the liveness analysis is just the intersection of two the sets.

We show the transfer function for each instruction in the following. Some auxiliary definitions are shown in Fig. 31.

- Assignment operation.

$$f_L(r := e, L_{nl}) \triangleq \begin{cases} L_{nl} & \textit{if } r \in L_{nl} \\ (L_{nl} \cup \{r\}) \backslash \mathsf{fv}(e) & \textit{otherwise} \end{cases}$$

- Memory load operation.

$$f_L(r := \mathsf{x}_{o_r}, L_{nl}) \triangleq \begin{cases} L_{nl} & \textit{if } r \in L_{nl} \\ (L_{nl} \cup \{r\}) \backslash \{\mathsf{x}\} & \textit{otherwise} \end{cases}$$

The transfer function for non-atomic read is taken from CompCert. Here, if the register r is dead after the execution of "$r := \mathsf{x}_{\mathsf{na}}$", the abstract interpretation before its execution is still L_{nl}, since the instruction

"$r := \mathsf{x}_{\mathsf{na}}$" is a dead code. The transfer function for atomic read does not have such case, since we do not optimize the atomic memory accesses.
The transfer function for atomic memory accesses show that the dead code elimination in our work supports the following optimization.

$$\begin{array}{l} \mathsf{x}_{\mathsf{na}} := 2; \\ \{\mathsf{x}, r\} \\ r := \mathsf{y}_{\mathsf{acq}}; \\ \{\mathsf{x}\} \\ \mathsf{x}_{\mathsf{na}} := 3; \\ \{\} \end{array} \quad \xRightarrow{DCE} \quad \begin{array}{l} \mathtt{skip}; \\ r := \mathsf{y}_{\mathsf{acq}}; \\ \mathsf{x}_{\mathsf{na}} := 3; \end{array}$$

For the current thread, the memory write "$\mathsf{x}_{\mathsf{na}} := 2$" is not read before the next assignment to the variable x. Thus, it is a dead code for the current thread. Since there is no write release operations after "$\mathsf{x}_{\mathsf{na}} := 2$" before the next assignment to the variable x, there is no requirement for other threads to read such memory write. The correctness of the above optimization can also be achieve by the soundness code transformations as shown below. The reordering shown below is called "roach-motal reordering".

$$\begin{array}{l} \mathsf{x}_{\mathsf{na}} := 2; \\ r := \mathsf{y}_{\mathsf{acq}}; \\ \mathsf{x}_{\mathsf{na}} := 3; \end{array} \rightsquigarrow \begin{array}{l} r := \mathsf{y}_{\mathsf{acq}}; \\ \mathsf{x}_{\mathsf{na}} := 2; \\ \mathsf{x}_{\mathsf{na}} := 3; \end{array} \rightsquigarrow \begin{array}{l} r := \mathsf{y}_{\mathsf{acq}}; \\ \mathtt{skip}; \\ \mathsf{x}_{\mathsf{na}} := 3; \end{array} \rightsquigarrow \begin{array}{l} \mathtt{skip}; \\ r := \mathsf{y}_{\mathsf{acq}}; \\ \mathsf{x}_{\mathsf{na}} := 3; \end{array}$$

- Memory store operation.

$$f_L(\mathsf{x}_{\mathsf{na}} := e, L_{nl}) \triangleq \begin{cases} L_{nl} & \text{if } \mathsf{x} \in L_{nl} \\ (L_{nl} \cup \{\mathsf{x}\}) \backslash \mathsf{fv}(e) & \text{otherwise} \end{cases}$$

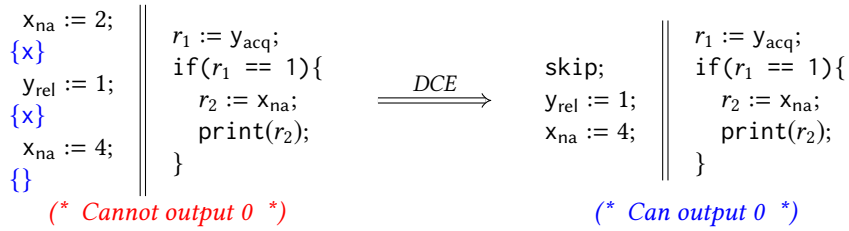
$$f_L(\mathsf{x}_{\mathsf{rlx}} := e, L_{nl}) \triangleq L_{nl} \backslash \mathsf{fv}(e)$$

$$f_L(\mathsf{x}_{\mathsf{rel}} := e, L_{nl}) \triangleq L_{nl} \backslash (Var \cup \mathsf{fv}(e))$$

We focus on the transfer function for atomic write. Doing dead code elimination across the relaxed atomic write is correct. Consider the following dead code elimination optimization.

$$\begin{array}{l} \mathsf{x}_{\mathsf{na}} := 2; \\ \{\mathsf{x}\} \\ \mathsf{y}_{\mathsf{rlx}} := 1; \\ \{\mathsf{x}\} \\ \mathsf{x}_{\mathsf{na}} := 4; \\ \{\} \end{array} \quad \xRightarrow{DCE} \quad \begin{array}{l} \mathtt{skip}; \\ \mathsf{y}_{\mathsf{rlx}} := 1; \\ \mathsf{x}_{\mathsf{na}} := 4; \end{array}$$

For the current thread, the memory write "$\mathsf{x}_{\mathsf{na}} := 2$" is not read before the next assignment to the variable x. Thus, the instruction "$\mathsf{x}_{\mathsf{na}} := 2$" is a dead code for the current thread. Since the execution of the relaxed atomic write "$\mathsf{y}_{\mathsf{rlx}} := 1$" does not release the information of the memory writes of the current thread to other threads, there is no requirement that the other threads must read the memory write "$\mathsf{x}_{\mathsf{na}} := 2$". We can find that "$\mathsf{x}_{\mathsf{na}} := 2$" is also a dead code for other threads.
Doing dead code elimination across the release atomic writes is not correct under any context, since the release write operation may send information about memory writes to other threads. Consider the following optimization if we permit doing dead code elimination across release atomic writes.



- Compare and set operation.

$$f_L(r := \mathsf{CAS}_{o_r,\mathsf{rlx}}(\mathsf{x}, e_r, e_w), L_{nl}) \triangleq (L_{nl} \cup \{r\}) \backslash (\mathsf{fv}(e_r) \cup \mathsf{fv}(e_w))$$

$$f_L(r := \mathsf{CAS}_{o_r,\mathsf{rel}}(e, e_r, e_w), L_{nl}) \triangleq (L_{nl} \cup \{r\}) \backslash (Var \cup \mathsf{fv}(e_r) \cup \mathsf{fv}(e_w))$$

The transfer function for the CAS operation can be viewed as a composition of the transfer functions for the memory load operation and the memory store operation. The dead code elimination optimization shown below is correct.

$x_{na} := 2;$
{x, r}
$r := \mathsf{CAS}_{\mathsf{acq},\mathsf{rlx}}(\mathsf{y}, 0, 1);$
{x}
$x_{na} := 4;$
{}
$\Longrightarrow_{DCE}$
skip;
$r := \mathsf{CAS}_{\mathsf{acq},\mathsf{rlx}}(\mathsf{y}, 0, 1);$
$x_{na} := 4;$

- Release and acquire fence operations.

$$f_L(\texttt{fence-rel}, L_{nl}) \triangleq L_{nl} \backslash Var$$

$$f_L(\texttt{fence-acq}, L_{nl}) \triangleq L_{nl}$$

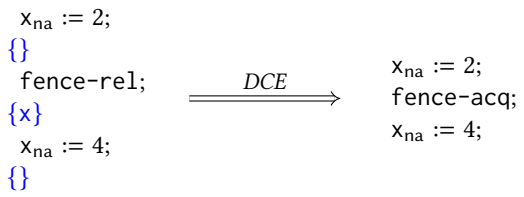
The transfer function for the release fence operation is similar with the transfer function for release atomic write. And the transfer function for the acquire fence operation is similar with the transfer function for acquire atomic read.
We permit the dead code elimination across the acquire fence as the following shown.

$x_{na} := 2;$
{x}
fence-acq;
{x}
$x_{na} := 4;$
{}
$\Longrightarrow_{DCE}$
skip;
fence-acq;
$x_{na} := 4;$

The correctness of the above optimization can also be shown by applying soundness code transformations as the following shown.

$x_{na} := 2;$ fence-acq; $x_{na} := 4;$ $\rightsquigarrow$ fence-acq; $x_{na} := 2;$ $x_{na} := 4;$ $\rightsquigarrow$ fence-acq; skip; $x_{na} := 4;$ $\rightsquigarrow$ skip; fence-acq; $x_{na} := 4;$

Doing dead code elimination across the release fence is forbidden, since the execution of the release fence may send the information of memory writes of the current thread to other threads.



- Unconditional and conditional branch

$$f_L(\texttt{jmp}\ l, L_{nl}) \triangleq L_{nl}$$

$$f_L(\texttt{be}\ e, l_1, l_2, L_{nl}) \triangleq L_{nl} \backslash \mathsf{fv}(e)$$

- Function call, return, system call and SC fence

$$f_L(\texttt{call}(l, l_{ret}), L_{nl}) \triangleq L_{nl} \backslash Var$$

$$f_L(\texttt{print}(e), L_{nl}) \triangleq L_{nl} \backslash (Var \cup \mathsf{fv}(e))$$

$$f_L(\texttt{fence-sc}, L_{nl}) \triangleq L_{nl} \backslash Var$$

Since we consider intraprocedural analysis in this work, the callee may modify memory state arbitrarily. The definition of the system call is the same as `fence-sc`, thus their transfer functions are same.

$$f_L(\texttt{return}, L_{nl}) \triangleq Reg$$

$$\langle e\rangle_{L_a} \triangleq \begin{cases} v & \text{if } e = v \\ r & \text{if } (r,e) \in L_a \wedge e \neq v \\ e_1'+e_2' & \text{if } e = e_1+e_2 \wedge \langle e_1\rangle_{L_a} = e_1' \wedge \langle e_2\rangle_{L_a} = e_2' \\ e_1-e_2 & \text{if } e = e_1-e_2 \wedge \langle e_1\rangle_{L_a} = e_1' \wedge \langle e_2\rangle_{L_a} = e_2' \\ e_1*e_2 & \text{if } e = e_1*e_2 \wedge \langle e_1\rangle_{L_v} = e_1' \wedge \langle e_2\rangle_{L_v} = e_2' \\ e & \text{otherwise} \end{cases}$$

$$L_a \cap \top \triangleq L_a \qquad \top \cap L_a' \triangleq L_a'$$

$$L_a \cap L_a' \triangleq \{(r,e) \mid (r,e) \in L_a \wedge (r,e) \in L_a'\} \cup \{(r,\mathsf{x}) \mid (r,\mathsf{x}) \in L_a \wedge (r,\mathsf{x}) \in L_a'\}$$

$$\mathsf{Kill}(L_a, r_0) \triangleq \{(r,e) \in L_a \mid r_0 \neq r \wedge r_0 \notin \mathsf{fv}(e)\} \cup \{(r,\mathsf{x}) \in L_a \mid r_0 \neq r\}$$

$$\mathsf{Kill}(L_a, \mathsf{x}) \triangleq \{(r,e) \mid (r,e) \in L_a\} \cup \{(r,\mathsf{y}) \in L_a \mid \mathsf{x} \neq \mathsf{y}\}$$

Fig. 32. Auxiliary definitions in available expression analysis

10.4 Available expression analysis

The set L_a records the abstract interpretation of the available expressions in the current state.

$$L_a \in \mathcal{P}((Reg \times Expr) + (Reg \times Var)) \cup \{\top\}$$

The Implementation of the value analysis is shown below.

$$\mathtt{Ave\_Analyzer}(C, l_0) \triangleq \mathtt{Ave\_Analyzer'}(C, \mathbb{L}_0, \mathrm{dom}(C), n)$$
$$\textit{where} \quad \mathbb{L}_0 = \{l \rightsquigarrow (\top :: \epsilon) \mid l \in \mathrm{dom}(C)\}\{l_0 \rightsquigarrow (\emptyset :: \epsilon)\}$$

$$\mathtt{Ave\_Analyzer'}(C, \mathbb{L}, W, n) \triangleq$$
$$\begin{cases} \mathtt{Ave\_Analyzer'}(C, \mathbb{L}\{l \rightsquigarrow L_a\}, W', n-1) & \text{if } l \in W, L_a = \bigcap_{l_p \in \mathrm{pred}(C,l_p)} \mathsf{OUT}[\mathbb{L}(l_p)], \\ & \quad L_a' = \mathsf{TF}_a(L_a, C(l)), \\ & \quad (L_a' \neq \mathsf{OUT}[\mathbb{L}(l)] \implies W' = ((W\backslash\{l\}) \cup \mathrm{succ}(l))), \\ & \quad (L_a' = \mathsf{OUT}[\mathbb{L}(l)] \implies W' = (W\backslash\{l\})) \\ \mathbb{L} & \text{if } W = \emptyset \\ \text{undef} & \text{otherwise} \end{cases}$$

The value analysis on the whole program is defined as the following form. In this work, we only consider the intra-procedural analysis.

$$\mathtt{Ave\_Analyzer}(\pi) \triangleq \{\mathsf{f} \rightsquigarrow \mathtt{Ave\_Analyzer}(C, l_0) \mid \pi(\mathsf{f}) = (C, l_0)\}$$

We define the join of two abstract interpretations and some auxiliary definitions can be found in Fig. 32.
The transfer function TF_a for basic code blocks in the value analysis is defined below.

$$\mathsf{TF}_a(L_a, B) \triangleq \begin{cases} \top :: \epsilon & \text{if } L_a = \top \\ L_a :: \mathsf{TF}_a(L_a', B') & \text{elif } B = c :: B' \wedge L_a' = f_a(c, L_a) \\ L_a :: f_a(B, L_a) :: \epsilon & \text{elif } B \in \{\mathtt{return}, \mathtt{call}(\mathsf{f}, l_{ret}), \mathtt{jmp}\ l, \mathsf{be}\ e, l_1, l_2\} \\ \text{undef} & \text{otherwise} \end{cases}$$

We show the transfer function for each instruction in the following.

- Assignment operation

$$f_a(r := e, L_a) \triangleq \begin{cases} L'_a \cup \{(r', r)\} & \text{if } (r', \langle e \rangle_{L'_a}) \in L'_a \text{ and } r \notin \mathsf{fv}(\langle e \rangle_{L'_a}) \\ L'_a \cup \{(r, \langle e \rangle_{L'_a})\} & \text{if } (r', \langle e \rangle_{L'_a}) \notin L'_a \text{ and } r \notin \mathsf{fv}(\langle e \rangle_{L'_a}) \\ L'_a & \text{otherwise} \end{cases}$$
$$\text{where } L'_a = \mathsf{Kill}(L_a, r)$$

- Memory store operation

$$f_a(\mathsf{x}_{o_w} := e, L_a) \triangleq \mathsf{Kill}(L_a, \mathsf{x})$$

- Memory load operation

$$f_a(r := \mathsf{x}_{\mathsf{na}}, L_a) \triangleq \begin{cases} L'_a \cup \{(r', r)\} & \text{if } (r', \mathsf{x}) \in L'_a \\ L'_a \cup \{(r, \mathsf{x})\} & \text{otherwise} \end{cases}$$
$$\text{where } L'_a = \mathsf{Kill}(L_a, r)$$
$$f_a(r := \mathsf{x}_{\mathsf{rlx}}, L_a) \triangleq \mathsf{Kill}(L_a, r)$$
$$f_a(r := \mathsf{x}_{\mathsf{acq}}, L_a) \triangleq \mathsf{Kill}(\{(r', e') \mid (r', e') \in L_a\}, r)$$

- Compare and set operation

$$f_a(r := \mathsf{CAS}_{\mathsf{rlx}, o_w}(\mathsf{x}, e_r, e_w), L_a) \triangleq \mathsf{Kill}(\mathsf{Kill}(L_a, \mathsf{x}), r)$$
$$f_a(r := \mathsf{CAS}_{\mathsf{acq}, o_w}(\mathsf{x}, e_r, e_w), L_a) \triangleq \mathsf{Kill}(\{(r', e') \mid (r', e') \in L_a\}, r)$$

- Release and acquire fence operations

$$f_a(\texttt{fence-rel}, L_a) \triangleq L_a$$
$$f_a(\texttt{fence-acq}, L_a) \triangleq \{(r, e) \mid (r, e) \in L_a\}$$

- Unconditional and conditional branch

$$f_a(\texttt{jmp}\ l, L_a) \triangleq L_a$$
$$f_a(\texttt{be}\ e, l_1, l_2, L_a) \triangleq L_a$$

- Function call, system call and SC fence

$$f_a(\texttt{call}(l, l_{ret}), L_a) \triangleq \{(r, e) \mid (r, e) \in L_a\}$$
$$f_a(\texttt{print}(e'), L_a) \triangleq \{(r, e) \mid (r, e) \in L_a\}$$
$$f_a(\texttt{fence-sc}, L_a) \triangleq \{(r, e) \mid (r, e) \in L_a\}$$

- Return

$$f_a(\texttt{return}, L_a) \triangleq \emptyset$$

10.5 Constant propagation

- Transformation for an individual instruction.

$$\mathsf{TransI}_c(c, L_\upsilon) \triangleq \begin{cases} r := \upsilon & \textit{if } c = (r := e) \wedge [\![e]\!]_{L_\upsilon} = \upsilon \\ r := \upsilon & \textit{if } c = (r := \mathsf{x_{na}}) \wedge L_\upsilon(\mathsf{x}) = \upsilon \\ c & \textit{otherwise} \end{cases}$$

- Transformation for a basic code block.

$$\mathsf{TransB}_c(B, LB) \triangleq \begin{cases} \mathsf{TransI}_c(c, L_\upsilon), \mathsf{TransB}_c(B', LB') & \textit{if } B = c, B' \wedge LB = L_\upsilon :: LB' \\ B & \textit{otherwise} \end{cases}$$

- Transformation for a code heap.

$$\mathsf{TransC}_c(C, \mathbb{L}) \triangleq \{l \rightsquigarrow \mathsf{TransB}_c(B, \mathbb{L}(l)) \mid C(l) = B\}$$

- Transformation for a program.

$$\mathsf{Translater}_c(\pi, A) \triangleq \{\mathsf{f} \rightsquigarrow \mathsf{TransC}_c(C, \mathbb{L}) \mid \pi(\mathsf{f}) = (C, l_0) \wedge A(\mathsf{f}) = \mathbb{L}\}$$

- Implementation of constant propagation.

$$\mathtt{ConstProp}(\pi, \iota) \triangleq \mathsf{Translater}_c(\pi, A) \quad \textit{where } A = \mathsf{PVal\_Analyzer}(\pi)$$

Our constant propagation optimization supports the following optimizations accross the atomic memory access and the fence operation.

- Optimization across release store.

$$\begin{array}{l} \{\} \\ \mathsf{x_{na}} := 2; \\ \{\mathsf{x} \rightsquigarrow 2\} \\ \mathsf{y_{rel}} := 1; \\ \{\mathsf{x} \rightsquigarrow 2\} \\ r := \mathsf{x_{na}}; \\ \{\mathsf{x} \rightsquigarrow 2, r \rightsquigarrow 2\} \end{array} \xrightarrow{\textit{ConstProp}} \begin{array}{l} \mathsf{x_{na}} := 2; \\ \mathsf{y_{rel}} := 1; \\ r := 2; \end{array}$$

(Performed in LLVM *)*

- Optimization across relaxed store.

$$\begin{array}{l} \{\} \\ \mathsf{x_{na}} := 2; \\ \{\mathsf{x} \rightsquigarrow 2\} \\ \mathsf{y_{rlx}} := 1; \\ \{\mathsf{x} \rightsquigarrow 2\} \\ r := \mathsf{x_{na}}; \\ \{\mathsf{x} \rightsquigarrow 2, r \rightsquigarrow 2\} \end{array} \xrightarrow{\textit{ConstProp}} \begin{array}{l} \mathsf{x_{na}} := 2; \\ \mathsf{y_{rel}} := 1; \\ r := 2; \end{array}$$

(Performed in LLVM *)*

- Optimization across release fence.

$$\begin{array}{l}\{\}\\ \mathsf{x}_{\mathsf{na}} := 2;\\ \{\mathsf{x} \rightsquigarrow 2\}\\ \texttt{fence-rel};\\ \{\mathsf{x} \rightsquigarrow 2\}\\ r := \mathsf{x}_{\mathsf{na}};\\ \{\mathsf{x} \rightsquigarrow 2, r \rightsquigarrow 2\}\end{array} \xRightarrow{\textit{ConstProp}} \begin{array}{l}\mathsf{x}_{\mathsf{na}} := 2;\\ \texttt{fence-rel};\\ r := 2;\end{array}$$

(Performed in LLVM *)*

- Optimize across CAS with relaxed read and release write.

$$\begin{array}{l}\{\}\\ \mathsf{x}_{\mathsf{na}} := 2;\\ \{\mathsf{x} \rightsquigarrow 2\}\\ r_1 := \mathsf{CAS}_{\mathsf{rlx},\mathsf{rel}}(\mathsf{y}, 0, 1);\\ \{\mathsf{x} \rightsquigarrow 2\}\\ r := \mathsf{x}_{\mathsf{na}};\\ \{\mathsf{x} \rightsquigarrow 2, r \rightsquigarrow 2\}\end{array} \xRightarrow{\textit{ConstProp}} \begin{array}{l}\mathsf{x}_{\mathsf{na}} := 2;\\ r_1 := \mathsf{CAS}_{\mathsf{rlx},\mathsf{rel}}(\mathsf{y}, 0, 1);\\ r := 2;\end{array}$$

(Not Performed in LLVM *)*

- Optimization across relaxed read.

$$\begin{array}{l}\{\}\\ \mathsf{x}_{\mathsf{na}} := 2;\\ \{\mathsf{x} \rightsquigarrow 2\}\\ r_1 := \mathsf{y}_{\mathsf{rlx}};\\ \{\mathsf{x} \rightsquigarrow 2\}\\ r := \mathsf{x}_{\mathsf{na}};\\ \{\mathsf{x} \rightsquigarrow 2, r \rightsquigarrow 2\}\end{array} \xRightarrow{\textit{ConstProp}} \begin{array}{l}\mathsf{x}_{\mathsf{na}} := 2;\\ r_1 := \mathsf{y}_{\mathsf{rlx}};\\ r := 2;\end{array}$$

(Performed in LLVM *)*

10.6 Dead code elimination

- Transformation for an individual instruction.

$$\mathsf{Transl}_d(c, L_{nl}) \triangleq \begin{cases} \mathtt{skip} & \textit{if } c = (r := e) \wedge r \in L_{nl} \\ \mathtt{skip} & \textit{if } c = (r := \mathsf{x}_{\mathsf{na}}) \wedge r \in L_{nl} \\ \mathtt{skip} & \textit{if } c = (\mathsf{x}_{\mathsf{na}} := \_) \wedge \mathsf{x} \in L_{nl} \\ c & \textit{otherwise} \end{cases}$$

- Transformation for a basic code block.

$$\mathsf{TransB}_d(B, LB_l) \triangleq \begin{cases} \mathsf{Transl}_d(c, L_{nl}) :: \mathsf{TransB}_d(B', LB'_l) & \textit{if } B = c :: B' \wedge LB_l = L_{nl} :: LB'_l \\ B & \textit{otherwise} \end{cases}$$

- Transformation for a code heap.

$$\mathsf{TransC}_d(C, \mathbb{L}_l) \triangleq \{l \rightsquigarrow \mathsf{TransB}_d(B, \mathbb{L}_l(l)[1\ldots]) \mid C(l) = B \wedge \mathsf{IN}[\mathbb{L}_l(l)] \neq \top\}$$

- Transformation for a program.

$$\mathsf{Translater}_d(\pi, A_l) \triangleq \{\mathsf{f} \rightsquigarrow \mathsf{TransC}(C, LB_l) \mid \pi(\mathsf{f}) = (C, l_0) \wedge A_l(\mathsf{f}) = LB_l\}$$

- Implementation of constant propagation.

$$\mathsf{DCE}(\pi, \iota) \triangleq \mathsf{Translater}_d(\pi, A_l) \quad \textit{where } A_l = \mathsf{PLv\_Analyzer}(\pi)$$

Our dead code elimination optimization supports the following optimizations accross the atomic memory access and the fence operation.

- Optimize across acquire read.

$$\begin{array}{l} \{\mathsf{x}, r\} \\ \mathsf{x}_{\mathsf{na}} := 1; \\ \{\mathsf{x}, r\} \\ r := \mathsf{y}_{\mathsf{acq}}; \\ \{\mathsf{x}\} \\ \mathsf{x}_{\mathsf{na}} := 2; \\ \{\} \end{array} \xrightarrow{DCE} \begin{array}{l} \mathtt{skip}; \\ r := \mathsf{y}_{\mathsf{acq}}; \\ \mathsf{x}_{\mathsf{na}} := 2; \end{array}$$

(Not Performed in LLVM *)*

- Optimize across relaxed read.

$$\begin{array}{l} \{\mathsf{x}, r\} \\ \mathsf{x}_{\mathsf{na}} := 1; \\ \{\mathsf{x}, r\} \\ r := \mathsf{y}_{\mathsf{rlx}}; \\ \{\mathsf{x}\} \\ \mathsf{x}_{\mathsf{na}} := 2; \\ \{\} \end{array} \xrightarrow{DCE} \begin{array}{l} \mathtt{skip}; \\ r := \mathsf{y}_{\mathsf{rlx}}; \\ \mathsf{x}_{\mathsf{na}} := 2; \end{array}$$

(Performed in LLVM *)*

- Optimize across acquire fence.

$$\begin{array}{l}\{x\}\\ x_{na} := 1;\\ \{x\}\\ \texttt{fence-acq};\\ \{x\}\\ x_{na} := 2;\\ \{\}\end{array} \xrightarrow{DCE} \begin{array}{l}\texttt{skip};\\ \texttt{fence-acq};\\ x_{na} := 2;\end{array}$$

(Not Performed in LLVM *)*

- Optimize across CAS with acquire read and relaxed write.

$$\begin{array}{l}\{x, r\}\\ x_{na} := 1;\\ \{x, r\}\\ r := \mathsf{CAS}_{acq,rlx}(y, 0, 1);\\ \{x\}\\ x_{na} := 2;\\ \{\}\end{array} \xrightarrow{DCE} \begin{array}{l}\texttt{skip};\\ r := \mathsf{CAS}_{acq,rlx}(y, 0, 1);\\ x_{na} := 2;\end{array}$$

(Not Performed in LLVM *)*

- Optimize across relaxed write.

$$\begin{array}{l}\{x\}\\ x_{na} := 1;\\ \{x\}\\ y_{rlx} := 1;\\ \{x\}\\ x_{na} := 2;\\ \{\}\end{array} \xrightarrow{DCE} \begin{array}{l}\texttt{skip};\\ y_{rlx} := 1;\\ x_{na} := 2;\end{array}$$

(Performed in LLVM *)*

10.7 Loop invariant code motion

The implement of the *loop invariant code motion* is divided into three steps:

(1) detecting loops and loop invariants in the program;
(2) inserting pre-header nodes to hoist the evaluations of loop invariants before entering loops;
(3) reusing *common subexpression elimination* and *dead code elimination* to eliminate redundant reads and writes.

Note that the *loop invariant code motion* optimization will not move division operations out of loops, since it will make the original safe program abort. We use the following example to show such implementation.

```
while(r1 < 100) {
    r2 := z_na;
    r1 := r1 + 1;
}
```

(1) We find that the instruction "$r_2 := \mathsf{z}_{\mathsf{na}}$" is a loop invariant;
(2) We allocate a new register to save the expression in the loop invariant as the following shown.

```
t := z_na;
while(r1 < 100) {
    r2 := z_na;
    r1 := r1 + 1;
}
```

(3) We use *common subexpression elimination* optimization to eliminate redundant reads.

```
t := z_na;
while(r1 < 100) {
    r2 := t;
    r1 := r1 + 1;
}
```

Detecting loops. To detect the loops in the code, we need to evaluate the dominator of each block block.

$$\textit{(Dominators)} \quad \mathbb{D} \quad \in \quad Lab \rightharpoonup \mathcal{P}(Lab)$$

We use the data flow analysis to evaluate the dominators of each block (where n is a very large constant).

$$\mathtt{Dominator}(C, l_0) \triangleq \mathtt{Dominator'}(C, \mathbb{D}_0, \mathrm{dom}(C), n) \quad \textit{where } \mathbb{D}_0 = \{l \rightsquigarrow \mathrm{dom}(C) \mid l \in \mathrm{dom}(C)\}\{l_0 \rightsquigarrow \emptyset\}$$

$$\mathtt{Dominator'}(C, \mathbb{D}, W, n) \triangleq \begin{cases} \mathtt{Dominator'}(C, \mathbb{D}\{l \rightsquigarrow D'\}, W', n-1) & \textit{if } l \in W, D = \bigcap\limits_{l_p \in \mathsf{pred}(C,l)} (\mathbb{D}(l_p) \cup \{l_p\}), \\ & \quad (D \neq \mathbb{D}(l) \implies W' = ((W \backslash \{l\}) \cup \mathrm{succ}(l))), \\ & \quad (D = \mathbb{D}(l) \implies W' = (W \backslash \{l\})) \\ \mathbb{D} & \textit{if } W = \emptyset \\ \text{undef} & \textit{otherwise} \end{cases}$$

After evaluating the dominators of each block, we can find the loop. l_{entry} and l_{exit} are entry and exit of a loop if l_{exit} points to l_{entry} and l_{entry} dominates l_{exit}. l_{entry} and l_{exit} constructs back edge.

$$\mathsf{back\_edge}(l_{exit}, l_{entry}, \mathbb{D}, C) \triangleq l_{entry} \in \mathrm{succ}(C(l_{exit})) \wedge (l_{entry} \in \mathbb{D}(l_{exit}))$$

$$\begin{array}{rcll} \textit{(LoopInv)} & \mathsf{loop\_inv} & \in & \mathit{List}\ ((\mathit{Exp} \cup \mathit{Var}) \times \mathit{Reg}) \\ \textit{(Loops)} & \mathsf{loops} & \in & \mathcal{P}(\mathit{Lab} \times \mathit{Lab} \times \mathit{LoopInv}) \\ \textit{(ProgLoops)} & \mathsf{loops\_P} & \in & \mathit{Fid} \rightharpoonup \mathit{Loops} \end{array}$$

Fig. 33. The result of detecting loops and loop invariants

The blocks in the loop whose entry and exit are l_{entry} and l_{exit} are evaluated below. `natural_loop` defined below returns the identifiers of blocks in the loop. The function `det_loops` returns the loops in the code C. A block with identifier l is in a loop, whose entry and exit are l_{entry} and l_{exit}, if l_{entry} is the dominator of l and l can reach the exit l_{exit}.

$$\begin{array}{l} \mathsf{Reach}(l, l', C, l_{entry}) \triangleq \exists l_0, \ldots, l_n \in \mathrm{dom}(C).\ (\forall i \in \{1, \ldots, n\}.\ l_i \in \mathrm{succ}(C(l_{i-1})) \wedge l_{i-1} \neq l_{entry}) \\ \qquad\qquad \wedge\ l_0 \in \mathrm{succ}(C(l)) \wedge l' \in \mathrm{succ}(C(l_n)) \\ \mathtt{natural\_loop}(l_{exit}, l_{entry}, C, \mathbb{D}) \triangleq \{l \mid l_{entry} \in \mathbb{D}(l) \wedge \mathsf{Reach}(l, l_{exit}, C, l_{entry})\} \\ \mathtt{det\_loops}(C, l_0) \triangleq \{(l_{entry}, l_{exit}, ls) \mid \mathsf{back\_edge}(l_{exit}, l_{entry}, \mathbb{D}, C) \wedge \\ \qquad\qquad ls \in \mathtt{natural\_loop}(l_{entry}, l_{exit}, C, \mathbb{D}) \wedge l_{entry} \in \mathrm{dom}(C) \wedge l_{exit} \in \mathrm{dom}(C)\} \\ \qquad \textit{where}\ \mathbb{D} = \mathtt{Dominator}(C, l_0) \end{array}$$

Loop invariants. A loop invariant is a non-atomic read of the variable or an evaluation of a expression, whose result is the same on every iteration of the loop. $B[i]$ represents the i-th instruction in the block B. We use $a \in ls$ to represent that a is an element in the list. The evaluation of the expression e is a loop invariant, if the registers in e is not updated in the loop. The reading of a variable x is a loop invariant, if there is no write to x in the loop. The parameter $\mathbb{B}$ in `loop_invB` is the set of blocks in the loop.

$$\begin{array}{l} \mathtt{loop\_invB}(B, \mathbb{B}, RS, \mathsf{loop\_inv}, \iota) \triangleq \\ \quad \begin{cases} \mathtt{loop\_invB}(B', \mathbb{B}, RS \cup \{r'\}, (e, r') :: \mathsf{loop\_inv}, \iota) & \textit{if}\ B = (r := e), B', \\ & \quad (\forall r_0 \in \mathsf{fv}(e).\ \neg(\exists B_0 \in \mathbb{B}, i.\ B_0[i] = (r_0 := \_)), \\ & \quad (e, \_) \notin \mathsf{loop\_inv}, r' \notin RS \\ \mathtt{loop\_invB}(B', \mathbb{B}, RS \cup \{r'\}, (\mathsf{x}, r') :: \mathsf{loop\_inv}, \iota) & \textit{elif}\ B = (r := \mathsf{x}_{\mathsf{na}}), B', \mathsf{x} \notin \iota, \\ & \quad \neg(\exists B_0 \in \mathbb{B}, i.\ B_0[i] = (\mathsf{x}_{o_w} := e)), \\ & \quad (\mathsf{x}, \_) \notin \mathsf{loop\_inv}, r' \notin RS \\ \mathtt{loop\_invB}(B', \mathbb{B}, RS, \mathsf{loop\_inv}, \iota) & \textit{elif}\ B = c :: B' \\ (\mathsf{loop\_inv}, RS) & \textit{otherwise} \end{cases} \\ \mathtt{loop\_invBS}(\mathbb{B}_0, \mathbb{B}, RS, \mathsf{loop\_inv}, \iota) \triangleq \\ \quad \begin{cases} \mathtt{loop\_invBS}(\mathbb{B}'_0, \mathbb{B}, RS', \mathsf{loop\_inv}', \iota) & \textit{if}\ B \in \mathbb{B}_0, \mathbb{B}'_0 = \mathbb{B} \backslash \{B\} \\ & \quad \mathtt{loop\_invB}(B, \mathbb{B}, RS, \mathsf{loop\_inv}, \iota) = (\mathsf{loop\_inv}', RS') \\ (\mathsf{loop\_inv}, RS) & \textit{otherwise} \end{cases} \end{array}$$

loop_invC defined below returns the loop invariants of each loop in the code C.

$$\texttt{loop\_invC}'(lps, RS, C, \iota) \triangleq \begin{cases} \{l_{entry}, l_{exit}, \texttt{loop\_inv}\} \cup \\ \quad \texttt{loop\_invC}'(lps', RS', C, \iota) & \textit{if } lps = ((l_{entry}, l_{exit}, ls) \cup lps'), (l_{entry}, l_{exit}, ls) \notin lps', \\ & \quad \mathbb{B} = \{B \mid C(l) = B \wedge l \in ls\}, \\ & \quad \texttt{loop\_invBS}(\mathbb{B}, \mathbb{B}, RS, \epsilon, \iota) = (\texttt{loop\_inv}, RS') \\ \emptyset & \textit{otherwise} \end{cases}$$

$$\texttt{loop\_invC}(C, l_0, \iota) \triangleq \texttt{loop\_invC}'(lps, \texttt{fv}(C), C, \iota) \quad \textit{where } lps = \texttt{det\_loops}(C, l_0)$$

We define det_loop_inv to evaluate the loop invariants in each function.

$$\texttt{det\_loop\_inv}(\pi, \iota) \triangleq \{\texttt{f} \rightsquigarrow \texttt{loop\_invC}(C, l_0, \iota) \mid \pi(\texttt{f}) = (C, l_0)\}$$

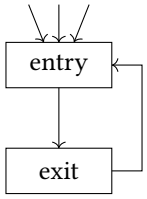
The implementation of detecting loops and loop invariants need to ensure the following property.

Lemma 10.1 (Well-formed detecting loops and loop invariants).

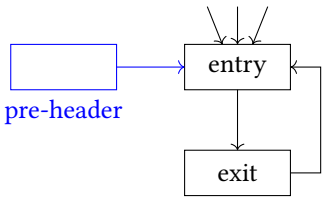
$$\begin{aligned} &\forall \pi, \texttt{loops\_P}, \texttt{f}, l_{entry}, l_{exit}, C, \iota. \\ &\quad \texttt{det\_loop\_inv}(\pi, \iota) = \texttt{loops\_P} \wedge \\ &\quad (l_{entry}, l_{exit}, \texttt{loop\_inv}) \in \texttt{loops\_P}(\texttt{f}) \wedge \pi(\texttt{f}) = (C, \_) \wedge \\ &\quad (\_, r) \in \texttt{loop\_inv} \\ &\Longrightarrow r \notin \texttt{fv}(C) \wedge l_{entry} \in \texttt{dom}(C) \wedge l_{exit} \in \texttt{dom}(C) \wedge \\ &\qquad (\forall (\texttt{x}, \_) \in \texttt{loop\_inv}.\ \texttt{x} \notin \iota) \end{aligned}$$

Proof. The correctness of Lemma 10.1 is straight-forward from the implementation of det_loop_inv. Since, we always allocate a new register to save the result of the loop invariants, we have $r \notin \texttt{fv}(C)$. According to the definition of det_loops, we have $l_{entry} \in \texttt{dom}(C)$ and $l_{exit} \in \texttt{dom}(C)$. According to the definition of loop_invB, since we only view the non-atomic read whose result is the same on every iteration of the loop as the invariant. We have $\forall (\texttt{x}, \_) \in \texttt{loop\_inv}.\ \texttt{x} \notin \iota$. □

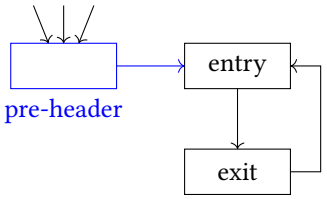
We define the allocation of pre-header in the following. We divide it into two steps. Consider a loop in the following form.



(1) We first allocate a new block according to the loop invariants of such loop as the pre-header of the entry of such loop.



(2) We let the nodes that are not the exit but point to the entry node of the loop point to the pre-header.



We first give the definition of allocating pre-header.

$$\text{alloc\_ph}(\text{loop\_inv}, l_{entry}) \triangleq \begin{cases} r := e, \text{alloc\_ph}(\text{loop\_inv}', l_{entry}) & \textit{if } \text{loop\_inv} = (e, r) :: \text{loop\_inv}' \\ r := \mathsf{x}_{\mathsf{na}}, \text{alloc\_ph}(\text{loop\_inv}', l_{entry}) & \textit{if } \text{loop\_inv} = (\mathsf{x}, r) :: \text{loop\_inv}' \\ \texttt{jmp}\ l_{entry} & \textit{otherwise} \end{cases}$$

$$\text{consInv}(\text{loop\_inv}, B) \triangleq \begin{cases} \text{consInv}(\text{loop\_inv}', (r := e, B)) & \textit{if } \text{loop\_inv} = (e, r) :: \text{loop\_inv}' \\ \text{consInv}(\text{loop\_inv}', (r := \mathsf{x}, B)) & \textit{if } \text{loop\_inv} = (e, \mathsf{x}) :: \text{loop\_inv}' \\ B & \textit{otherwise} \end{cases}$$

We give a mapping that records the pre-header of the entry of each loop.

$$\textit{(PreHeader)} \quad \text{pre-header} \in \textit{Lab} \rightharpoonup \textit{Lab}$$

$$\text{ptB-ph}(B, l_{ph}, l_{entry}) \triangleq \begin{cases} c, \text{ptB-ph}(B', l_{ph}, l_{entry}) & \textit{if } B = c, B' \\ \texttt{jmp}\ l_{ph} & \textit{if } B = \texttt{jmp}\ l_{entry} \\ \texttt{be}\ e, l_{ph}, l_2 & \textit{if } B = \texttt{be}\ e, l_{entry}, l_2 \textit{ and } l_2 \neq l_{entry} \\ \texttt{be}\ e, l_1, l_{ph} & \textit{if } B = \texttt{be}\ e, l_1, l_{entry} \textit{ and } l_1 \neq l_{entry} \\ \texttt{be}\ e, l', l' & \textit{if } B = \texttt{be}\ e, l_{entry}, l_{entry} \\ B & \textit{otherwise} \end{cases}$$

$$\text{ptC-ph}(C, l_{ph}, l_{entry}, \text{loops}) \triangleq \begin{cases} \{l \rightsquigarrow B'\} \cup \text{ptC-ph}(C', l_{ph}, l_{entry}, \text{loops}) & \textit{if } C = \{l \rightsquigarrow B\} \uplus C', (l_{entry}, l, \_) \notin \text{loops}, \\ & \quad B' = \text{ptB-ph}(B, l_{ph}, l_{entry}) \\ \{l \rightsquigarrow B\} \cup \text{ptC-ph}(C', l_{ph}, l_{entry}, \text{loops}) & \textit{elif } C = \{l \rightsquigarrow B\} \uplus C' \\ C & \textit{otherwise} \end{cases}$$

We define the transformation for function in *loop invariant code motion* formally below.

$$\text{TransC}'(C, \text{pre-header}, \text{loops}, \text{loops}_0) \triangleq \begin{cases} \text{TransC}'(C', \text{pre-header}, \text{loops}', \text{loops}_0) & \textit{if } (l_{entry}, l_{exit}, \text{loop\_inv}) \uplus \text{loops}' = \text{loops}, \\ & \quad \text{pre-header}(l_{entry}) = l_{ph}, C(l_{ph}) = B_{ph}, \\ & \quad B' = \text{consInv}(\text{loop\_inv}, B_{ph}) \textit{ and } C' = C\{l \rightsquigarrow B'\} \\ \text{TransC}'(C', \text{pre-header}', \text{loops}', \text{loops}_0) & \textit{if } (l_{entry}, l_{exit}, \text{loop\_inv}) \uplus \text{loops}' = \text{loops}, \\ & \quad \text{pre-header}(l_{entry}) = \bot, l_{ph} \notin \text{dom}(C), \\ & \quad B_{ph} = \text{alloc\_ph}(\text{loop\_inv}, l_{entry}), \\ & \quad \text{pre-header}' = \text{pre-header}\{l_{entry} \rightsquigarrow l_{ph}\} \textit{ and} \\ & \quad C' = \text{ptC-ph}(C, l_{ph}, l_{entry}, \text{loops}_0) \cup \{l_{ph} \rightsquigarrow B_{ph}\} \\ (C, \text{pre-header}) & \textit{otherwise} \end{cases}$$

$$\mathsf{TransC}(C, l_0, \mathsf{loops}) \triangleq \begin{cases} (C', l_0') & \textit{if } (C', \text{pre-header}) = \mathsf{TransC}'(C, \emptyset, \mathsf{loops}, \mathsf{loops}) \textit{ and} \\ & \quad \text{pre-header}(l_0) = l_0' \\ (C', l_0) & \textit{if } (C', \text{pre-header}) = \mathsf{TransC}'(C, \emptyset, \mathsf{loops}, \mathsf{loops}) \textit{ and} \\ & \quad \text{pre-header}(l_0) = \bot \\ \text{undef} & \textit{otherwise} \end{cases}$$

We give the transformation for program in *loop invariant code motion* formally below.

$$\begin{aligned} \mathtt{LInv}(\pi, \iota) \triangleq\ & \{\mathsf{f} \rightsquigarrow (C', l_0') \mid \pi(\mathsf{f}) = (C, l_0) \wedge \mathsf{loops\_P}(l_0) = \mathsf{loops} \wedge \\ & \qquad \mathsf{TransC}(C, l_0, \mathsf{loops}) = (C', l_0')\} \\ & \textit{where } \mathsf{loops\_P} = \mathsf{det\_loop\_inv}(\pi, \iota) \end{aligned}$$

$$\mathtt{LICM} \triangleq \mathtt{LInv} \circ \mathtt{CSE}$$

10.8 Common subexpression elimination

- Transformation for an individual instruction.

$$\mathsf{TransI}_{cse}(c, L_a) \triangleq \begin{cases} r := r' & \textit{if } c = (r := e) \wedge (r', e) \in L_a \\ r := r' & \textit{if } c = (r := \mathsf{x}_{\mathsf{na}}) \wedge (r', \mathsf{x}) \in L_a \\ c & \textit{otherwise} \end{cases}$$

- Transformation for a basic code block.

$$\mathsf{TransB}_{cse}(B, LB) \triangleq \begin{cases} \mathsf{TransI}_{cse}(c, L_a) :: \mathsf{TransB}_{cse}(B', LB') & \textit{if } B = c :: B' \wedge LB = L_a :: LB' \\ B & \textit{otherwise} \end{cases}$$

- Transformation for a code heap.

$$\mathsf{TransC}_{cse}(C, \mathbb{L}) \triangleq \{l \rightsquigarrow \mathsf{TransB}_{cse}(B, \mathbb{L}(l)) \mid C(l) = B\}$$

- Transformation for a program.

$$\mathsf{Translater}_{cse}(\pi, A) \triangleq \{\mathsf{f} \rightsquigarrow \mathsf{TransC}_{cse}(C, \mathbb{L}) \mid \pi(\mathsf{f}) = (C, l') \wedge A(\mathsf{f}) = \mathbb{L}\}$$

- Implementation of constant propagation.

$$\mathsf{CSE}(\pi, \iota) \triangleq \mathsf{Translater}_{cse}(\pi, A) \quad \textit{where } A = \mathtt{Ave\_Analyzer}(\pi)$$

Our common subexpression elimination optimization supports the following optimizations accross the atomic memory access and the fence operation.

- Optimization across release store.

$$\begin{array}{l} \{\} \\ \quad r := \mathsf{x}_{\mathsf{na}}; \\ \{(r, \mathsf{x})\} \\ \quad \mathsf{y}_{\mathsf{rel}} := 1; \\ \{(r, \mathsf{x})\} \\ \quad r' := \mathsf{x}_{\mathsf{na}}; \\ \{(r, \mathsf{x}), (r, r'), (r', \mathsf{x})\} \end{array} \xrightarrow{CSE} \begin{array}{l} r := \mathsf{x}_{\mathsf{na}}; \\ \mathsf{y}_{\mathsf{rel}} := 1; \\ r' := r; \end{array}$$

- Optimization across relaxed store.

$$\begin{array}{l} \{\} \\ \quad r := \mathsf{x}_{\mathsf{na}}; \\ \{(r, \mathsf{x})\} \\ \quad \mathsf{y}_{\mathsf{rlx}} := 1; \\ \{(r, \mathsf{x})\} \\ \quad r' := \mathsf{x}_{\mathsf{na}}; \\ \{(r, \mathsf{x}), (r, r'), (r', \mathsf{x})\} \end{array} \xrightarrow{CSE} \begin{array}{l} r := \mathsf{x}_{\mathsf{na}}; \\ \mathsf{y}_{\mathsf{rlx}} := 1; \\ r' := r; \end{array}$$

- Optimization across release fence.

$$\begin{array}{l}\{\}\\ r := \mathsf{x}_{\mathsf{na}};\\ \{(r,\mathsf{x})\}\\ \texttt{fence-rel};\\ \{(r,\mathsf{x})\}\\ r' := \mathsf{x}_{\mathsf{na}};\\ \{(r,\mathsf{x}),(r,r'),(r',\mathsf{x})\}\end{array} \xrightarrow{CSE} \begin{array}{l} r := \mathsf{x}_{\mathsf{na}};\\ \texttt{fence-rel};\\ r' := r;\end{array}$$

- Optimize across CAS with relaxed read and release write.

$$\begin{array}{l}\{\}\\ r := \mathsf{x}_{\mathsf{na}};\\ \{(r,\mathsf{x})\}\\ r_1 := \mathsf{CAS}_{\mathsf{rlx},\mathsf{rel}}(\mathsf{y},0,1);\\ \{(r,\mathsf{x})\}\\ r' := \mathsf{x}_{\mathsf{na}};\\ \{(r,\mathsf{x}),(r,r'),(r',\mathsf{x})\}\end{array} \xrightarrow{CSE} \begin{array}{l} r := \mathsf{x}_{\mathsf{na}};\\ r_1 := \mathsf{CAS}_{\mathsf{rlx},\mathsf{rel}}(\mathsf{y},0,1);\\ r' := r;\end{array}$$

- Optimization across relaxed read.

$$\begin{array}{l}\{\}\\ r := \mathsf{x}_{\mathsf{na}};\\ \{(r,\mathsf{x})\}\\ r_1 := \mathsf{y}_{\mathsf{rlx}};\\ \{(r,\mathsf{x})\}\\ r' := \mathsf{x}_{\mathsf{na}};\\ \{(r,\mathsf{x}),(r,r'),(r',\mathsf{x})\}\end{array} \xrightarrow{CSE} \begin{array}{l} r := \mathsf{x}_{\mathsf{na}};\\ r_1 := \mathsf{y}_{\mathsf{rlx}};\\ r' := r;\end{array}$$

$$\frac{M_t = M_s \qquad \mathcal{S}_t = \mathcal{S}_s \qquad \lVert M_t \rVert = \mathrm{dom}(\varphi) \qquad (\forall (\mathsf{x}, t) \in \mathrm{dom}(\varphi).\ \varphi(\mathsf{x}, t) = t)}{I_{cp}(\varphi, (\mathcal{S}_t, M_t), (\mathcal{S}_s, M_s))}$$

Fig. 34. Invariant in constant propagation proof

$$\begin{array}{lll}
(R, \mathcal{V}, M) \models \{r \rightsquigarrow v\} & ::= & R(r) = v \\
(R, \mathcal{V}, M) \models \{\mathsf{x} \rightsquigarrow v\} & ::= & \exists t.\ \mathcal{V}.\mathsf{cur}.T_{\mathsf{na}}(\mathsf{x}) = t \wedge \langle \mathsf{x} : v@(_, t], _ \rangle \in M \\
(R, \mathcal{V}, M) \models_\iota L_v & ::= & (\forall r, v.\ L_v(r) = v \implies (R, \mathcal{V}, M) \models \{r \rightsquigarrow v\}) \wedge \\
 & & (\forall \mathsf{x}, v.\ L_v(\mathsf{x}) = v \implies ((R, \mathcal{V}, M) \models \{\mathsf{x} \rightsquigarrow v\} \wedge \mathsf{x} \notin \iota))
\end{array}$$

$$\frac{\begin{array}{c}\mathsf{Val\_Analyzer}(C_s, l_0) = \mathbb{L}_v \qquad \mathsf{TransC}_c(C_s, \mathbb{L}_v) = C_t \\ B_s = C_s(l')[i \ldots] \qquad \mathsf{TransB}_c(B_s, LB_v) = B_t \\ R_t = R_s \qquad (R_s, \mathcal{V}_s, M_s) \models_\iota \mathsf{IN}[LB_v] \\ \forall l_p \in \mathrm{succ}(B_s).\ \mathsf{OUT}[LB_v] \geq \mathsf{IN}[\mathbb{L}_v(l_p)]\end{array}}{\mathcal{V}_s, M_s, \iota \vdash (R_t, B_t, C_t) \sim_{cp} (R_s, B_s, C_s)}$$

$$\frac{K_t = K_s = \epsilon}{\iota \vdash K_t \sim_{cp} K_s} \qquad \frac{\forall \mathcal{V}_s, M_s.\ \mathcal{V}_s, M_s, \iota \vdash (R_t, B_t, C_t) \sim_{cp} (R_s, B_s, C_s) \qquad \iota \vdash K'_t \sim_{cp} K'_s}{\iota \vdash ((R_t, B_t, C_t) :: K'_t) \sim_{cp} ((R_s, B_s, C_s) :: K'_s)}$$

$$\frac{\mathsf{PVal\_Analyzer}(\pi_s) = A \qquad \mathsf{Translater}_c(\pi_s, A) = \pi_t \qquad \mathcal{V}_s, M_s, \iota \vdash (R_t, B_t, C_t) \sim_{cp} (R_s, B_s, C_s) \qquad \iota \vdash K_t \sim_{cp} K_s}{\mathcal{V}_s, M_s, \iota \vdash (R_t, B_t, C_t, K_t, \pi_t) \sim_{cp} (R_s, B_s, C_s, K_s, \pi_s)}$$

$$\frac{\mathcal{V}_s, M_s, \iota \vdash \sigma_t \sim_{cp} \sigma_s \qquad \mathcal{V}_t = \mathcal{V}_s \qquad P_t = P_s}{\iota \vdash (\sigma_t, \mathcal{V}_t, P_t) \sim_{cp} ((\sigma_s, \mathcal{V}_s, P_s), M_s)}$$

$$\frac{\begin{array}{c}I_{cp}(\varphi, (\mathcal{S}_t, M_t), (\mathcal{S}_s, M_s)) \qquad \iota \vdash TS_t \sim_{cp} (TS_s, M_s) \\ (\beta = \circ \wedge \varphi = \varphi') \vee (\beta = \bullet \wedge \varphi \subseteq \varphi') \qquad \lVert TS_t.P_t \rVert \subseteq \mathrm{dom}(\varphi)\end{array}}{\Phi_{cp}(\varphi, \iota, (TS_t, \mathcal{S}_t, M_t), (TS_s, \mathcal{S}_s, M_s), \beta)}$$

Fig. 35. Match state in constant propagation proof

11 CORRECTNESS PROOF OF OPTIMIZERS

In this section, we show the correctness proof of Constant Propagation, Dead Code Elimination, Loop invariant code motion and Common subexpression elimination.

11.1 Correctness proof of Constant Propagation

Invariant in constant propagation proof. We show the invariant I_{cp} for shared resource in Fig. 34.

Match state in constant propagation proof. We define the match state in constant propagation proof in Fig. 35.

Correctness proof of constant propagation optimizer. We present the correctness proof of constant propagation optimizer in the following.

Lemma 11.1 (Well-defined constant propagation).

$$\forall \pi_s, \pi_t, \iota.\ \mathtt{ConstProp}(\pi_s, \iota) = \pi_t \implies I_{cp}, \iota \models \pi_t \preccurlyeq \pi_s$$

Proof. From the premise, we have the following.

$$\mathtt{ConstProp}(\pi_s, \iota) = \pi_t \tag{1}$$

We unfold the proof goal and need to prove that the following subgoals hold.

$$I_{cp}(\iota, \varphi_0, (\mathcal{S}_\perp, M_0), (\mathcal{S}_\perp, M_0)) \tag{g-1}$$

$$\begin{aligned}&\forall \sigma_t, \mathtt{f}.\ \ \mathsf{Init}(\pi_t, \mathtt{f}) = \sigma_t \implies \\ &\quad \exists \sigma_s.\ (\mathsf{Init}(\pi_s, \mathtt{f}) = \sigma_s \wedge \\ &\qquad I_{cp}, \iota \models ((\sigma_t, \mathcal{V}_\perp, \emptyset), \mathcal{S}_\perp, M_0) \preccurlyeq_\varphi^{\circ, \emptyset} ((\sigma_s, \mathcal{V}_\perp, \emptyset), \mathcal{S}_\perp, M_0))\end{aligned} \tag{g-2}$$

The subgoal (g-1) can be proved by definitions directly.
We consider the correctness proof of the subgoal (g-2). We have the following.

$$I_{cp}(\iota, \varphi_0, (\mathcal{S}_\perp, M_0, \mathcal{S}_\perp, M_0)) \tag{2}$$

$$\mathsf{Init}(\pi_t, \mathtt{f}) = \sigma_t \tag{3}$$

We unfold (2) and have that there exist C_t, l_0 and B_t such that:

$$\pi_t(\mathtt{f}) = (C_t, l_0) \wedge C_t(l_0) = B_t \tag{4}$$

$$\sigma_t = (R_\perp, B_t, C_t, \epsilon, \pi_t) \tag{5}$$

We unfold (1) and have that there exists A such that:

$$\mathsf{PVal\_Analyzer}(\pi_s) = A \tag{6}$$

$$\mathsf{Translater}_c(\pi_s, A) = \pi_t \tag{7}$$

From (4) and (7), we have that there exist π_s, B_s and σ_s such that:

$$\pi_s(\mathtt{f}) = (C_s, l_0) \wedge C_s(l_0) = B_s \tag{8}$$

$$\mathsf{TransC}_c(C_s, A(\mathtt{f})) = C_t \tag{9}$$

$$\mathsf{TransB}_c(B_s, A(\mathtt{f})(l_0)) = B_t \tag{10}$$

$$\mathsf{Init}(\pi_s, \mathtt{f}) = \sigma_s \tag{11}$$

$$\sigma_s = (R_\perp, B_s, C_s, \epsilon, \pi_s) \tag{12}$$

From (2), (4), (5), (6), (7), (8), (9), (10) and (12), we prove that the following hold.

$$\Phi_{cp}(\varphi_0, \iota, ((\sigma_t, \mathcal{V}_\perp, \emptyset), \mathcal{S}_\perp, M_0), ((\sigma_s, \mathcal{V}_\perp, \emptyset), \mathcal{S}_\perp, M_0), \circ, \emptyset) \tag{13}$$

By applying Lemma. 11.2 on (13), we prove the following.

$$I_{cp}, \iota \models ((\sigma_t, \mathcal{V}_\perp, \emptyset), \mathcal{S}_\perp, M_0) \preccurlyeq_{\varphi_0}^{\circ, \emptyset} ((\sigma_s, \mathcal{V}_\perp, \emptyset), \mathcal{S}_\perp, M_0)$$

□

Lemma 11.2 (match state implies simulation - constprop).

$$\begin{aligned}&\forall \varphi, \iota, TS_t, \mathcal{S}_t, M_t, TS_s, \mathcal{S}_s, M_s, \beta.\\ &\quad \Phi_{cp}(\varphi, \iota, (TS_t, \mathcal{S}_t, M_t), (TS_s, \mathcal{S}_s, M_s), \beta)\\ &\implies I_{cp}, \iota \models (TS_t, \mathcal{S}_t, M_t) \preccurlyeq_\varphi^{\beta, \emptyset} (TS_s, \mathcal{S}_s, M_s)\end{aligned}$$

Proof. By co-induction. From the premise, we know

$$\Phi_{cp}(\varphi, \iota, (TS_t, \mathcal{S}_t, M_t), (TS_s, \mathcal{S}_s, M_s), \beta) \tag{1}$$

We need to prove that the following hold.

(1) for any TS_t', $\mathcal{S}_t'$, M_t' and te, if

$$\iota \vdash (TS_t, \mathcal{S}_t, M_t) \xrightarrow{te} (TS_t', \mathcal{S}_t', M_t') \tag{2}$$

then, we need to prove that the following hold:

- if $te \in AT$, there exist TS_s', $\mathcal{S}_s'$, M_s' and φ' such that

$$\iota \vdash (TS_s, \mathcal{S}_s, M_s) \xrightarrow{na}{}^{*} \xrightarrow{te} (TS_s', \mathcal{S}_s', M_s') \tag{g1.1}$$
$$\varphi \subseteq \varphi' \wedge I_{cp}(\iota, \varphi', (\mathcal{S}_t', M_t', \mathcal{S}_s', M_s')) \tag{g1.2}$$
$$I_{cp}, \iota \models (TS_t', \mathcal{S}_t', M_t') \preccurlyeq_{\varphi'}^{\circ,\emptyset} (TS_s', \mathcal{S}_s', M_s') \tag{g1.3}$$

  By applying Lemma. 11.3 on (1) and the preserving of the match state Φ_{cp}.

$$\Phi_{cp}(\varphi', \iota, (TS_t', \mathcal{S}_t', M_t'), (TS_s', \mathcal{S}_s', M_s'), \circ) \tag{3}$$

  We prove the subgoal (g1.2) and (g1.3) by co-inductive hypothesis and (3).

- if $te \in NA$, there exist TS_s', $\mathcal{S}_s'$, M_s', and $\mathcal{D}_1$, such that:

$$(TS_t.P, M_t), (TS_t'.P, M_t') \vdash \emptyset \overset{te}{\rightsquigarrow} \mathcal{D}_1 \tag{g2.1}$$
$$\iota \vdash (TS_s, \mathcal{S}_s, M_s, \mathcal{D}_1) \xrightarrow{na} (TS_s', \mathcal{S}_s', M_s', \emptyset) \tag{g2.2}$$
$$I_{cp}, \iota \models (TS_t', \mathcal{S}_t', M_t') \preccurlyeq_{\varphi}^{\bullet,\emptyset} (TS_s', \mathcal{S}_s', M_s') \tag{g2.3}$$

  We consider that, if $te \in \{\mathsf{R}(\mathsf{na}, \mathsf{x}, v), \mathsf{W}(\mathsf{na}, \mathsf{x}, v)\}$, the subgoals (g2.1), (g2.2), (g2.3) can be proved by applying Lemma. 11.4 on (1) and (2). And, if $te = \tau$, the subgoals (g2.1), (g2.2), (g2.3) can be proved by applying Lemma. 11.5 on (1) and (2). We prove the preserving of the match state Φ_{cp}.

$$\Phi_{cp}(\varphi, \iota, (TS_t', \mathcal{S}_t', M_t'), (TS_s', \mathcal{S}_s', M_s'), \bullet) \tag{4}$$

  We prove the subgoal (g2.3) by co-inductive hypothesis and (4).

- if $te \in \{\mathsf{prm}, \mathsf{rsv}, \mathsf{ccl}\}$, the proof is similar with the case that $te \in (Atm \cup \mathsf{out}(v))$. Thus, we omit the proof of these cases.

(2) if $\beta = \circ$, let $\mathbb{S} = (\mathcal{S}_t, M_t, \mathcal{S}_s, M_s)$ and for any φ' and $\mathbb{S}' = (\mathcal{S}_t', M_t', \mathcal{S}_s', M_s')$, if

$$\mathsf{R}(\iota, (\varphi, \mathbb{S}), (\varphi', \mathbb{S}'), TS_t.P, TS_s.P) \wedge I_{cp}(\iota, \varphi', \mathbb{S}') \tag{5}$$

we need to prove the following hold:

$$I_{cp}, \iota \models (TS_t, \mathcal{S}_t', M_t') \preccurlyeq_{\varphi'}^{\circ,\emptyset} (TS_s, \mathcal{S}_s', M_s') \tag{g3.2}$$

By applying Lemma. 11.6 on (1) and (5), we have the following.

$$\Phi_{cp}(\varphi', \iota, (TS_t, \mathcal{S}_t', M_t'), (TS_s, \mathcal{S}_s', M_s'), \circ) \tag{6}$$

We prove the subgoal (g3.2) by co-inductive and (6).

(3) if $\iota \vdash (TS_t, \mathcal{S}_t, M_t) \longrightarrow \textbf{done}$, the proof of such case is straight-foward and we omit the proof details.

(4) if $\iota \vdash (TS_t, \mathcal{S}_t, M_t) \longrightarrow \textbf{abort}$, there exist TS'_s, $\mathcal{S}'_s$ and M'_s such that:

$$\iota \vdash (TS_s, \mathcal{S}_s, M_s) \xrightarrow{na}{}^{*} (TS'_s, \mathcal{S}'_s, M'_s) \tag{7}$$

$$\iota \vdash (TS'_s, \mathcal{S}'_s, M'_s) \longrightarrow \textbf{abort} \tag{8}$$

We finish the proof of such case by applying Lemma. 11.7.

□

Lemma 11.3 (Match state cp preserving - atomic&output).

$$\begin{aligned}
&\forall \varphi, \iota, TS_t, \mathcal{S}_t, M_t, TS_s, \mathcal{S}_s, M_s, \beta, TS'_t, \mathcal{S}'_t, M'_t, te.\\
&\quad \Phi_{cp}(\varphi, \iota, (TS_t, \mathcal{S}_t, M_t), (TS_s, \mathcal{S}_s, M_s), \beta) \wedge\\
&\quad \iota \vdash (TS_t, \mathcal{S}_t, M_t) \xrightarrow{te} (TS'_t, \mathcal{S}'_t, M'_t) \wedge te \in AT\\
&\Longrightarrow \exists TS'_s, \mathcal{S}'_s, M'_s, \varphi'.\\
&\quad \iota \vdash (TS_s, \mathcal{S}_s, M_s) \xrightarrow{te} (TS'_s, \mathcal{S}'_s, M'_s) \wedge\\
&\quad \varphi \subseteq \varphi' \wedge \Phi_{cp}(\varphi', \iota, (TS'_t, \mathcal{S}'_t, M'_t), (TS'_s, \mathcal{S}'_s, M'_s), \circ)
\end{aligned}$$

Lemma 11.4 (Match state cp preserving - non-atomic read/write).

$$\begin{aligned}
&\forall \varphi, \iota, TS_t, \mathcal{S}_t, M_t, TS_s, \mathcal{S}_s, M_s, \beta, TS'_t, \mathcal{S}'_t, M'_t, te.\\
&\quad \Phi_{cp}(\varphi, \iota, (TS_t, \mathcal{S}_t, M_t), (TS_s, \mathcal{S}_s, M_s), \beta) \wedge\\
&\quad \iota \vdash (TS_t, \mathcal{S}_t, M_t) \xrightarrow{te} (TS'_t, \mathcal{S}'_t, M'_t) \wedge te \in \{\mathsf{R}(\mathsf{na}, \mathsf{x}, _), \mathsf{W}(\mathsf{na}, \mathsf{x}, _)\}\\
&\Longrightarrow \exists TS'_s, \mathcal{S}'_s, M'_s.\\
&\quad \iota \vdash (TS_s, \mathcal{S}_s, M_s) \xrightarrow{te} (TS'_s, \mathcal{S}'_s, M'_s) \wedge\\
&\quad \Phi_{cp}(\varphi, \iota, (TS'_t, \mathcal{S}'_t, M'_t), (TS'_s, \mathcal{S}'_s, M'_s), \bullet)
\end{aligned}$$

Lemma 11.5 (Match state cp preserving - tau).

$$\begin{aligned}
&\forall \varphi, \iota, TS_t, \mathcal{S}_t, M_t, TS_s, \mathcal{S}_s, M_s, \beta, TS'_t, \mathcal{S}'_t, M'_t, te.\\
&\quad \Phi_{cp}(\varphi, \iota, (TS_t, \mathcal{S}_t, M_t), (TS_s, \mathcal{S}_s, M_s), \beta) \wedge\\
&\quad \iota \vdash (TS_t, \mathcal{S}_t, M_t) \xrightarrow{\tau} (TS'_t, \mathcal{S}'_t, M'_t)\\
&\Longrightarrow \exists TS'_s, \mathcal{S}'_s, M'_s, te.\\
&\quad \iota \vdash (TS_s, \mathcal{S}_s, M_s) \xrightarrow{te} (TS'_s, \mathcal{S}'_s, M'_s) \wedge te \in \{\tau, \mathsf{R}(\mathsf{na}, \mathsf{x}, _)\} \wedge\\
&\quad \Phi_{cp}(\varphi, \iota, (TS'_t, \mathcal{S}'_t, M'_t), (TS'_s, \mathcal{S}'_s, M'_s), \bullet)
\end{aligned}$$

Lemma 11.6 (Match state cp preserving - rely).

$$\begin{aligned}
&\forall \iota, \varphi, \varphi', TS_t, TS_s, \mathbb{S} = (\mathcal{S}_t, M_t, \mathcal{S}_s, M_s), \mathbb{S}' = (\mathcal{S}'_t, M'_t, \mathcal{S}'_s, M'_s).\\
&\quad \Phi_{cp}(\varphi, \iota, (TS_t, \mathcal{S}_t, M_t), (TS_s, \mathcal{S}_s, M_s), \circ) \wedge\\
&\quad \mathsf{R}(\iota, (\varphi, \mathbb{S}), (\varphi', \mathbb{S}'), TS_t.P, TS_s.P) \wedge I_{cp}(\iota, \varphi', \mathbb{S}')\\
&\Longrightarrow \Phi_{cp}(\varphi', \iota, (TS_t, \mathcal{S}'_t, M'_t), (TS_s, \mathcal{S}'_s, M'_s), \circ)
\end{aligned}$$

Lemma 11.7 (Match state implies abort preserving).

$$\begin{aligned}
&\forall \varphi, \iota, TS_t, \mathcal{S}_t, M_t, TS_s, \mathcal{S}_s, M_s, \beta.\\
&\quad \Phi_{cp}(\varphi, \iota, (TS_t, \mathcal{S}_t, M_t), (TS_s, \mathcal{S}_s, M_s), \beta) \wedge\\
&\quad \iota \vdash (TS_t, \mathcal{S}_t, M_t) \longrightarrow \textbf{abort}\\
&\Longrightarrow \iota \vdash (TS_s, \mathcal{S}_s, M_s) \longrightarrow \textbf{abort}
\end{aligned}$$

$$
\begin{array}{rcl}
\varphi(T_1, T_2) & \triangleq & \forall \mathsf{x}.\ \varphi(\mathsf{x}, T_1(\mathsf{x})) = T_2(\mathsf{x}) \\
\varphi(V_1, V_2) & \triangleq & \varphi(V_1.T_{\mathsf{na}}, V_2.T_{\mathsf{na}}) \wedge \varphi(V_1.T_{\mathsf{rlx}}, V_2.T_{\mathsf{rlx}}) \\
\varphi(m_t, m_s) & \triangleq & m_t.\mathsf{var} = m_s.\mathsf{var} \wedge \varphi(m_t.\mathsf{var}, m_t.\mathsf{to}) = m_s.\mathsf{to} \wedge \\
 & & \varphi(m_t.\mathsf{view}, m_s.\mathsf{view}) \\
\varphi, \iota \vdash M_t \sim M_s & \triangleq & (\forall m_t \in M_t.\ \exists m_s \in M_s.\ \varphi(m_t, m_s)) \wedge \\
 & & \mathsf{dom}(\varphi) = \lfloor\!\lfloor M_t \rfloor\!\rfloor \wedge \mathsf{mon}(\varphi) \wedge (\forall \langle \mathsf{x} : (f, t] \rangle \in M_s.\ \mathsf{x} \in \iota)
\end{array}
$$

Fig. 36. Auxiliary definitions in defining I_{dce}

11.2 Correctness proof of Dead Code Elimination

Invariant in dead code elimination. We instantiate the invariant for shared memory in proof of dead code elimination.

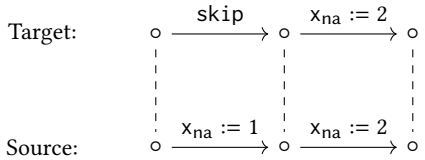
Definition 11.8 (Invariant in dead code elimination proof).

$$
\begin{array}{l}
I_{dce}(\iota, \varphi, (\mathcal{S}_t, M_t, \mathcal{S}_s, M_s)) \triangleq \varphi(\mathcal{S}_t, \mathcal{S}_s) \wedge (\varphi, \iota \vdash M_t \sim M_s) \wedge \\
\quad (\forall \mathsf{x} \notin \iota, t > 0.\ \langle \mathsf{x} : v@(\_, t], \_ \rangle \in M_t. \\
\qquad \Longrightarrow\ \exists \langle \mathsf{x} : v@(f', t'], \_ \rangle \in M_s, t_r. \\
\qquad\quad \varphi(\mathsf{x}, t) = t' \wedge t_r < f' \wedge \\
\qquad\quad (\forall m \in M_s(\mathsf{x}).\ m.\mathsf{to} \leq t'_r \vee t' \leq m.\mathsf{from}))
\end{array}
$$

The most important restriction in I_{dce} is the item (3), which says that each message that does not in the initial state and has a corresponding message in target memory reserves a timestamp interval before it. Consider the following dead code elimination code transformation.

$$
\begin{array}{lcl}
\mathsf{x}_{\mathsf{na}} := 1; & & \mathtt{skip}; \\
\mathsf{x}_{\mathsf{na}} := 2; & \rightsquigarrow & \mathsf{x}_{\mathsf{na}} := 2; \\
r := \mathsf{x}; & & r := \mathsf{x};
\end{array}
$$

If we want to establish the simulation relation for the above program as the following form, we will find that the problem will arise.



Consider that the target thread executes "`skip`" and the source thread executes "$\mathsf{x}_{\mathsf{na}} := 1$" corresponding. The execution of the source thread will generate a message valued 1. Establishing such simulation requires us to find a place to insert such message.

To handler such problem, we need to depict the timestamps reserved for inserting messages generated by the execution of the source thread. Consider that the states of the target memory and the source memory before the target thread executing "`skip`" and the source thread executing "$\mathsf{x}_{\mathsf{na}} := 1$" are shown in Fig. 37. The message valued 5 and the message valued 8 are generated by other threads. Now, we can find the proper place to insert the messages generated by the source thread from the timestamp reserved.

- Consider that the target thread executes "`skip`" and the source thread executes "$\mathsf{x}_{\mathsf{na}} := 1$". We find that the next message of the lastest message viewed by the target thread is the message valued 8. From φ_r, we

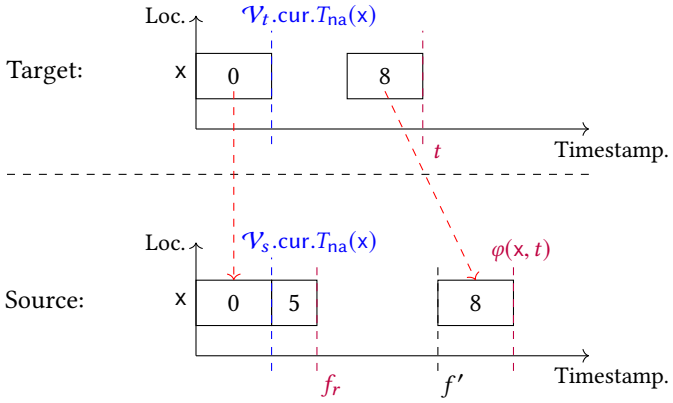


Fig. 37. Timestamps reservation for source writes

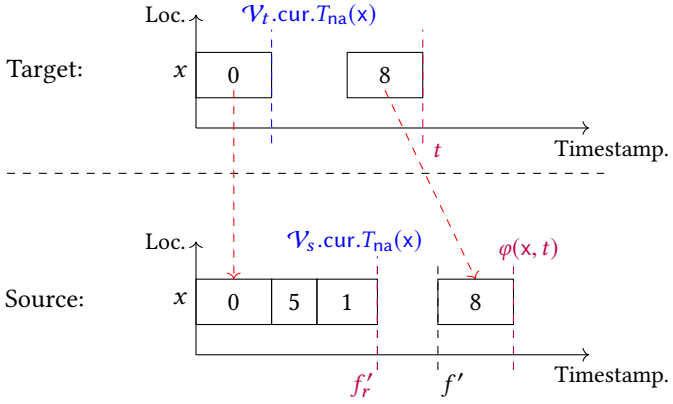


Fig. 38. Function of timestamps reservation for source write - I

know that the range of timestamps from f_r to the lower bound f' of the message valued 8 in the source memory are reserved. Thus, we can insert the message generated by the execution of "$\mathsf{x}_{\mathsf{na}} := 1$" as shown in Fig. 38. For example, the new message can have the form as "$\langle \mathsf{x} : 1@(f_r, (f_r + f')/2], V_\perp \rangle$". Note that the source thread does not need to care about the message valued 5, which is generated by a redundant write of the other thread.

- Then, we consider that the target and source threads both execute "$\mathsf{x}_{\mathsf{na}} := 2$". We find that the next message of the message generated by "$\mathsf{x}_{\mathsf{na}} := 2$" in the target memory is the message valued 8. From the invariant, we know that the range of timestamps from f'_r to the lower bound of the message valued 8 in the source memory are reserved. Thus, we can insert the message of "$\mathsf{x}_{\mathsf{na}} := 2$" as shown in Fig. 39. Inserting message generated by the execution of "$\mathsf{x}_{\mathsf{na}} := 2$" still needs to reserve some timestamps previous it. For example, the new message can have the form as "$\langle \mathsf{x} : 2@((f'_r + t')/2, t'], V_\perp \rangle$", where $t' = (f' + f'_r)/2$. The range $(f'_r, (f'_r + t')/2]$ is reserved for inserting messages.

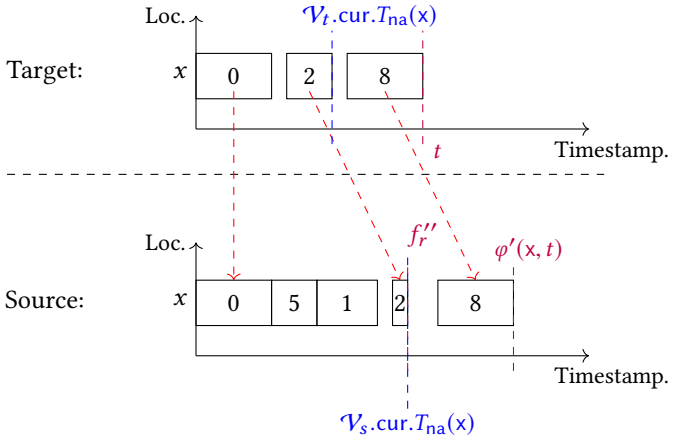


Fig. 39. Function of timestamps reservation for source write - II

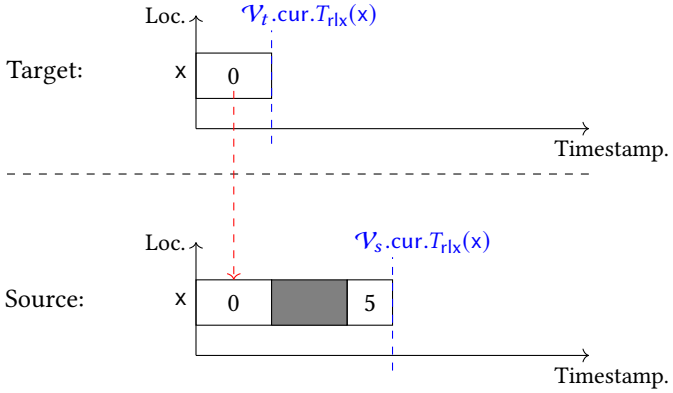


Fig. 40. Necessity to require no reservation on non-atomic locations

We also require that there is no reservations on non-atomic locations in $\varphi, \iota \vdash M_t \sim M_s$ defined in Fig. 36. It forbids that other source threads insert some new messages, that have corresponding messages in the target level, to break the item (1) in the step invariant. Consider the condition in Fig. 40. The execution of the environment (Rely condition) may cancel such reservation and insert new message between the message valued 0 and 5. It will break the item (1) in the step invariant as shown in Fig. 41.

Match state in dead code elimination proof. We define the match state in proving dead code elimination. Some auxiliary definitions that will be used in defining match state are shown in Fig. 42. We define the match state in Fig. 43.

Correctness proof of dead code elimination. To prove that correctness of dead code elimination, we need to prove that the following Lemma. 11.9 holds.

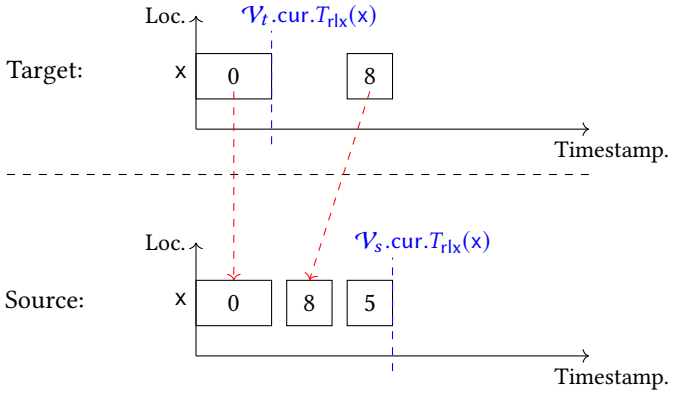


Fig. 41. Necessity to require no reservation on non-atomic locations - II

$$\mathsf{covered\_C}(\mathsf{x}, t', M_s) \triangleq \exists \langle \mathsf{x} : v@(f,t], V\rangle \in M_s.\ t' \in (f,t]$$

$$\begin{aligned}\mathsf{TM}(\varphi, \mathsf{x}, T_t, (T_s, M_s)) \triangleq\ & (\forall (\mathsf{x}, t) \in \mathrm{dom}(\varphi).\ T_t(\mathsf{x}) < t \implies T_s(\mathsf{x}) < \varphi(\mathsf{x}, t)) \wedge \\ & (\exists t'.\ \varphi(\mathsf{x}, T_t(\mathsf{x})) = t' \wedge t' \le T_s(\mathsf{x}) \wedge (\forall t_0 \in (t', T_s(\mathsf{x})].\ \mathsf{covered\_C}(\mathsf{x}, t_0, M_s)))\end{aligned}$$

$$\begin{aligned}&\mathsf{InvView}_{dce}(\varphi, \iota, \mathcal{V}_t, (\mathcal{V}_s, M_s)) \triangleq \\ &\quad (*\ \textit{For current view}\ *) \\ &\quad (\forall \mathsf{x} \in \iota.\ (\varphi(\mathsf{x}, cur_t.T_{\mathsf{na}}(\mathsf{x})) = cur_s.T_{\mathsf{na}}(\mathsf{x}) \wedge \varphi(\mathsf{x}, cur_t.T_{\mathsf{rlx}}(\mathsf{x})) = cur_s.T_{\mathsf{rlx}}(\mathsf{x}))) \wedge \\ &\quad (\forall \mathsf{x} \notin \iota.\ \mathsf{TM}(\varphi, \mathsf{x}, cur_t.T_{\mathsf{rlx}}, (cur_s.T_{\mathsf{rlx}}, M_s))) \wedge \\ &\quad (*\ \textit{For acquire view}\ *) \\ &\quad (\forall \mathsf{x} \in \iota.\ (\varphi(\mathsf{x}, acq_t.T_{\mathsf{na}}(\mathsf{x})) = cur_s.T_{\mathsf{na}}(\mathsf{x}) \wedge \varphi(\mathsf{x}, acq_t.T_{\mathsf{rlx}}(\mathsf{x})) = cur_s.T_{\mathsf{rlx}}(\mathsf{x}))) \wedge \\ &\quad (\forall \mathsf{x} \notin \iota.\ \mathsf{TM}(\varphi, \mathsf{x}, acq_t.T_{\mathsf{rlx}}, (acq_s.T_{\mathsf{rlx}}, M_s))) \wedge \\ &\quad (*\ \textit{For release view}\ *) \\ &\quad (\forall \mathsf{x}.\ \mathsf{x} \in \iota \implies \varphi(rel_t(\mathsf{x}), rel_s(\mathsf{x})) \\ &\ \textit{where}\ \mathcal{V}_t = (cur_t, acq_t, rel_t)\ \textit{and}\ \mathcal{V}_s = (cur_s, acq_s, rel_s)\end{aligned}$$

$$\begin{aligned}&(\varphi, \mathcal{V}_t, \mathcal{V}_s)) \models \{\mathsf{x}\} \triangleq \\ &\quad \varphi(\mathsf{x}, \mathcal{V}_t.\mathsf{cur}.T_{\mathsf{na}}(\mathsf{x})) = \mathcal{V}_s.\mathsf{cur}.T_{\mathsf{na}}(\mathsf{x}) \wedge \varphi(\mathsf{x}, \mathcal{V}_t.\mathsf{acq}.T_{\mathsf{na}}(\mathsf{x})) = \mathcal{V}_s.\mathsf{acq}.T_{\mathsf{na}}(\mathsf{x}) \wedge \\ &\quad \varphi(\mathsf{x}, \mathcal{V}_t.\mathsf{cur}.T_{\mathsf{rlx}}(\mathsf{x})) = \mathcal{V}_s.\mathsf{cur}.T_{\mathsf{rlx}}(\mathsf{x}) \wedge \varphi(\mathsf{x}, \mathcal{V}_t.\mathsf{acq}.T_{\mathsf{rlx}}(\mathsf{x})) = \mathcal{V}_s.\mathsf{acq}.T_{\mathsf{rlx}}(\mathsf{x})\end{aligned}$$

$$(R_t, R_s) \models \{r\} \triangleq R_t(r) = R_s(r)$$

$$\begin{aligned}&(\varphi, (R_t, \mathcal{V}_t), (R_s, \mathcal{V}_s)) \models L_{nl} \triangleq \\ &\quad (\forall \mathsf{x} \notin L_{nl}.\ (\varphi, \mathcal{V}_t, \mathcal{V}_s) \models \{\mathsf{x}\}) \wedge (\forall r \notin L_{nl}.\ (R_t, R_s) \models \{r\})\end{aligned}$$

$$\varphi, \iota \vdash P_t \sim_{dce} P_s \triangleq [P_t]_\iota \approx [P_s]_\iota \wedge \varphi(P_t) = \lVert P_s \rVert \wedge (\forall m \in \widetilde{P_t}.\ m.\mathsf{from} < m.\mathsf{to})$$

Fig. 42. Auxiliary definitions in match state for dead code elimination

$$\mathsf{cur\_acq}(\iota, \varphi, (cur_t, acq_t), (cur_s, acq_s)) \triangleq$$
$$\forall \mathsf{x} \notin \iota.\ (cur_t.T_{\mathsf{rlx}}(\mathsf{x}) < acq_t.T_{\mathsf{rlx}}(\mathsf{x}) \wedge \varphi(\mathsf{x}, acq_t.T_{\mathsf{rlx}}(\mathsf{x})) = acq_s.T_{\mathsf{rlx}}(\mathsf{x})) \vee$$
$$(cur_t.T_{\mathsf{rlx}}(\mathsf{x}) = acq_t.T_{\mathsf{rlx}}(\mathsf{x}) \wedge cur_s.T_{\mathsf{rlx}}(\mathsf{x}) = acq_s.T_{\mathsf{rlx}}(\mathsf{x}))$$

$$\frac{\begin{array}{c}\mathsf{Lv\_Analyzer}(C_s) = \mathbb{L}_l \qquad \mathsf{TransC}_d(C_s, \mathbb{L}_l) = C_t \\ B_s = C_s(l)[i\ldots] \qquad \mathsf{TransB}_d(B_s, \mathbb{L}_l(l)[i{+}1\ldots]) = B_t \\ (\varphi, (R_t, \mathcal{V}_t), (R_s, \mathcal{V}_s)) \models \mathbb{L}_l(l)(i)\end{array}}{\varphi, \iota \vdash ((R_t, B_t, C_t), \mathcal{V}_t) \sim_{dce} ((R_s, B_s, C_s), \mathcal{V}_s)}$$

$$\frac{\begin{array}{c}\forall \mathcal{V}_t, \mathcal{V}_s, \varphi'.\ (\varphi', (R_t, \mathcal{V}_t), (R_s, \mathcal{V}_s)) \models \emptyset \\ \implies \varphi', \iota \vdash ((R_t, B_t, C_t), \mathcal{V}_t) \sim_{dce} ((R_s, B_s, C_s), \mathcal{V}_s)\end{array}}{\varphi, \iota \vdash (R_t, B_t, C_t) :: K'_t \sim_{dce} (R_s, B_s, C_s) :: K'_s}$$

$$\frac{K_t = K_s = \epsilon}{\varphi, \iota \vdash K_t \sim_{dce} K_s}$$

$$\frac{\begin{array}{c}\mathsf{PLv\_Analyzer}(\pi_s) = A \qquad \mathsf{Translater}_d(\pi_s, A) = \pi_t \\ \varphi, \iota \vdash ((R_t, B_t, C_t), \mathcal{V}_t) \sim_{dce} ((R_s, B_s, C_s), \mathcal{V}_s) \qquad \varphi, \iota \vdash K_t \sim_{dce} K_s\end{array}}{\varphi, \iota \vdash ((R_t, B_t, C_t, K_t, \pi_t), \mathcal{V}_t) \sim_{dce} ((R_s, B_s, C_s, K_s, \pi_s), \mathcal{V}_s)}$$

$$\frac{\begin{array}{c}I_{dce}(\varphi', \iota, (\mathcal{S}_t, M_t), (\mathcal{S}_s, M_s)) \qquad [M_t]_\iota \approx [M_s]_\iota \\ \varphi', \iota \vdash (TS_t.\sigma, TS_t.\mathcal{V}_t) \sim_{dce} (TS_s.\sigma, TS_s.\mathcal{V}) \qquad \varphi, \iota \vdash TS_t.P \sim TS_s.P \\ (\beta = \circ \wedge \varphi = \varphi') \vee (\beta = \bullet \wedge \varphi \subseteq \varphi') \\ \mathsf{cur\_acq}(\iota, \varphi, (TS_t.\mathcal{V}.\mathsf{cur}, TS_t.\mathcal{V}.\mathsf{acq}), (TS_s.\mathcal{V}.\mathsf{cur}, TS_s.\mathcal{V}.\mathsf{acq}))\end{array}}{\Phi_{dce}(\varphi, \iota, (TS_t, \mathcal{S}_t, M_t), (TS_s, \mathcal{S}_s, M_s), \beta)}$$

Fig. 43. Match state for dead code elimination

Lemma 11.9 (Well-defined dead code elimination).

$$\forall \pi_s, \pi_t, \iota.\ \mathsf{DCE}(\pi_s, \iota) = \pi_t \implies I_{dce}, \iota \models \pi_t \preccurlyeq \pi_s$$

Proof. Prove by applying Lemma. 11.10. □

Lemma 11.10 (match state implies simulation - dead code elimination).

$$\forall \varphi, \iota, TS_t, \mathcal{S}_t, M_t, TS_s, \mathcal{S}_s, M_s, \beta.$$
$$\Phi_{dce}(\varphi, \iota, (TS_t, \mathcal{S}_t, M_t), (TS_s, \mathcal{S}_s, M_s), \beta)$$
$$\implies I_{dce}, \iota \models (TS_t, \mathcal{S}_t, M_t) \preccurlyeq_\varphi^{\beta, \emptyset} (TS_s, \mathcal{S}_s, M_s)$$

Proof. Prove by cofix and applying Lemma. 11.11, 11.12, 11.13 and 11.14. □

Lemma 11.11 (Match state dce preserving - tau).

$$\forall \varphi, \iota, TS_t, \mathcal{S}_t, M_t, TS_s, \mathcal{S}_s, M_s, \beta, TS'_t, \mathcal{S}'_t, M'_t, rc.$$
$$\Phi_{dce}(\varphi, I_{dce}, \iota, (TS_t, \mathcal{S}_t, M_t), (TS_s, \mathcal{S}_s, M_s), \beta) \wedge$$
$$\iota \vdash (TS_t, \mathcal{S}_t, M_t) \xrightarrow{\tau} (TS'_t, \mathcal{S}'_t, M'_t)$$
$$\implies (\exists TS'_s, \mathcal{S}'_s, M'_s.$$
$$\iota \vdash (TS_s, \mathcal{S}_s, M_s) \xrightarrow{na} (TS'_s, \mathcal{S}'_s, M'_s) \wedge$$
$$\Phi_{dce}(\varphi, \iota, (TS'_t, \mathcal{S}'_t, M'_t), (TS'_s, \mathcal{S}'_s, M'_s), \bullet))) \vee$$
$$\iota \vdash (TS_s, \mathcal{S}_s, M_s) \xrightarrow{na} \textbf{abort}$$

Lemma 11.12 (Match state dce preserving - na).

$$\begin{aligned}
&\forall \varphi, \iota, TS_t, \mathcal{S}_t, M_t, TS_s, \mathcal{S}_s, M_s, \beta, TS'_t, \mathcal{S}'_t, M'_t, te \in \{\mathsf{R}(\mathsf{na}, \mathsf{x}, _), \mathsf{W}(\mathsf{na}, \mathsf{x}, _)\}.\\
&\quad \Phi_{dce}(\varphi, \iota, (TS_t, \mathcal{S}_t, M_t), (TS_s, \mathcal{S}_s, M_s), \beta) \wedge\\
&\quad \iota \vdash (TS_t, \mathcal{S}_t, M_t) \xrightarrow{te} (TS'_t, \mathcal{S}'_t, M'_t)\\
&\implies \exists TS'_s, \mathcal{S}'_s, M'_s.\\
&\qquad \iota \vdash (TS_s, \mathcal{S}_s, M_s) \xrightarrow{te} (TS'_s, \mathcal{S}'_s, M'_s) \wedge\\
&\qquad \Phi_{dce}(\varphi, \iota, (TS'_t, \mathcal{S}'_t, M'_t), (TS'_s, \mathcal{S}'_s, M'_s), \bullet)
\end{aligned}$$

Lemma 11.13 (Match state dce preserving - atm).

$$\begin{aligned}
&\forall \varphi, \iota, TS_t, \mathcal{S}_t, M_t, TS_s, \mathcal{S}_s, M_s, \beta, TS'_t, \mathcal{S}'_t, M'_t, te \in AT.\\
&\quad \Phi_{dce}(\varphi, \iota, (TS_t, \mathcal{S}_t, M_t), (TS_s, \mathcal{S}_s, M_s), \beta) \wedge\\
&\quad \iota \vdash (TS_t, \mathcal{S}_t, M_t) \xrightarrow{te} (TS'_t, \mathcal{S}'_t, M'_t)\\
&\implies \exists TS'_s, \mathcal{S}'_s, M'_s.\\
&\qquad \iota \vdash (TS_s, \mathcal{S}_s, M_s) \xrightarrow{te} (TS'_s, \mathcal{S}'_s, M'_s) \wedge\\
&\qquad \Phi_{dce}(\varphi, \iota, (TS'_t, \mathcal{S}'_t, M'_t), (TS'_s, \mathcal{S}'_s, M'_s), \circ)
\end{aligned}$$

Lemma 11.14 (Match state dce preserving - rely).

$$\begin{aligned}
&\forall \iota, \varphi, \varphi', TS_t, TS_s, \mathbb{S} = (\mathcal{S}_t, M_t, \mathcal{S}_s, M_s), \mathbb{S}' = (\mathcal{S}'_t, M'_t, \mathcal{S}'_s, M'_s).\\
&\quad \Phi_{dce}(\varphi, \iota, (TS_t, \mathcal{S}_t, M_t), (TS_s, \mathcal{S}_s, M_s), \circ) \wedge\\
&\quad \mathsf{R}(\iota, (\varphi, \mathbb{S}), (\varphi', \mathbb{S}'), TS_t.P, TS_s.P) \wedge I_{dce}(\iota, \varphi', \mathbb{S}')\\
&\implies \Phi_{dce}(\varphi', \iota, (TS_t, \mathcal{S}'_t, M'_t), (TS_s, \mathcal{S}'_s, M'_s), \circ)
\end{aligned}$$

$$
\begin{array}{rcl}
V \le V' & \triangleq & V.T_{\mathsf{na}} \le V'.T_{\mathsf{na}} \land V.T_{\mathsf{rlx}} \le V'.T_{\mathsf{rlx}} \\
\mathcal{V} \le \mathcal{V}' & \triangleq & cur \le cur' \land acq \le acq' \land (\forall \mathsf{x}.\ rel(\mathsf{x}) \le rel'(\mathsf{x})) \\
 & & where\ \mathcal{V} = (cur, acq, rel)\ and\ \mathcal{V}' = (cur', acq', rel') \\
M \le M' & \triangleq & M \approx M' \land \\
 & & (\forall \langle \mathsf{x} : v@(f,t], V\rangle \in M.\ \exists V'.\ \langle \mathsf{x} : v@(f,t], V'\rangle \in M' \land V \le V')
\end{array}
$$

$$
\frac{M_s \le M_t \quad \mathcal{S}_s \le \mathcal{S}_t \quad [\![M_t]\!] = \mathrm{dom}(\varphi) \quad (\forall (\mathsf{x}, t) \in \mathrm{dom}(\varphi).\ \varphi(\mathsf{x}, t) = t)}{I_{licm}(\iota, \varphi, (\mathcal{S}_t, M_t, \mathcal{S}_s, M_s))}
$$

Fig. 44. Invariant in loop invariant code motion

$$
\mathsf{wdph}(B_t, \mathsf{f}_{entry}, C_s, \iota) \ ::= \ \begin{cases}
\mathsf{wdph}(B_t', \mathsf{f}_{entry}, C_s, \iota) & if\ B_t = r := e, B_t'\ and\ r \notin \mathsf{fv}(C_s) \\
\mathsf{wdph}(B_t', \mathsf{f}_{entry}, C_s, \iota) & if\ B_t = r := \mathsf{x}_{\mathsf{na}}, B_t'\ and \\
 & \quad r \notin \mathsf{fv}(C_s)\ and\ \iota(\mathsf{x}) = \mathsf{na} \\
\mathsf{true} & if\ B_t = \mathtt{jmp}\ \mathsf{f}_{entry} \\
\mathsf{false} & otherwise
\end{cases}
$$

$$
\frac{}{\mathsf{ptB\_ph\_rel}((c, B_t), (c, B_s), \mathsf{pre\text{-}header})} \qquad \frac{\mathsf{pre\text{-}header}(\mathsf{f}) = \mathsf{f}'}{\mathsf{ptB\_ph\_rel}(\mathtt{jmp}\ \mathsf{f}', \mathtt{jmp}\ \mathsf{f}, \mathsf{pre\text{-}header})}
$$

$$
\frac{\mathsf{pre\text{-}header}(\mathsf{f}_1) = \mathsf{f}_1'}{\mathsf{ptB\_ph\_rel}((\mathtt{be}\ e, \mathsf{f}_1, \mathsf{f}_2), (\mathtt{be}\ e, \mathsf{f}_1', \mathsf{f}_2), \mathsf{pre\text{-}header})} \qquad \frac{\mathsf{pre\text{-}header}(\mathsf{f}_2) = \mathsf{f}_2'}{\mathsf{ptB\_ph\_rel}((\mathtt{be}\ e, \mathsf{f}_1, \mathsf{f}_2), (\mathtt{be}\ e, \mathsf{f}_1, \mathsf{f}_2'), \mathsf{pre\text{-}header})}
$$

$$
\frac{\mathsf{pre\text{-}header}(\mathsf{f}_1) = \mathsf{f}_1' \quad \mathsf{pre\text{-}header}(\mathsf{f}_2) = \mathsf{f}_2'}{\mathsf{ptB\_ph\_rel}((\mathtt{be}\ e, \mathsf{f}_1, \mathsf{f}_2), (\mathtt{be}\ e, \mathsf{f}_1', \mathsf{f}_2'), \mathsf{pre\text{-}header})}
$$

Fig. 45. Auxiliary definitions in the match state of loop invariant code motion proof

11.3 Correctness proof of Loop Invariant Code Motion

Invariant in loop invariant code motion. We show the invariant I_{licm} for shared resource in Fig. 44.

Match state in loop invariant code motion. We define the match state in loop invariant code motion in Fig. 46. Some auxiliary definitions in defining loop invariant code motion are shown in Fig. 45.

Correctness proof of loop invariant code motion. We present the correctness proof of loop invariant code motion in the following.

Lemma 11.15 (Well-defined loop invariant code motion).

$$
\forall \pi_s, \pi_t, \iota.\ \mathsf{Trans\_licm}(\pi_s, \iota) = \pi_t \implies I_{licm}, \iota \models \pi_t \preccurlyeq \pi_s
$$

Proof. From the premises, we have the following.

$$
\mathsf{Trans\_licm}(\pi_s, \iota) = \pi_t \tag{1}
$$

$$\frac{\mathsf{TransC}'(C_s, \emptyset, \mathsf{loops}, \mathsf{loops}) = (C_t, \text{pre-header}) \quad \mathsf{loops\_P}(\mathsf{f}) = \mathsf{loops} \quad \forall r \in \mathsf{fv}(C_s).\ R_t(r) = R_s(r) \quad \mathsf{fv}(B_s) \subseteq \mathsf{fv}(C_s) \quad B_t = B_s \vee \mathsf{ptB\_ph\_rel}(B_t, B_s, \text{pre-header})}{\mathsf{loops\_P}, \iota \vdash (R_t, B_t, C_t, \pi_t) \sim_{licm} (R_s, B_s, C_s, \pi_s)}$$

$$\frac{\mathsf{TransC}'(C_s, \emptyset, \mathsf{loops}, \mathsf{loops}) = (C_t, \text{pre-header}) \quad \mathsf{loops\_P}(\mathsf{f}) = \mathsf{loops} \quad \forall r \in \mathsf{fv}(C_s).\ R_t(r) = R_s(r) \quad \mathsf{wdph}(B_t, \mathsf{f}_{entry}, C_s, \iota) \quad C_s(\mathsf{f}_{entry}) = B_s}{\mathsf{loops\_P}, \iota \vdash (R_t, B_t, C_t, \pi_t) \sim_{licm} (R_s, B_s, C_s, \pi_s)}$$

$$\frac{K_t = K_s = \emptyset}{\mathsf{loops\_P}, \iota \vdash (K_t, \pi_t) \sim_{licm} (K_s, \pi_s)} \qquad \frac{K_t = (R_t, B_t, C_t) :: K_t' \quad K_s = (R_s, B_s, C_s) :: K_s' \quad \mathsf{TransC}'(C_s, \emptyset, \mathsf{loops}, \mathsf{loops}) = (C_t, \text{pre-header}) \quad \mathsf{loops\_P}(\mathsf{f}) = \mathsf{loops} \quad \forall r \in \mathsf{fv}(C_s).\ R_t(r) = R_s(r) \quad \mathsf{fv}(B_s) \subseteq \mathsf{fv}(C_s) \quad B_t = B_s \vee \mathsf{ptB\_ph\_rel}(B_t, B_s, \text{pre-header}) \quad \mathsf{loops}, \iota \vdash (K_t', \pi_t) \sim_{licm} (K_s', \pi_s)}{\mathsf{loops\_P}, \iota \vdash (K_t, \pi_t) \sim_{licm} (K_s, \pi_s)}$$

$$\frac{\mathsf{Trans\_licm}(\pi_s, \iota) = \pi_t \quad \mathsf{det\_loop\_inv}(\pi_s, \iota) = \mathsf{loops\_P} \quad \mathsf{loops\_P}, \iota \vdash (R_t, B_t, C_t, \pi_t) \sim_{licm} (R_s, B_s, C_s, \pi_s) \quad \mathsf{loops\_P}, \iota \vdash (K_t, \pi_t) \sim_{licm} (K_s, \pi_s)}{\iota \vdash (R_t, B_t, C_t, K_t, \pi_t) \sim_{licm} (R_s, B_s, C_s, K_s, \pi_s)}$$

$$\frac{\iota \vdash \sigma_t \sim_{licm} \sigma_s \quad \mathcal{V}_s \leq \mathcal{V}_t \quad P_s = P_t}{\iota \vdash (\sigma_t, \mathcal{V}_t, P_t) \sim_{licm} (\sigma_s, \mathcal{V}_s, P_s)}$$

$$\frac{I_{licm}(\iota, \varphi', (\mathcal{S}_t, M_t, \mathcal{S}_s, M_s)) \quad \iota \vdash TS_t \sim_{licm} TS_s \quad (\beta = \circ \wedge \varphi = \varphi') \vee (\beta = \bullet \wedge \varphi \subseteq \varphi') \quad [\![TS_t.P]\!] \subseteq \mathrm{dom}(\varphi)}{\Phi_{licm}(\varphi, \iota, (TS_t, \mathcal{S}_t, M_t), (TS_s, \mathcal{S}_s, M_s), \beta)}$$

Fig. 46. Match state in loop invariant code motion proof

We need to prove the following.

$$I_{licm}, \iota \models \pi_t \preccurlyeq \pi_s \tag{g}$$

We unfold (g) and need to prove the following.

$$I_{licm}(\iota, \varphi_0, (\mathcal{S}_\perp, M_0, \mathcal{S}_\perp, M_0)) \tag{g1}$$

$$\forall \sigma_t, \mathsf{f}.\ \mathsf{Init}(\pi_t, \mathsf{f}) = \sigma_t \implies \exists \sigma_s.\ \mathsf{Init}(\pi_s, \mathsf{f}) = \sigma_s \wedge I_{licm}, \iota \models ((\sigma_t, \mathcal{V}_\perp, \emptyset), \mathcal{S}_\perp, M_0) \preccurlyeq_\varphi^{\circ, \emptyset} ((\sigma_s, \mathcal{V}_\perp, \emptyset), \mathcal{S}_\perp, M_0) \tag{g2}$$

The goal (g1) can be proved by definitions directly.
We focus on the correctness proof of (g2). We have the following assumptions.

$$I_{licm}(\iota, \varphi_0, (\mathcal{S}_\perp, M_0, \mathcal{S}_\perp, M_0)) \tag{2}$$

$$\mathsf{Init}(\pi_t, \mathsf{f}) = \sigma_t \tag{3}$$

By applying Lemma. 11.16 on (3), (2) and (1), we have that there exists σ_s such that:

$$\mathsf{Init}(\pi_s, \mathtt{f}) = \sigma_s \tag{4}$$
$$\Phi_{licm}(\varphi_0, \iota, ((\sigma_t, \mathcal{V}_\perp, \emptyset), \mathcal{S}_\perp, M_0), ((\sigma_s, \mathcal{V}_\perp, \emptyset), \mathcal{S}_\perp, M_0), \circ, \emptyset) \tag{5}$$

By applying Lemma. 11.17 on (5), we have the following.

$$I_{licm}, \iota \models ((\sigma_t, \mathcal{V}_\perp, \emptyset), \mathcal{S}_\perp, M_0) \preccurlyeq^{\circ,\emptyset}_{\varphi_0} ((\sigma_s, \mathcal{V}_\perp, \emptyset), \mathcal{S}_\perp, M_0) \tag{6}$$

From (4) and (6), we finish the proof. □

Lemma 11.16 (match state holding in initial state - licm).

$$\begin{aligned}
&\forall \pi_t, \mathtt{f}, \sigma_t, \varphi, \iota, \mathcal{S}_t, M_t, \mathcal{S}_s, M_s.\\
&\quad \mathsf{Init}(\pi_t, \mathtt{f}) = \sigma_t \wedge I_{licm}(\iota, \varphi, (\mathcal{S}_t, M_t, \mathcal{S}_s, M_s)) \wedge\\
&\quad \mathsf{Trans_licm}(\pi_s, \mathtt{f}) = \pi_t\\
&\implies \exists \sigma_s.\ \mathsf{Init}(\pi_s, \mathtt{f}) = \sigma_s \wedge\\
&\qquad \Phi_{licm}(\varphi, \iota, ((\sigma_t, \mathcal{V}_\perp, \emptyset), \mathcal{S}_t, M_t), ((\sigma_s, \mathcal{V}_\perp, \emptyset), \mathcal{S}_s, M_s), \circ)
\end{aligned}$$

Proof. From the premises, we have the following.

$$\mathsf{Init}(\pi_t, \mathtt{f}) = \sigma_t \tag{1}$$
$$I_{licm}(\iota, \varphi, (\mathcal{S}_t, M_t, \mathcal{S}_s, M_s)) \tag{2}$$
$$\mathsf{Trans\_licm}(\pi_s, \iota) = \pi_t \tag{3}$$

We unfold (1) and have that there exist C_t, B_t and $\mathtt{f}_t$ such that:

$$\sigma_t = (R_\perp, B_t, C_t, \epsilon, \pi_t) \tag{1.1}$$
$$\pi_t(\mathtt{f}) = (C_t, \mathtt{f}_t) \tag{1.2}$$
$$C_t(\mathtt{f}_t) = B_t \tag{1.3}$$

We unfold (3) and have that there exists loops_P such that:

$$\mathsf{loops\_P} = \mathsf{det\_loop\_inv}(\pi_s, \iota) \tag{3.1}$$

$$\begin{aligned}
&\forall \mathtt{f}, C_t, \mathtt{f}_t.\ \pi_t(\mathtt{f}) = (C_t, \mathtt{f}_t) \implies\\
&\quad \exists C_s, \mathtt{f}_s, \mathsf{loops}.\\
&\qquad \mathsf{loops_P}(\mathtt{f}) = \mathsf{loops} \wedge \pi_s(\mathtt{f}) = (C_s, \mathtt{f}_s) \wedge\\
&\qquad \mathsf{TransC}(C_s, \mathtt{f}_s, \mathsf{loops}) = (C_t, \mathtt{f}_t)
\end{aligned} \tag{3.2}$$

We apply (3.2) on (1.2) and have that there exist C_s, $\mathtt{f}_s$ and loops such that:

$$\mathsf{loops\_P}(\mathtt{f}) = \mathsf{loops} \tag{4}$$
$$\pi_s(\mathtt{f}) = (C_s, \mathtt{f}_s) \tag{5}$$
$$\mathsf{TransC}(C_s, \mathtt{f}_s, \mathsf{loops}) = (C_t, \mathtt{f}_t) \tag{6}$$

We unfold (6) and have that there exists pre-header such that:

$$\mathsf{TransC}'(C_s, \emptyset, \mathsf{loops}, \mathsf{loops}) = (C_t, \text{pre-header}) \tag{7}$$

We discuss whether $\mathtt{f}_s$ is in the domain of pre-header.

- We first consider that f_s is in the domain of pre-header.

$$\text{pre-header}(\mathsf{f}_s) = \mathsf{f}_t \tag{8}$$

From Lemma. 10.1, we have the following.

$$\begin{aligned}&\forall(l_{entry}, l_{exit}, \text{loop\_inv}) \in \text{loops}, (\_, r) \in \text{loop\_inv}.\\ &\quad r \notin \text{fv}(C_s) \wedge l_{entry} \in \text{dom}(C_s) \wedge l_{exit} \in \text{dom}(C_s) \wedge\\ &\quad (\forall(\mathsf{x}, \_) \in \text{loop\_inv}.\ \mathsf{x} \notin \iota)\end{aligned} \tag{9}$$

By applying Lemma. 11.18 on (8), (1.3) (9) and (7), we have that there exists B_s such that:

$$C_s(\mathsf{f}_s) = B_s \tag{8.1}$$

$$\text{wdph}(B_t, \mathsf{f}_s, C_s, \iota) \tag{8.2}$$

From (5) and (8.1), we have that there exists σ_s such that:

$$\sigma_s = (R_\perp, B_s, C_s, \epsilon, \pi_s) \tag{10}$$

$$\text{Init}(\pi_s, \mathsf{f}) = \sigma_s \tag{11}$$

We focus on the proof of the match state holding.

$$\Phi_{licm}(\varphi, \iota, ((\sigma_t, \mathcal{V}_\perp, \emptyset), \mathcal{S}_t, M_t), ((\sigma_s, \mathcal{V}_\perp, \emptyset), \mathcal{S}_s, M_s), \circ) \tag{g1}$$

We unfold (g1) and we need to prove that the following hold.

$$I_{licm}(\iota, \varphi, (\mathcal{S}_t, M_t, \mathcal{S}_s, M_s)) \tag{g1.1}$$

$$((\sigma_t, \mathcal{V}_\perp, \emptyset), \mathcal{S}_t, M_t) \sim_{licm} ((\sigma_s, \mathcal{V}_\perp, \emptyset), \mathcal{S}_s, M_s) \tag{g1.2}$$

From (2), we prove (g1.1). We unfold (g1.2) and we need to prove the following.

$$\text{Trans\_licm}(\pi_s, \iota) = \pi_t \tag{g1.2.1}$$

$$\text{det\_loop\_inv}(\pi_s, \iota) = \text{loops\_P} \tag{g1.2.2}$$

$$\text{loops\_P} \vdash (R_\perp, B_t, C_t, \pi_t) \sim_{licm} (R_\perp, B_s, C_s, \pi_s) \tag{g1.2.3}$$

From (3), we prove (g1.2.1). From (3.1), we prove (g1.2.2). From (7), (4), (8.2) and (8.1), we prove (g1.2.3).

- Then, we consider that f_s is not in the domain of pre-header.

$$\mathsf{f}_s = \mathsf{f}_t \tag{12}$$

By applying Lemma. 11.19 on (1.3) and (7), we have that there exists B_s such that:

$$C_s(\mathsf{f}_s) = B_s \tag{12.1}$$

$$(B_t = B_s \vee \text{ptB\_ph\_rel}(B_t, B_s, \text{pre-header})) \tag{12.2}$$

From (5) and (12.1), we have that there exists σ_s such that:

$$\sigma_s = (R_\perp, B_s, C_s, \epsilon, \pi_s) \tag{13}$$

$$\text{Init}(\pi_s, \mathsf{f}) = \sigma_s \tag{14}$$

We focus on the proof of the match state holding.

$$\Phi_{licm}(\varphi, \iota, ((\sigma_t, \mathcal{V}_\perp, \emptyset), \mathcal{S}_t, M_t), ((\sigma_s, \mathcal{V}_\perp, \emptyset), \mathcal{S}_s, M_s), \circ) \tag{g2}$$

We unfold (g2) and we need to prove that the following hold mainly.

$$I_{licm}(\iota, \varphi, (\mathcal{S}_t, M_t, \mathcal{S}_s, M_s)) \tag{g2.1}$$

$$\iota \vdash ((\sigma_t, \mathcal{V}_\perp, \emptyset), \mathcal{S}_t, M_t) \sim_{licm} ((\sigma_s, \mathcal{V}_\perp, \emptyset), \mathcal{S}_s, M_s) \tag{g2.2}$$

From (2), we prove (g2.1). We unfold (g2.2) and we need to prove the following.

$$\mathsf{Trans\_licm}(\pi_s, \iota) = \pi_t \tag{g2.2.1}$$
$$\mathsf{det\_loop\_inv}(\pi_s, \iota) = \mathsf{loops\_P} \tag{g2.2.2}$$
$$\mathsf{loops\_P} \vdash (R_\perp, B_t, C_t, \pi_t) \sim_{licm} (R_\perp, B_s, C_s, \pi_s) \tag{g2.2.3}$$

From (3), we prove (g1.2.1). From (3.1), we prove (g1.2.2). From (7), (4), (12.1) and (12.2), we prove (g2.2.3). □

Lemma 11.17 (match state implies simulation - licm).

$$\begin{aligned}&\forall \varphi, \iota, TS_t, \mathcal{S}_t, M_t, TS_s, \mathcal{S}_s, M_s, \beta.\\ &\quad \Phi_{licm}(\varphi, \iota, (TS_t, \mathcal{S}_t, M_t), (TS_s, \mathcal{S}_s, M_s), \beta)\\ &\implies I_{licm}, \iota \models (TS_t, \mathcal{S}_t, M_t) \preccurlyeq_\varphi^{\beta,\emptyset} (TS_s, \mathcal{S}_s, M_s)\end{aligned}$$

Proof. By co-fix. From the premises, we have the following.

$$\Phi_{licm}(\varphi, \iota, (TS_t, \mathcal{S}_t, M_t), (TS_s, \mathcal{S}_s, M_s), \beta) \tag{1}$$

We need to prove the following.

$$I_{licm}, \iota \models (TS_t, \mathcal{S}_t, M_t) \preccurlyeq_\varphi^{\beta,\emptyset} (TS_s, \mathcal{S}_s, M_s) \tag{g}$$

We unfold (g) and need to prove the following.

- The invariant between the target thread and source thread configurations holds.

$$\mathsf{SI}(\iota, \varphi, (TS_t, M_t), (TS_s, M_s), \emptyset) \tag{g1}$$

  We prove (g1) by applying Lemma. 11.20.
- for any TS_t', $\mathcal{S}_t'$, M_t' and te, if

$$\iota \vdash (TS_t, \mathcal{S}_t, M_t) \xrightarrow{te} (TS_t', \mathcal{S}_t', M_t') \tag{2}$$

  We need to prove the following.
  – if $te \in AT$, we need to prove that there exist TS_s', $\mathcal{S}_s'$, M_s' and φ' such that:

$$\iota \vdash (TS_s, \mathcal{S}_s, M_s) \xrightarrow{na}{}^{*} \xrightarrow{te} (TS_s', \mathcal{S}_s', M_s') \tag{g2.1}$$
$$\varphi \subseteq \varphi' \wedge I_{licm}(\iota, \varphi', (\mathcal{S}_t', M_t', \mathcal{S}_s', M_s')) \tag{g2.2}$$
$$I_{licm}, \iota \models (TS_t', \mathcal{S}_t', M_t') \preccurlyeq_{\varphi'}^{\circ,\emptyset} (TS_s'', \mathcal{S}_s'', M_s'') \tag{g2.3}$$

    We finish the proof by applying Lemma. 11.21 on (1) and (2) and from co-inductive hypothesis.
  – if $te \in NA$, there exist TS_s', $\mathcal{S}_s'$, M_s', and $\mathcal{D}_1$, such that:

$$(TS_t.P, M_t), (TS_t'.P, M_t') \vdash \emptyset \overset{te}{\rightsquigarrow} \mathcal{D}_1 \tag{g3.1}$$
$$\iota \vdash (TS_s, \mathcal{S}_s, M_s, \mathcal{D}_1) \xrightarrow{na}{}^{*} (TS_s', \mathcal{S}_s', M_s', \emptyset) \tag{g3.2}$$
$$I_{licm}, \iota \models (TS_t', \mathcal{S}_t', M_t') \preccurlyeq_\varphi^{\bullet,\emptyset} (TS_s', \mathcal{S}_s', M_s') \tag{g3.3}$$

    We finish the proof from Lemma. 11.22 and co-inductive hypothesis.
  – The case that $te \in PRC$ is simpler. Thus, we omit the proof details.

– If $\beta = \circ$, let $\mathbb{S} = (\mathcal{S}_t, M_t, \mathcal{S}_s, M_s)$ and we need to prove that for any φ' and $\mathbb{S}' = (\mathcal{S}'_t, M'_t, \mathcal{S}'_s, M'_s)$, if

$$\mathsf{R}(\iota, (\varphi, \mathbb{S}), (\varphi', \mathbb{S}'), TS_t.P, TS_s.P) \wedge I_{licm}(\iota, \varphi', \mathbb{S}') \tag{3}$$

the following holds.

$$I_{licm}, \iota \models (TS_t, \mathcal{S}'_t, M'_t) \preccurlyeq^{\circ,\emptyset}_{\varphi'} (TS_s, \mathcal{S}'_s, M'_s) \tag{g4.2}$$

By applying Lemma. 11.23 on (1) and (3), we have the following.

$$\Phi_{licm}(\varphi', \iota, (TS_t, \mathcal{S}'_t, M'_t), (TS_s, \mathcal{S}'_s, M'_s), \circ) \tag{4}$$

We finish the proof of such case from (4) and co-inductive hypothesis.

– The done case is similar with the case that $te \in AT$ and we omit the proof details here.
– Finally, we consider the abort case. From the premises, we have the following.

$$\iota \vdash (TS_t, \mathcal{S}_t, M_t) \longrightarrow \textbf{abort} \tag{5}$$

And we need to prove that there exist TS'_s, $\mathcal{S}'_s$ and M'_s such that:

$$\iota \vdash (TS_s, \mathcal{S}_s, M) \xrightarrow{na}{}^{*} (TS'_s, \mathcal{S}'_s, M'_s) \wedge \iota \vdash (TS'_s, \mathcal{S}'_s, M'_s) \longrightarrow \textbf{abort} \tag{g6}$$

□

Lemma 11.18 (well-defined preheader - I).

$$\begin{aligned}
&\forall \text{pre-header}, \mathsf{f}_s, \mathsf{f}_t, \text{loops}, C_s, C_t, B_t, \iota.\\
&\quad \text{pre-header}(\mathsf{f}_s) = \mathsf{f}_t \wedge C_t(\mathsf{f}_t) = B_t \wedge\\
&\quad \mathsf{TransC}'(C_s, \emptyset, \text{loops}, \text{loops}) = (C_t, \text{pre-header}) \wedge\\
&\quad (\forall(\mathsf{f}_{enty}, _, \text{loop_inv}) \in \text{loops}.\ \mathsf{f}_{entry} \in \mathrm{dom}(C_s) \wedge\\
&\qquad (\forall(\mathsf{x}, _) \in \text{loop_inv}.\ \mathsf{x} \notin \iota))\\
&\Longrightarrow \exists B_s.\ C_s(\mathsf{f}_s) = B_s \wedge \mathsf{wdph}(B_t, \mathsf{f}_s, C_s)
\end{aligned}$$

Lemma 11.19 (well-defined preheader - II).

$$\begin{aligned}
&\forall \text{pre-header}, \mathsf{f}_s, \mathsf{f}_t, \text{loops}.\\
&\quad C_t(\mathsf{f}) = B_t \wedge\\
&\quad \mathsf{TransC}'(C_s, \emptyset, \text{loops}, \text{loops}) = (C_t, \text{pre-header})\\
&\Longrightarrow \exists B_s.\ C_s(\mathsf{f}) = B_s \wedge\\
&\qquad (B_s = B_t \vee \text{ptB_ph_rel}(B_t, B_s, \text{pre-header}))
\end{aligned}$$

Lemma 11.20 (match state implies invT - licm).

$$\begin{aligned}
&\forall \varphi, \iota, TS_t, \mathcal{S}_t, M_t, TS_s, \mathcal{S}_s, M_s, \beta.\\
&\quad \Phi_{licm}(\varphi, \iota, (TS_t, \mathcal{S}_t, M_t), (TS_s, \mathcal{S}_s, M_s), \beta)\\
&\Longrightarrow \mathsf{invT}(\varphi, \iota, (TS_t, \mathcal{S}_t, M_t), (TS_s, \mathcal{S}_s, M_s), \beta, \emptyset)
\end{aligned}$$

Lemma 11.21 (match state licm preserving - atomic&output).

$$\begin{aligned}
&\forall \varphi, \iota, TS_t, \mathcal{S}_t, M_t, TS'_t, \mathcal{S}'_t, M'_t, TS_s, \mathcal{S}_s, M_s, \beta, te \in (Atm \cup \{\mathsf{out}(v)\}).\\
&\quad \Phi_{licm}(\varphi, \iota, (TS_t, \mathcal{S}_t, M_t), (TS_s, \mathcal{S}_s, M_s), \beta) \wedge\\
&\quad \iota \vdash (TS_t, \mathcal{S}_t, M_t) \xrightarrow{te} (TS'_t, \mathcal{S}'_t, M'_t)\\
&\Longrightarrow \exists TS'_s, \mathcal{S}'_s, M'_s, \varphi'.\\
&\qquad \Phi_{licm}(\varphi', \iota, (TS'_t, \mathcal{S}'_t, M'_t), (TS'_s, \mathcal{S}'_s, M'_s), \circ) \wedge\\
&\qquad \varphi \subseteq \varphi' \wedge \iota \vdash (TS_s, \mathcal{S}_s, M_s) \xrightarrow{te} (TS'_s, \mathcal{S}'_s, M'_s)
\end{aligned}$$

Lemma 11.22 (match state licm preserving - non-atomic).

$$\begin{aligned}
&\forall \varphi, \iota, TS_t, \mathcal{S}_t, M_t, TS_t', \mathcal{S}_t', M_t', TS_s, \mathcal{S}_s, M_s, \beta, ws.\\
&\quad \Phi_{licm}(\varphi, \iota, (TS_t, \mathcal{S}_t, M_t), (TS_s, \mathcal{S}_s, M_s), \beta) \wedge\\
&\quad \iota \vdash (TS_t, \mathcal{S}_t, M_t) \xrightarrow[ws]{na} (TS_t', \mathcal{S}_t', M_t')\\
&\Longrightarrow \exists TS_s', \mathcal{S}_s', M_s', ws'.\\
&\quad \Phi_{licm}(\varphi, \iota, (TS_t', \mathcal{S}_t', M_t'), (TS_s', \mathcal{S}_s', M_s'), \bullet) \wedge\\
&\quad \iota \vdash (TS_s, \mathcal{S}_s, M_s) \xrightarrow[ws']{na}{}^{*} (TS_s', \mathcal{S}_s', M_s') \wedge ws \subseteq ws'
\end{aligned}$$

Lemma 11.23 (match state licm preserving - rely).

$$\begin{aligned}
&\forall \varphi, \iota, TS_t, \mathcal{S}_t, M_t, TS_s, \mathcal{S}_s, M_s, \varphi', \mathcal{S}_t', M_t', \mathcal{S}_s', M_s'.\\
&\quad \Phi_{licm}(\varphi, \iota, (TS_t, \mathcal{S}_t, M_t), (TS_s, \mathcal{S}_s, M_s), \circ) \wedge\\
&\quad \mathsf{R}(\iota, (\varphi, (\mathcal{S}_t, M_t, \mathcal{S}_s, M_s), (\varphi', (\mathcal{S}_t', M_t', \mathcal{S}_s', M_s'), TS_t.P, TS_s.P) \wedge\\
&\quad I_{licm}(\iota, \varphi', (\mathcal{S}_t', M_t', \mathcal{S}_s', M_s'))\\
&\Longrightarrow \Phi_{licm}(\varphi', \iota, (TS_t, \mathcal{S}_t', M_t'), (TS_s, \mathcal{S}_s', M_s'), \circ)
\end{aligned}$$

Lemma 11.24 (match state licm preserving abort step).

$$\begin{aligned}
&\forall \varphi, \iota, TS_t, \mathcal{S}_t, M_t, TS_s, \mathcal{S}_s, M_s, \beta.\\
&\quad \Phi_{licm}(\varphi, \iota, (TS_t, \mathcal{S}_t, M_t), (TS_s, \mathcal{S}_s, M_s), \beta) \wedge\\
&\quad \iota \vdash (TS_t, \mathcal{S}_t, M_t) \longrightarrow \textbf{abort}\\
&\Longrightarrow \iota \vdash (TS_s, \mathcal{S}_s, M_s) \longrightarrow \textbf{abort}
\end{aligned}$$

$$\frac{M_t = M_s \qquad \mathcal{S}_t = \mathcal{S}_s \qquad \lVert M_t \rVert = \mathrm{dom}(\varphi) \qquad (\forall (\mathsf{x}, t) \in \mathrm{dom}(\varphi).\ \varphi(\mathsf{x}, t) = t)}{I_{cse}(\iota, \varphi, (\mathcal{S}_t, M_t), (\mathcal{S}_s, M_s))}$$

Fig. 47. Invariant in common subexpression elimination proof

$$\begin{array}{lll} (R, \mathcal{V}, M) \models (r, e) & ::= & R(r) = [\![e]\!]_R \\ (R, \mathcal{V}, M) \models (r, \mathsf{x}, t) & ::= & \langle \mathsf{x} : R(r)@(\_, t], \_\rangle \in M \wedge (\mathcal{V}.\mathrm{cur}.T_{\mathsf{na}}(\mathsf{x}) \le t \le \mathcal{V}.\mathrm{cur}.T_{\mathsf{rlx}}(\mathsf{x})) \\ (R, \mathcal{V}, M) \models_\iota L_a & ::= & (\forall (r, e) \in L_a.\ (R, \mathcal{V}, M) \models_\iota (r, e)) \wedge \\ & & (\forall (r, \mathsf{x}) \in L_a.\ (R, \mathcal{V}, M) \models (r, \mathsf{x}) \wedge \mathsf{x} \notin \iota) \end{array}$$

$$\frac{\begin{array}{c}\mathtt{Ave\_Analyzer}(C_s, l_0) = \mathbb{L}_a \qquad \mathsf{TransC}_{cse}(C_s, \mathbb{L}_a) = C_t \\ B_s = C_s(l)[i \ldots] \qquad \mathsf{TransB}_{cse}(B_s, LB_a) = B_t \\ R_t = R_s \qquad (R_s, \mathcal{V}_s, M_s) \models_\iota \mathsf{IN}[LB_a] \\ \forall l_p \in \mathrm{succ}(B_s).\ \mathsf{OUT}[LB_a] \ge \mathsf{IN}[\mathbb{L}_a(l_p)]\end{array}}{\mathcal{V}_s, M_s, \iota \vdash (R_t, B_t, C_t) \sim_{cse} (R_s, B_s, C_s)}$$

$$\frac{K_t = K_s = \epsilon}{\iota \vdash K_t \sim_{cse} K_s} \qquad \frac{\forall \mathcal{V}_s, M_s.\ \mathcal{V}_s, M_s, \iota \vdash (R_t, B_t, C_t) \sim_{cse} (R_s, B_s, C_s) \qquad \iota \vdash K'_t \sim_{cse} K'_s}{\iota \vdash ((R_t, B_t, C_t) :: K'_t) \sim_{cse} ((R_s, B_s, C_s) :: K'_s)}$$

$$\frac{\begin{array}{c}\mathtt{Ave\_Analyzer}(\pi_s) = A_a \qquad \mathsf{Translater}_{cse}(\pi_s, A_a) = \pi_t \\ \mathcal{V}_s, M_s, \iota \vdash (R_t, B_t, C_t) \sim_{cse} (R_s, B_s, C_s) \qquad \iota \vdash K_t \sim_{cse} K_s\end{array}}{\mathcal{V}_s, M_s, \iota \vdash (R_t, B_t, C_t, K_t, \pi_t) \sim_{cse} (R_s, B_s, C_s, K_s, \pi_s)}$$

$$\frac{\mathcal{V}_s, M_s, \iota \vdash \sigma_t \sim_{cse} \sigma_s \qquad \mathcal{V}_t = \mathcal{V}_s \qquad P_t = P_s}{\iota \vdash (\sigma_t, \mathcal{V}_t, P_t) \sim_{cse} ((\sigma_s, \mathcal{V}_s, P_s), M_s)}$$

$$\frac{\begin{array}{c}I_{cse}(\iota, \varphi', (\mathcal{S}_t, M_t, \mathcal{S}_s, M_s)) \qquad \iota \vdash TS_t \sim_{cse} (TS_s, M_s) \\ (\beta = \circ \wedge \varphi = \varphi') \vee (\beta = \bullet \wedge \varphi \subseteq \varphi') \qquad \lVert TS_t.P \rVert \subseteq \mathrm{dom}(\varphi)\end{array}}{\Phi_{cse}(\varphi, \iota, (TS_t, \mathcal{S}_t, M_t), (TS_s, \mathcal{S}_s, M_s), \beta)}$$

Fig. 48. Match state in common subexpression elimination proof

11.4 Correctness proof of Common Subexpression Elimination

Invariant in common subexpression elimination proof. We show the invariant I_{cse} for shared resource in Fig. 47.

Match state in common subexpression elimination proof. We define the match state in common subexpression elimination proof in Fig. 48.

Correctness proof of common subexpression elimination optimizer. The correctness proof of common subexpression elimination optimizer is similar with the correctness proof of constant propagation.

A CAPPED MEMORY

We give the formal definition of contructing capped memory below.

- The last message of a memory M to a location x.

$$\overline{m}(M, \mathsf{x}) \ ::= \ \underset{m \in M(\mathsf{x})}{\arg\max} \ m.\mathsf{to}$$

- The cap timemap of a memory M.

$$\widehat{T}(M) \ ::= \ \lambda \mathsf{x}.\, \overline{m}(\widetilde{M}, \mathsf{x}).\mathsf{to}$$

- Cap message of a memory M to a location x.

$$\widehat{m}(M, \mathsf{x}) \ ::= \ \langle \mathsf{x} : (\overline{m}(M, \mathsf{x}).\mathsf{to}, \overline{m}(M, \mathsf{x}).\mathsf{to}{+}1] \rangle$$

- Capped memory.

Definition A.1 (Capped Message). $M_c \in \widehat{M}$ holds iff $M \subseteq M_c$ and the following hold:

(1) for any $m_1, m_2 \in M$, if $m_1.\mathsf{var} = m_2.\mathsf{var}$, $m_1.\mathsf{to} < m_2.\mathsf{to}$, $\neg(\exists m \in M.\ m.\mathsf{var} = m_1.\mathsf{var} \wedge m_1.\mathsf{to} < m.\mathsf{to} < m_2.\mathsf{to})$ and $m_1.\mathsf{to} < m_2.\mathsf{from}$, then $\langle m_1.\mathsf{var} : (m_1.\mathsf{to}, m.\mathsf{from}] \rangle \in M_c$;
(2) $\forall \mathsf{x} \in \mathit{Var}.\ \widehat{m}(M, \mathsf{x}) \in M_c$;
(3) for any $m \in M_c$, if $m \notin M$, then there exists $m' \in M$, such that $m'.\mathsf{var} = m.\mathsf{var}$ and $m'.\mathsf{to} < m.\mathsf{from}$.

$$\frac{\begin{array}{c}\text{for any } i \in \{1,\ldots,n\}.\ \mathsf{Init}(\pi, \mathtt{f}_i) = \sigma_i \qquad TS_i = (\sigma_i, \mathcal{V}_i, \emptyset) \\ \mathcal{TP} = \{1 \rightsquigarrow TS_1, \ldots, n \rightsquigarrow TS_n\} \qquad \mathtt{t} \in \{1,\ldots,n\} \qquad M = \{\langle \mathtt{x} : 0@(0,0], V_\perp\rangle \mid x \in Var\}\end{array}}{\textbf{let } (\pi, \iota) \textbf{ in } \mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n \xrightarrow{load} \triangleright (\mathcal{TP}, \mathtt{t}, \lambda x.0, M)^\iota} \text{(Load)}$$

$$\frac{\iota \vdash (\mathcal{TP}(\mathtt{t}), \mathcal{S}, M) \xrightarrow{na}{}^{+} (TS', \mathcal{S}', M') \quad \mathsf{consistent}(TS', M', \iota)}{(\mathcal{TP}, \mathtt{t}, \mathcal{S}, M)^\iota \xrightarrow{na} \triangleright (\mathcal{TP}\{\mathtt{t} \rightsquigarrow TS'\}, \mathtt{t}, \mathcal{S}', M')^\iota} \text{(NA-step)}$$

$$\frac{\iota \vdash (\mathcal{TP}(\mathtt{t}), \mathcal{S}, M) \xrightarrow{prc}{}^{+} (TS', \mathcal{S}', M') \quad \mathsf{consistent}(TS', M', \iota)}{(\mathcal{TP}, \mathtt{t}, \mathcal{S}, M)^\iota \xrightarrow{prc} \triangleright (\mathcal{TP}\{\mathtt{t} \rightsquigarrow TS'\}, \mathtt{t}, \mathcal{S}', M')^\iota} \text{(PRC-step)}$$

$$\frac{\begin{array}{c}\iota \vdash (\mathcal{TP}(\mathtt{t}), \mathcal{S}, M) \xrightarrow{prc}{}^{*} (TS_0, \mathcal{S}_0, M_0) \\ \iota \vdash (TS_0, \mathcal{S}_0, M_0) \xrightarrow{\mathsf{atmBlk}}{}^{+} (TS', \mathcal{S}', M') \\ \mathsf{consistent}(TS', M', \iota)\end{array}}{(\mathcal{TP}, \mathtt{t}, \mathcal{S}, M)^\iota \xrightarrow{at} \triangleright (\mathcal{TP}\{\mathtt{t} \rightsquigarrow TS'\}, \mathtt{t}, \mathcal{S}', M')^\iota} \text{(ATM-step)}$$

$$\frac{\begin{array}{c}\iota \vdash (\mathcal{TP}(\mathtt{t}), \mathcal{S}, M) \xrightarrow{na}{}^{*} (TS_0, \mathcal{S}_0, M_0) \\ \iota \vdash (TS_0, \mathcal{S}_0, M_0) \xrightarrow{\mathsf{out}(v)} (TS', \mathcal{S}', M') \\ \mathsf{consistent}(TS', M', \iota)\end{array}}{(\mathcal{TP}, \mathtt{t}, \mathcal{S}, M)^\iota \xrightarrow{\mathsf{out}(v)} \triangleright (\mathcal{TP}\{\mathtt{t} \rightsquigarrow TS'\}, \mathtt{t}, \mathcal{S}', M')^\iota} \text{(Out-step)}$$

$$\frac{\mathtt{t}' \in \mathrm{dom}(\mathcal{TP})}{(\mathcal{TP}, \mathtt{t}, \mathcal{S}, M)^\iota \xrightarrow{\mathsf{sw}} \triangleright (\mathcal{TP}, \mathtt{t}', \mathcal{S}, M)^\iota} \text{(sw-step)}$$

$$\frac{\iota \vdash (\mathcal{TP}(\mathtt{t}), \mathcal{S}, M) \longrightarrow \textbf{done} \quad \mathtt{t}' \in \mathrm{dom}(\mathcal{TP}\backslash\{\mathtt{t}\})}{(\mathcal{TP}, \mathtt{t}, \mathcal{S}, M)^\iota \xrightarrow{\mathsf{tterm}} \triangleright (\mathcal{TP}\backslash\{\mathtt{t}\}, \mathtt{t}', \mathcal{S}, M)^\iota} \text{(thrd-term)}$$

$$\frac{\iota \vdash (\mathcal{TP}(\mathtt{t}), \mathcal{S}, M) \longrightarrow \textbf{done} \quad \mathrm{dom}(\mathcal{TP}) = \{\mathtt{t}\}}{(\mathcal{TP}, \mathtt{t}, \mathcal{S}, M)^\iota \Longrightarrow \triangleright \textbf{done}} \text{(prog-done)}$$

$$\frac{\iota \vdash (\mathcal{TP}(\mathtt{t}), \mathcal{S}, M) \longrightarrow^{*} (TS', \mathcal{S}', M') \qquad \iota \vdash (TS', \mathcal{S}', M') \longrightarrow \textbf{abort}}{(\mathcal{TP}, \mathtt{t}, \mathcal{S}, M)^\iota \Longrightarrow \triangleright \textbf{abort}} \text{(abort)}$$

Fig. 49. Machine step in auxiliary promising semantics

B PROOF OF SEMANTICS EQUIVALENCE

We show the correctness proof of Lemma. 6.1 in this section. In this proof, we first define an auxiliary promising semantics.

Auxiliary promising semantics. In order to facilitate the proof of the semanitcs equivalence between promising semnatics and the non-preemptive semantics, which will be introduced in Sec. 6, we provide the auxiliary promising semantics defined in Fig. 49 (an auxiliary definition is defined below).

$$\begin{array}{l}\iota \vdash (TS, \mathcal{S}, M) \xrightarrow{\mathsf{atmBlk}} (TS', \mathcal{S}', M') \ ::= \\ \quad \exists TS_0, \mathcal{S}_0, M_0, TS_1, \mathcal{S}_1, M_1. \\ \qquad \iota \vdash (TS, \mathcal{S}, M) \xrightarrow{na}{}^{*} (TS_0, \mathcal{S}_0, M_0) \wedge \iota \vdash (TS_0, \mathcal{S}_0, M_0) \xrightarrow{at} (TS_1, \mathcal{S}_1, M_1) \wedge \\ \qquad\quad \iota \vdash (TS_1, \mathcal{S}_1, M_1) \xrightarrow{prc}{}^{*} (TS_2, \mathcal{S}_2, M_2)\end{array}$$

We can prove that the following conclusion holds.

Lemma B.1 (PS to aux-PS). For any W and W', if $W \Longrightarrow W'$, then $W \Longrightarrow\!\triangleright^{*} W'$.

Proof. Prove by applying Lemma. B.2. □

$$\mathit{AProgEtr}(\mathbb{P}, \mathcal{B}) \quad \text{iff} \ \exists W, n.\ (\mathbb{P} \overset{\mathit{load}}{\Longrightarrow\!\!\triangleright}\ W) \wedge \mathit{AEtr}^n(W, \mathcal{B})$$

$$\frac{}{\mathit{AEtr}^0(W, \epsilon)} \qquad \frac{W \Longrightarrow\!\!\triangleright \mathbf{abort}}{\mathit{AEtr}^{n+1}(W, \mathbf{abort})} \qquad \frac{W \Longrightarrow\!\!\triangleright \mathbf{done}}{\mathit{AEtr}^{n+1}(W, \mathbf{done})}$$

$$\frac{W \overset{\mathrm{out}(v)}{\Longrightarrow\!\!\triangleright}\ W' \qquad \mathit{AEtr}^n(W', \mathcal{B})}{\mathit{AEtr}^{n+1}(W, \mathrm{out}(v) :: \mathcal{B})} \qquad \frac{W \Longrightarrow\!\!\triangleright\ W' \qquad \mathit{AEtr}^n(W', \mathcal{B})}{\mathit{AEtr}^{n+1}(W, \mathcal{B})}$$

Fig. 50. Event trace under the auxiliary promisng semantics

Lemma B.2 (PS steps split).

$$\begin{aligned}
&\forall TS, \mathcal{S}, M, TS', \mathcal{S}', M', \iota, n.\\
&\quad \iota \vdash (TS, \mathcal{S}, M) \longrightarrow^n (TS', \mathcal{S}', M')\\
&\Longrightarrow\ \exists TS_0, \mathcal{S}_0, M_0, TS_1, \mathcal{S}_1, M_1.\\
&\qquad \iota \vdash (TS, \mathcal{S}, M) \xrightarrow{prc}{}^{*} (TS_0, \mathcal{S}_0, M_0) \wedge\\
&\qquad\quad \iota \vdash (TS_0, \mathcal{S}_0, M_0) \xrightarrow{\mathsf{atmBlk}}{}^{*} (TS_1, \mathcal{S}_1, M_1) \wedge\\
&\qquad\qquad \iota \vdash (TS_1, \mathcal{S}_1, M_1) \xrightarrow{na}{}^{*} (TS', \mathcal{S}', M')
\end{aligned}$$

From Lemma. B.1, we can prove the equivalence between the promising semantics and the auxiliary promising semantics as the following shown.

Lemma B.3 (Semantics equivalence - PS2APS).

$$\begin{aligned}
&\forall \pi, \iota, \mathtt{f}_1, \ldots, \mathtt{f}_n, \mathcal{B}.\\
&\quad \mathit{ProgEtr}(\mathbf{let}\ \pi\ \mathbf{in}\ \mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n, \mathcal{B}) \Longleftrightarrow \mathit{AProgEtr}(\mathbf{let}\ \pi\ \mathbf{in}\ \mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n, \mathcal{B})
\end{aligned}$$

Proof. Prove by applying Lemma. B.1. □

Equivalence between the auxiliary promising semantics and the non-preemptive semantics. Then, we prove that the auxiliary promising semantics and the non-preemptive semantics are equivalent.

Lemma B.4 (Semantics equvialence - APS2NP).

$$\begin{aligned}
&\forall \pi, \mathtt{f}_1, \ldots, \mathtt{f}_n, \iota, \mathcal{B}.\\
&\quad \mathit{AProgEtr}(\mathbf{let}\ (\pi, \iota)\ \mathbf{in}\ \mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n, \mathcal{B}) \Longleftrightarrow \mathit{NPProgEtr}(\mathbf{let}\ (\pi, \iota)\ \mathbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n, \mathcal{B})
\end{aligned}$$

Proof. For "$\mathit{NPProgEtr}(\mathbf{let}\ (\pi, \iota)\ \mathbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n, \mathcal{B}) \implies \mathit{AProgEtr}(\mathbf{let}\ (\pi, \iota)\ \mathbf{in}\ \mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n, \mathcal{B})$", since every step in the non-preemptive semantics can be easily converted to a step of promising semantics, it is obviously that every event trace in the non-preemptive semantics can be produced in promising semantics.

We show the proof of "$\mathit{AProgEtr}(\mathbf{let}\ (\pi, \iota)\ \mathbf{in}\ \mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n, \mathcal{B}) \implies \mathit{NPProgEtr}(\mathbf{let}\ (\pi, \iota)\ \mathbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n, \mathcal{B})$". We do intros and have the following.

$$\mathit{AProgEtr}(\mathbf{let}\ (\pi, \iota)\ \mathbf{in}\ \mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n, \mathcal{B}) \tag{1}$$

We unfold (1) and get that there exists $\mathcal{TP}$, t, $\mathcal{S}$, M and n such that the followings hold.

$$\mathbf{let}\ (\pi, \iota)\ \mathbf{in}\ \mathtt{f}_1 \parallel \cdots \parallel \mathtt{f}_n \overset{\mathit{load}}{\Longrightarrow\!\!\triangleright} (\mathcal{TP}, \mathtt{t}, \mathcal{S}, M)^{\iota} \tag{2}$$

$$\mathit{AEtr}^n((\mathcal{TP}, \mathtt{t}, \mathcal{S}, M)^{\iota}, \mathcal{B}) \tag{3}$$

$$\frac{W \overset{na/\mathrm{sw}}{\Longrightarrow\!\!\!\!\!\!\triangleright}{}^{*}\ W'}{\text{sw-procs}^{0}(W, W', \epsilon)} \qquad \frac{W \overset{na/\mathrm{sw}}{\Longrightarrow\!\!\!\!\!\!\triangleright}{}^{*}\ W' \qquad W' \Longrightarrow\!\triangleright\ \mathbf{done}}{\text{sw-procs}^{n+1}(W, W', \mathbf{done})}$$

$$\frac{W \overset{na/\mathrm{sw}}{\Longrightarrow\!\!\!\!\!\!\triangleright}{}^{*}\ W' \qquad W' \Longrightarrow\!\triangleright\ \mathbf{abort}}{\text{sw-procs}^{n+1}(W, W', \mathbf{abort})}$$

$$\frac{W \overset{na/\mathrm{sw}}{\Longrightarrow\!\!\!\!\!\!\triangleright}{}^{*}\ W_1 \qquad W_1 \overset{\mathrm{out}(v)}{\Longrightarrow\!\!\!\!\!\!\triangleright}\ W_2 \qquad \text{sw-procs}^{n}(W_2, W', \mathcal{B})}{\text{sw-procs}^{n+1}(W, W', \mathrm{out}(v) :: \mathcal{B})}$$

$$\frac{\begin{array}{c} W \overset{na/\mathrm{sw}}{\Longrightarrow\!\!\!\!\!\!\triangleright}{}^{*}\ W_1 \\ (W_1 \overset{at}{\Longrightarrow\!\!\triangleright}\ W_2) \vee (W_1 \overset{prc}{\Longrightarrow\!\!\triangleright}\ W_2) \vee (W_1 \overset{\mathrm{tterm}}{\Longrightarrow\!\!\!\!\!\triangleright}\ W_2) \\ \text{sw-procs}^{n}(W_2, W', \mathcal{B}) \end{array}}{\text{sw-procs}^{n+1}(W, W', \mathcal{B})}$$

$$\frac{W^{\mathrm{t}} = W'}{\text{NAStep}^{0}(W, W')} \qquad \frac{W \overset{na}{\Longrightarrow\!\!\triangleright}\ W_0 \qquad \text{NAStep}^{n}(W_0^{\mathrm{t}_0}, W')}{\text{NAStep}^{n+1}(W, W')}$$

$$\frac{W^{\mathrm{t}} = W'}{\text{PRCStep}^{0}(W, W')} \qquad \frac{W \overset{prc}{\Longrightarrow\!\!\triangleright}\ W_0 \qquad \text{PRCStep}^{n}(W_0, W')}{\text{PRCStep}^{n+1}(W, W')}$$

$$((\mathcal{TP}, \mathrm{t}, \mathcal{S}, M)^{\iota})^{\mathrm{t}'} \quad ::= \quad (\mathcal{TP}, \mathrm{t}', \mathcal{S}, M)^{\iota}$$

Fig. 51. Auxiliary definitions in semantics equivalent proof

We apply Lemma. B.5 on (3) and get that there exists n' and W' such that the followings hold:

$$\text{sw-procs}^{n'}((\mathcal{TP}, \mathrm{t}, \mathcal{S}, M)^{\iota}, W', \mathcal{B}) \tag{4}$$

We can construct a non-preemptive program state such that:

$$\mathbf{let}\ (\pi, \iota)\ \mathbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n :\overset{load}{\Longrightarrow} (\mathcal{TP}, \mathrm{t}, \mathcal{S}, M, \circ)^{\iota} \tag{5}$$

We apply Lemma. B.6 on (4) and get that the following holds.

$$NPEtr^{*}((\mathcal{TP}, \mathrm{t}, \mathcal{S}, M, \circ)^{\iota}, \mathcal{B})$$

□

Lemma B.5 (AEtr to sw-procs).

$$\forall W, n, \mathcal{B}.\ AEtr^{n}(W, \mathcal{B}) \implies \exists n', W'.\ \text{sw-procs}^{n'}(W, W', \mathcal{B})$$

Proof. Prove by induction on n. □

Lemma B.6 (sw-procs to NPEtr).

$$\begin{array}{l}\forall \mathcal{TP}, \mathcal{S}, M, \mathrm{t}, \iota, W', \hat{W}, n, \mathcal{B}.\\ \quad \text{sw-procs}^n((\mathcal{TP}, \mathrm{t}, \mathcal{S}, M)^\iota, \mathrm{t}', W', \mathcal{B}) \wedge \mathsf{wdSt}(\mathcal{TP}, \mathcal{S}, M)\\ \Longrightarrow \ \exists \mathrm{t}'.\ \mathit{NPEtr}^*((\mathcal{TP}, \mathrm{t}', \mathcal{S}, M, \circ)^\iota, \mathcal{B})\end{array}$$

Proof. Prove by induction on n.

0: We get that $\mathcal{B} = \epsilon$. We can prove that $\mathit{NPEtr}^0((\mathcal{TP}, \mathrm{t}, \mathcal{S}, M)^\iota, \epsilon)$.

$n+1$: We do intro and have the following.

$$\text{sw-procs}^{n+1}((\mathcal{TP}, \mathrm{t}, \mathcal{S}, M)^\iota, W', \mathcal{B}) \tag{1}$$
$$\mathsf{wdSt}(\mathcal{TP}, \mathcal{S}, M) \tag{2}$$

We unfold (1) and discuss each case respectively.

- $\mathcal{B} \in \{\textbf{done}, \textbf{abort}\}$. We finish the proof directly.
- $\mathcal{B} = \mathrm{out}(v) :: \mathcal{B}'$. We have that there exist W_1 and W_2 such that:

$$(\mathcal{TP}, \mathrm{t}, \mathcal{S}, M)^\iota \xrightarrow{na/\text{sw}}{}^* W_1 \tag{3}$$
$$W_1 \xrightarrow{\mathrm{out}(v)} W_2 \tag{4}$$
$$\text{sw-procs}^n(W_2, W', \mathcal{B}) \tag{5}$$

  We apply Lemma. B.7 on (3) and (4) and have that there exist W_{01} and W_{02} such that:

$$(\mathcal{TP}, \mathrm{t}, \mathcal{S}, M)^\iota \xrightarrow{prc/\text{sw}}{}^* W_{01} \tag{6}$$
$$W_{01} \xrightarrow{\mathrm{out}(v)} W_{02} \tag{7}$$
$$W_{02} \xrightarrow{na/\text{sw}}{}^* W_2 \tag{8}$$

  Let $W_{02} = (\mathcal{TP}_{02}, \mathrm{t}_{02}, \mathcal{S}_{02}, M_{02})^\iota$. From (6) and (7), we have the following that there exists $\hat{W}_{01}$ such that:

$$(\mathcal{TP}, \mathrm{t}, \mathcal{S}, M, \circ)^\iota :\Longrightarrow^* \hat{W}_{01} \tag{9}$$
$$\hat{W}_{01} :\xrightarrow{\mathrm{out}(v)} (\mathcal{TP}_{02}, \mathrm{t}_{02}, \mathcal{S}_{02}, M_{02}, \circ)^\iota \tag{10}$$

  From (8) and (5), we have the following.

$$\text{sw-procs}^n((\mathcal{TP}_{02}, \mathrm{t}_{02}, \mathcal{S}_{02}, M_{02})^\iota, W', \mathcal{B}) \tag{11}$$

  We finish the proof by applying the inductive hypothesis on (11).
- We consider the case that the program takes an atomic step. We have that there exist W_1 and W_2 such that:

$$(\mathcal{TP}, \mathrm{t}, \mathcal{S}, M)^\iota \xrightarrow{na/\text{sw}}{}^* W_1 \tag{12}$$
$$W_1 \xrightarrow{at} W_2 \tag{13}$$
$$\text{sw-procs}^n(W_2, W', \mathcal{B}) \tag{14}$$

We apply Lemma. B.8 on (12) and (13) and have that there exist W_{01} and W_{02} such that:

$$(\mathcal{TP}, \mathrm{t}, \mathcal{S}, M)^{\iota} \xrightarrow{prc/\mathrm{sw}}{}^{*} W_{01} \tag{15}$$

$$W_{01} \xrightarrow{at} W_{02} \tag{16}$$

$$W_{02} \xrightarrow{na/\mathrm{sw}}{}^{*} W_2 \tag{17}$$

Let $W_{02} = (\mathcal{TP}_{02}, \mathrm{t}_{02}, \mathcal{S}_{02}, M_{02})^{\iota}$. From (15) and (16), we have the following.

$$(\mathcal{TP}, \mathrm{t}, \mathcal{S}, M, \circ)^{\iota} :\Longrightarrow^{+} (\mathcal{TP}_{02}, \mathrm{t}_{02}, \mathcal{S}_{02}, M_{02}, \circ)^{\iota} \tag{18}$$

From (17) and (14), we have the following.

$$\text{sw-procs}^{n}((\mathcal{TP}_{02}, \mathrm{t}_{02}, \mathcal{S}_{02}, M_{02})^{\iota}, W', \mathcal{B}) \tag{19}$$

We finish the proof by applying the inductive hypothesis on (19).

- Similarily, if the program takes a PRC-step, we finish the proof by applying Lemma. B.9 and the inductive hypothesis. And if the program takes a thread termination step, we finish the proof by applying Lemma. B.10 and the inductive hypothesis.

□

Lemma B.7 (Switch point fowarding - output).

$$\begin{aligned}&\forall W, W', W'', n.\\ &\quad W \xrightarrow{na/\mathrm{sw}}{}^{n} W' \wedge W' \xrightarrow{\mathrm{out}(v)} W''\\ &\Longrightarrow \exists W_0, W_1.\ W \xrightarrow{prc/\mathrm{sw}}{}^{*} W_0 \wedge W_0 \xrightarrow{\mathrm{out}(v)} W_1 \wedge W_1 \xrightarrow{na/\mathrm{sw}}{}^{*} W''\end{aligned}$$

Proof. From the premises, we have the following.

$$W \xrightarrow{na/\mathrm{sw}}{}^{n} W' \tag{1}$$

$$W' \xrightarrow{\mathrm{out}(v)} W'' \tag{2}$$

By applying Lemma. B.11 on (1), we have that there exist t such that.

$$\mathsf{NAStep}^{*}(W^{\mathrm{t}}, W') \tag{3}$$

By applying Lemma. B.12 on (3) and (2), we have that there exist W_0, W_1, t′ and t_0 such that:

$$\mathsf{PRCStep}^{*}(W^{\mathrm{t}'}, W_0) \tag{4}$$

$$W_0 \xrightarrow{\mathrm{out}(v)} W_1 \tag{5}$$

$$W_1 \xrightarrow{na/\mathrm{sw}}{}^{*} W'' \tag{6}$$

From (4), we have the following.

$$W^{\mathrm{t}'} \xrightarrow{prc/\mathrm{sw}}{}^{*} W_0 \tag{7}$$

We finish the proof. □

Lemma B.8 (Switch point fowarding - atomic).

$$\begin{aligned}&\forall W, W', W'', n.\\ &\quad W \xrightarrow{na/\mathrm{sw}}{}^{n} W' \wedge W' \xrightarrow{at} W''\\ &\Longrightarrow \exists W_0, W_1.\ W \xrightarrow{prc/\mathrm{sw}}{}^{*} W_0 \wedge W_0 \xrightarrow{at} W_1 \wedge W_1 \xrightarrow{na/\mathrm{sw}}{}^{*} W''\end{aligned}$$

Proof. From the premise, we have the following.

$$W \xrightarrow{na/sw} \triangleright^{n} W' \tag{8}$$

$$W' \xrightarrow{at} \triangleright\, W'' \tag{9}$$

By applying Lemma. B.11 on (8), we have that there exists t such that:

$$\mathsf{NAStep}^{*}(W^{\mathsf{t}}, W') \tag{10}$$

By applying Lemma. B.13 on (10), (9), we have that there exist W_0, W_1, t' and t_0 such that:

$$\mathsf{PRCStep}^{*}(W^{\mathsf{t}'}, W_0) \tag{11}$$

$$W_0 \xrightarrow{at} \triangleright\, W_1 \tag{12}$$

$$W_1 \xrightarrow{na/sw} \triangleright^{*} W'' \tag{13}$$

From (11), we have the following.

$$W \xrightarrow{prc/sw} \triangleright^{*} W_0 \tag{14}$$

We finish the proof. □

Lemma B.9 (Switch point fowarding - prc).

$$\begin{aligned} &\forall W, W', W'', n. \\ &\quad W \xrightarrow{na/sw} \triangleright^{n} W' \wedge W' \xrightarrow{prc} \triangleright\, W'' \\ &\implies \exists W_0.\ W \xrightarrow{prc/sw} \triangleright^{*} W_0 \wedge W_0 \xrightarrow{na/sw} \triangleright^{*} W'' \end{aligned}$$

Proof. From the premises, we have the following.

$$W \xrightarrow{na/sw} \triangleright^{n} W' \tag{15}$$

$$W' \xrightarrow{prc} \triangleright\, W'' \tag{16}$$

We apply Lemma. B.11 on (15) and have that there exists t such that:

$$\mathsf{NAStep}^{*}(W^{\mathsf{t}}, W') \tag{17}$$

By applying Lemma. B.14 on (17) and (16), we have that there exist W_0 and t' such that:

$$\mathsf{PRCStep}^{*}(W^{\mathsf{t}'}, W_0) \tag{18}$$

$$W_0 \xrightarrow{na/sw} \triangleright^{*} W'' \tag{19}$$

From (18), we have the following.

$$W \xrightarrow{na/sw} \triangleright^{*} W_0 \tag{20}$$

□

Lemma B.10 (Switch point fowarding - thread termination).

$$\begin{aligned} &\forall W, W', W'', n. \\ &\quad W \xrightarrow{na/sw} \triangleright^{n} W' \wedge W' \xrightarrow{\mathsf{tterm}} \triangleright\, W'' \\ &\implies \exists W_0, W_1.\ W \xrightarrow{prc/sw} \triangleright^{*} W_0 \wedge W_0 \xrightarrow{\mathsf{tterm}} \triangleright\, W_1 \wedge W_1 \xrightarrow{na/sw} \triangleright^{*} W'' \end{aligned}$$

We show some auxiliary lemmas in the following that are used in the proof of the above lemmas.

Lemma B.11 (NA/sw to NA step).

$$\forall W, W', n.\ (W \overset{na/\mathrm{sw}}{\Longrightarrow\!\!\!\!\!\!\triangleright}{}^{n}\ W') \implies \exists t.\ \mathsf{NAStep}^{*}(W^{\mathrm{t}}, W')$$

Lemma B.12 (Switch point forwarding - output aux).

$$\begin{aligned}&\forall W, W', W'', n.\\ &\quad \mathsf{NAStep}^{n}(W, W') \wedge W' \overset{\mathrm{out}(v)}{\Longrightarrow\!\!\!\!\!\!\triangleright}\ W''\\ &\implies \exists W_0, W_1, \mathrm{t}, \mathrm{t}_0.\\ &\qquad \mathsf{PRCStep}^{*}(W^{\mathrm{t}}, W_0) \wedge W_0^{\mathrm{t}_0} \overset{\mathrm{out}(v)}{\Longrightarrow\!\!\!\!\!\!\triangleright}\ W_1 \wedge W_1 \overset{na/\mathrm{sw}}{\Longrightarrow\!\!\!\!\!\!\triangleright}{}^{*}\ W''\end{aligned}$$

Lemma B.13 (Switch point forwarding - atomic aux).

$$\begin{aligned}&\forall W, W', W'', n.\\ &\quad \mathsf{NAStep}^{n}(W, W') \wedge W' \overset{at}{\Longrightarrow\!\!\triangleright}\ W''\\ &\implies \exists W_0, W_1, \mathrm{t}, \mathrm{t}_0.\\ &\qquad \mathsf{PRCStep}^{*}(W^{\mathrm{t}}, W_0) \wedge W_0^{\mathrm{t}_0} \overset{at}{\Longrightarrow\!\!\triangleright}\ W_1 \wedge W_1 \overset{na/\mathrm{sw}}{\Longrightarrow\!\!\!\!\!\!\triangleright}{}^{*}\ W''\end{aligned}$$

Proof. Prove by induction on n.

0: We finish the proof directly.

$n+1$: From the premises, we have the following.

$$\mathsf{NAStep}^{n+1}(W, W') \tag{21}$$

$$W' \overset{at}{\Longrightarrow\!\!\triangleright}\ W'' \tag{22}$$

We unfold (21) and have that there exist W_0 and t_0 such that:

$$W \overset{na}{\Longrightarrow\!\!\triangleright}\ W_0 \tag{23}$$

$$\mathsf{NAStep}^{n}(W_0^{\mathrm{t}_0}, W') \tag{24}$$

We apply the inductive hypothesis on (24) and (22), and have that there exist W_1, W_2, t_0', t_1 such that:

$$\mathsf{PRCStep}^{*}(W_0^{\mathrm{t}_0'}, W_1) \tag{25}$$

$$W_1^{\mathrm{t}_1} \overset{at}{\Longrightarrow\!\!\triangleright}\ W_2 \tag{26}$$

$$W_2 \overset{na/\mathrm{sw}}{\Longrightarrow\!\!\!\!\!\!\triangleright}{}^{*}\ W'' \tag{27}$$

We apply Lemma. B.15 on (24) and (25), and have that there exist W_3, t and t_3 such that:

$$\mathsf{PRCStep}^{*}(W^{\mathrm{t}}, W_3) \tag{28}$$

$$W_3^{\mathrm{t}_3} \overset{na}{\Longrightarrow\!\!\triangleright}\ W_1^{\mathrm{t}_3} \tag{29}$$

By applying Lemma. B.16 on (29) and (26), we have that there exist W_4 and W_5 such that:

$$\mathsf{PRCStep}^{*}(W_3^{\mathrm{t}_3}, W_4) \tag{30}$$

$$W_4^{\mathrm{t}_1} \overset{at}{\Longrightarrow\!\!\triangleright}\ W_5 \tag{31}$$

$$W_5 \overset{na/\mathrm{sw}}{\Longrightarrow\!\!\!\!\!\!\triangleright}{}^{*}\ W_2 \tag{32}$$

Thus, we are done. □

$$
\begin{aligned}
M \approx^{\iota}_{\mathsf{at}} M' \quad ::= \quad & (\forall \langle \mathsf{x} : v@(f,t], V\rangle \in M. \\
& \qquad \exists V'.\ (\langle \mathsf{x} : v@(f,t], V'\rangle \in M' \wedge (\iota(x) = \mathsf{at} \implies V = V'))) \wedge \\
& (\forall \langle \mathsf{x} : v@(f,t], V'\rangle \in M'.\ \langle \mathsf{x} : v@(f,t], \_\rangle \in M) \wedge \\
& (\forall x, f, t.\ \langle \mathsf{x} : (f,t]\rangle \in M \iff \langle \mathsf{x} : (f,t]\rangle \in M')
\end{aligned}
$$

Fig. 52. Auxiliary definitions in proving semantics equivalence

Lemma B.14 (Switch point forwarding - prc aux).

$$
\begin{aligned}
& \forall W, W', W'', n. \\
& \quad \mathsf{NAStep}^n(W, W') \wedge W' \xRightarrow{prc} \triangleright\, W'' \\
& \implies \exists W_0, \mathsf{t}.\ \mathsf{PRCStep}^*(W^{\mathsf{t}}, W_0) \wedge W_0 \xRightarrow{na/\mathsf{sw}} \triangleright^*\, W''
\end{aligned}
$$

Lemma B.15 (NA step delay - prc).

$$
\begin{aligned}
& \forall W, W_0, W_1, \mathsf{t}_0. \\
& \quad W \xRightarrow{na} \triangleright\, W_0 \wedge \mathsf{PRCStep}^n(W_0^{\mathsf{t}_0}, W_1) \\
& \implies \exists W_2, \mathsf{t}, \mathsf{t}_1. \\
& \qquad \mathsf{PRCStep}^*(W_0^{\mathsf{t}}, W_2) \wedge W_2^{\mathsf{t}_1} \xRightarrow{na} \triangleright\, W_1^{\mathsf{t}_1}
\end{aligned}
$$

Lemma B.16 (NA step delay - atomic step).

$$
\begin{aligned}
& \forall W, W_0, \mathsf{t}_0. \\
& \quad W \xRightarrow{na} \triangleright\, W_0 \wedge W_0^{\mathsf{t}_0} \xRightarrow{at} \triangleright\, W' \\
& \implies \exists W_1, W_2. \\
& \qquad \mathsf{PRCStep}^*(W, W_1) \wedge W_1^{\mathsf{t}_0} \xRightarrow{at} \triangleright\, W_2 \wedge W_2 \xRightarrow{na/\mathsf{sw}} \triangleright^*\, W'
\end{aligned}
$$

Proof. In this proof, we assume that the program configuration is well-defined. From the premises, we have the following.

$$W \xRightarrow{na} \triangleright\, W_0 \tag{1}$$

$$W_0^{\mathsf{t}_0} \xRightarrow{at} \triangleright\, W' \tag{2}$$

We let $W = (\mathcal{TP}, \mathsf{t}, \mathcal{S}, M)^{\iota}$ and have $\mathsf{wdSt}(\mathcal{TP}, \mathcal{S}, M)$. We unfold (1) and have that there exist TS_0, $\mathcal{S}_0$ and M_0 such that:

$$\iota \vdash (\mathcal{TP}(\mathsf{t}), \mathcal{S}, M) \xrightarrow{na}{}^{+} (TS_0, \mathcal{S}_0, M_0) \tag{1.1}$$

$$\mathsf{consistent}(TS_0, \mathcal{S}_0, M_0, \iota) \tag{1.2}$$

$$\mathcal{TP}_0 = \mathcal{TP}\{\mathsf{t} \rightsquigarrow TS_0\} \tag{1.3}$$

$$W_0 = (\mathcal{TP}_0, \mathsf{t}, \mathcal{S}_0, M_0)^{\iota} \tag{1.4}$$

We unfold (2). We have that there exist TS_0', $\mathcal{S}_0'$, M_0', TS', $\mathcal{S}'$ and M' such that:

$$\iota \vdash (\mathcal{TP}_0(\mathsf{t}_0), \mathcal{S}_0, M_0) \xrightarrow{prc}{}^{*} (TS_0', \mathcal{S}_0', M_0') \tag{2.1}$$

$$\iota \vdash (TS_0', \mathcal{S}_0', M_0') \xrightarrow{\mathsf{atmBlk}}{}^{+} (TS', \mathcal{S}', M') \tag{2.2}$$

$$\mathsf{consistent}(TS', \mathcal{S}', M', \iota) \tag{2.3}$$

$$W' = (\mathcal{TP}_0\{\mathsf{t}_0 \rightsquigarrow TS'\}, \mathsf{t}_0, \mathcal{S}', M')^{\iota} \tag{2.4}$$

We discuss whether t equals to t_0.

- We first consider $t = t_0$. We apply Lemma. B.17 on (1.1) and (2.1) and have that there exist TS_1, $\mathcal{S}_1$ and M_1 such that:

$$\iota \vdash (\mathcal{TP}(t), \mathcal{S}, M) \xrightarrow{prc}{}^{*} (TS_1, \mathcal{S}_1, M_1) \tag{3}$$

$$\iota \vdash (TS_1, \mathcal{S}_1, M_1) \xrightarrow{na}{}^{*} (TS_0', \mathcal{S}_0', M_0') \tag{4}$$

From (3), (4), (2.2) and (2.3), we have the following.

$$(\mathcal{TP}, t, \mathcal{S}, M)^{\iota} \overset{at}{\Longrightarrow}\triangleright (\mathcal{TP}\{t \rightsquigarrow TS'\}, t, \mathcal{S}', M')^{\iota} \tag{5}$$

We finish the proof of such case.

- Then, we consider that $t \neq t_0$. We apply Lemma. B.18 on (1.1) and there exist TS_1 and M_1 such that:

$$\iota \vdash (\mathcal{TP}(t), \mathcal{S}, M) \xrightarrow{\mathsf{prm}}{}^{*} (TS_1, \mathcal{S}_1, M_1) \tag{6}$$

$$\iota \vdash (TS_1, \mathcal{S}_1, M_1) \xrightarrow{na}{}^{+} (TS_0, \mathcal{S}_0, M_0) \tag{7}$$

$$M_1 \approx_{\mathsf{at}}^{\iota} M_0 \tag{8}$$

We apply Lemma. B.19 on (7), (8) and (1.2) and have the following.

$$\mathsf{consistent}(TS_1, \mathcal{S}_1, M_1, \iota) \tag{9}$$

We apply Lemma. B.20 on (7), (8) and (2.1), we have that there exists M_1'' such that:

$$\iota \vdash (\mathcal{TP}_0(t_0), \mathcal{S}_1, M_1) \xrightarrow{prc}{}^{*} (TS_0', \mathcal{S}_1, M_1'') \tag{10}$$

$$\iota \vdash (TS_1, \mathcal{S}_1, M_1'') \xrightarrow{na}{}^{+} (TS_0, \mathcal{S}_0', M_0') \tag{11}$$

$$M_1'' \approx_{\mathsf{at}}^{\iota} M_0' \tag{12}$$

We apply Lemma. B.21 on (11), (12) and (2.2) and have that there exists M'' such that:

$$\iota \vdash (TS_0', \mathcal{S}_1, M_1'') \xrightarrow{\mathsf{atmBlk}}{}^{+} (TS', \mathcal{S}', M'') \tag{13}$$

$$\iota \vdash (TS_1, \mathcal{S}', M'') \xrightarrow{na}{}^{+} (TS_0, \mathcal{S}', M') \tag{14}$$

$$M'' \approx_{\mathsf{at}}^{\iota} M' \tag{15}$$

From (6) and (9), we have the following.

$$\mathsf{PRCStep}^{*}((\mathcal{TP}, t, \mathcal{S}, M)^{\iota}, (\mathcal{TP}\{t \rightsquigarrow TS_1\}, t, \mathcal{S}_1, M_1)^{\iota}) \tag{16}$$

Since we have "$\mathsf{wdSt}(\mathcal{TP}, \mathcal{S}, M)^{\iota}$", according to (3), we have "$\mathsf{wdSt}(\mathcal{TP}\{t \rightsquigarrow TS_1\}, \mathcal{S}, M_1)$". Then, according to (10) and (13), we have "$\mathsf{wdSt}(\mathcal{TP}\{t \rightsquigarrow TS_1, t_0 \rightsquigarrow TS'\}, \mathcal{S}', M'')$". Thus, we have the following.

$$TS'.P \subseteq M'' \tag{17}$$

By applying Lemma. B.22 on (2.3), (15) and (17), we have the following.

$$\mathsf{consistent}(TS', \mathcal{S}', M'', \iota) \tag{18}$$

From (10), (13) and (18), we have the following.

$$(\mathcal{TP}\{t \rightsquigarrow TS_1\}, t, \mathcal{S}_1, M_1)^{\iota} \overset{at}{\Longrightarrow}\triangleright (\mathcal{TP}\{t \rightsquigarrow TS_1, t_0 \rightsquigarrow TS'\}, t_0, \mathcal{S}', M'')^{\iota} \tag{19}$$

By applying Lemma. B.23 on (1.2), (2.1) and (2.2), we have the following.

$$\mathsf{consistent}(TS_0, \mathcal{S}', M', \iota) \tag{20}$$

From (14) and (20), we have the following.

$$(\mathcal{TP}\{\mathrm{t} \rightsquigarrow TS_1, \mathrm{t}_0 \rightsquigarrow TS'\}, \mathrm{t}_0, \mathcal{S}', M'')^\iota \overset{na}{\Longrightarrow\!\triangleright} (\mathcal{TP}\{\mathrm{t} \rightsquigarrow TS_0, \mathrm{t}_0 \rightsquigarrow TS'\}, \mathrm{t}_0, \mathcal{S}', M')^\iota \tag{21}$$

From (16), (19) and (21), we finish the proof.

□

Lemma B.17 (prc steps forwarding in the same thread).

$$\begin{aligned}
&\forall TS, \mathcal{S}, M, TS', \mathcal{S}', M', TS'', \mathcal{S}'', M'', \iota, n_1, n_2.\\
&\quad \iota \vdash (TS, \mathcal{S}, M) \xrightarrow{na}{}^{n_1} (TS', \mathcal{S}', M') \wedge\\
&\qquad \iota \vdash (TS', \mathcal{S}', M') \xrightarrow{prc}{}^{n_2} (TS'', \mathcal{S}'', M'')\\
&\Longrightarrow \exists TS_0, \mathcal{S}_0, M_0.\\
&\qquad \iota \vdash (TS, \mathcal{S}, M) \xrightarrow{prc}{}^{*} (TS_0, \mathcal{S}_0, M_0) \wedge\\
&\qquad\quad \iota \vdash (TS_0, \mathcal{S}_0, M_0) \xrightarrow{na}{}^{*} (TS'', \mathcal{S}'', M'')
\end{aligned}$$

Lemma B.18 (promises forwarding non-atomic steps).

$$\begin{aligned}
&\forall TS, \mathcal{S}, M, TS', \mathcal{S}', M', \iota, n.\\
&\quad \iota \vdash (TS, \mathcal{S}, M) \xrightarrow{na}{}^{n} (TS', \mathcal{S}', M')\\
&\Longrightarrow \exists TS_0, M_0.\ \iota \vdash (TS, \mathcal{S}, M) \xrightarrow{\mathsf{prm}}{}^{*} (TS_0, \mathcal{S}, M_0) \wedge\\
&\qquad\quad \iota \vdash (TS_0, \mathcal{S}, M_0) \xrightarrow{na}{}^{n} (TS', \mathcal{S}', M') \wedge M_0 \approx^{\iota}_{\mathsf{at}} M'
\end{aligned}$$

Proof. Prove by induction on n. □

Lemma B.19 (consistency forwarding non-atomic steps).

$$\begin{aligned}
&\forall TS, \mathcal{S}, M, TS', \mathcal{S}', M', \iota, n.\\
&\quad \iota \vdash (TS, \mathcal{S}, M) \xrightarrow{na}{}^{n} (TS', \mathcal{S}', M') \wedge\\
&\qquad M \approx^{\mathsf{at}}_{\iota} M' \wedge \mathsf{consistent}(TS', \mathcal{S}', M', \iota)\\
&\Longrightarrow \mathsf{consistent}(TS, \mathcal{S}, M, \iota)
\end{aligned}$$

Lemma B.20 (non-atomic steps and prc step reordering).

$$\begin{aligned}
&\forall TS_1, \mathcal{S}, M, TS_1', \mathcal{S}_1, M_1, TS_2, TS_2', \mathcal{S}_2, M_2, n_1, n_2.\\
&\quad \iota \vdash (TS_1, \mathcal{S}, M) \xrightarrow{na}{}^{n_1} (TS_1', \mathcal{S}_1, M_1) \wedge M \approx^{\iota}_{\mathsf{at}} M_1 \wedge\\
&\qquad \iota \vdash (TS_2, \mathcal{S}_1, M_1) \xrightarrow{prc}{}^{n_2} (TS_2', \mathcal{S}_2, M_2)\\
&\Longrightarrow \exists M_{20}.\ \iota \vdash (TS_2, \mathcal{S}, M) \xrightarrow{prc}{}^{n_2} (TS_2', \mathcal{S}, M_{20}) \wedge\\
&\qquad\quad \iota \vdash (TS_1, \mathcal{S}, M_{20}) \xrightarrow{na}{}^{n_1} (TS_1', \mathcal{S}_2, M_2) \wedge M_{20} \approx^{\iota}_{\mathsf{at}} M_2
\end{aligned}$$

Lemma B.21 (non-atomic steps and atomic-block step reordering).

$$\begin{aligned}
&\forall TS_1, \mathcal{S}, M, TS_1', \mathcal{S}_1, M_1, TS_2, TS_2', \mathcal{S}_2, M_2, n_1, n_2.\\
&\quad \iota \vdash (TS_1, \mathcal{S}, M) \xrightarrow{na}{}^{n_1} (TS_1', \mathcal{S}_1, M_1) \wedge M \approx^{\iota}_{\mathsf{at}} M_1 \wedge\\
&\qquad \iota \vdash (TS_2, \mathcal{S}_1, M_1) \xrightarrow{\mathsf{atmBlk}}{}^{n_2} (TS_2', \mathcal{S}_2, M_2)\\
&\Longrightarrow \exists M_{20}.\ \iota \vdash (TS_2, \mathcal{S}, M) \xrightarrow{\mathsf{atmBlk}}{}^{n_2} (TS_2', \mathcal{S}_2, M_{20}) \wedge\\
&\qquad\quad \iota \vdash (TS_1, \mathcal{S}_2, M_{20}) \xrightarrow{na}{}^{n_1} (TS_1', \mathcal{S}_2, M_2) \wedge M_{20} \approx^{\iota}_{\mathsf{at}} M_2
\end{aligned}$$

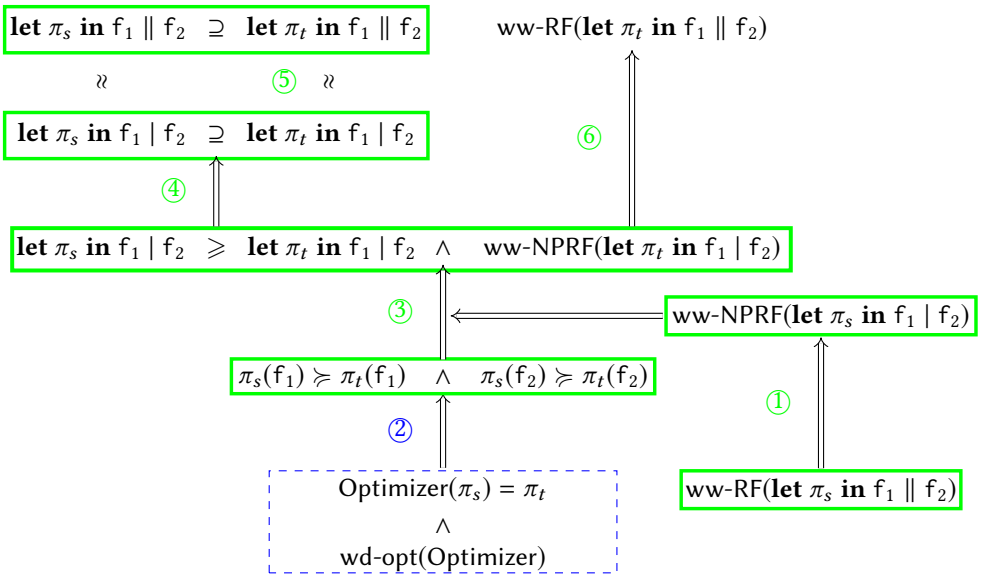


Fig. 53. Proof sketch

Lemma B.22 (mem approx equal consistency preserving).

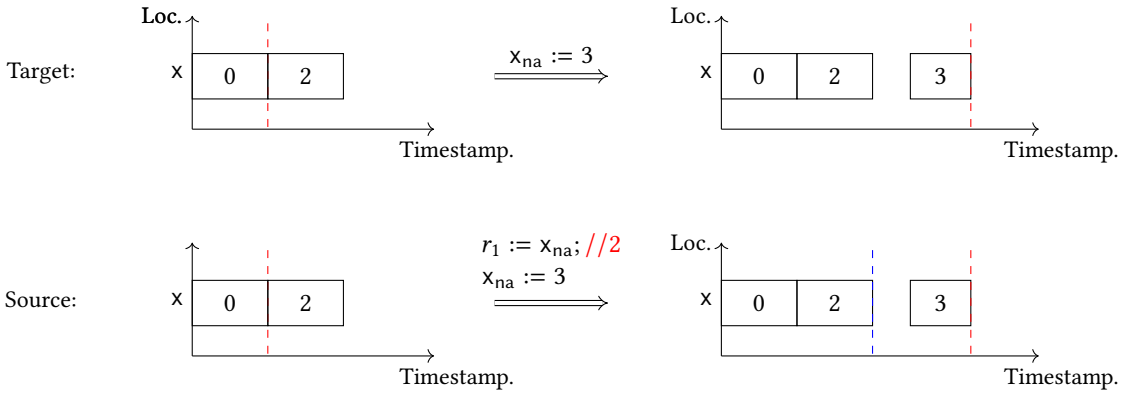
$$\begin{aligned}&\forall TS, \mathcal{S}, M, \iota, M_0.\\ &\quad \mathsf{consistent}(TS, \mathcal{S}, M, \iota) \wedge M \approx^{\iota}_{\mathsf{at}} M_0 \wedge TS.P \subseteq M_0\\ &\implies \mathsf{consistent}(TS, \mathcal{S}, M_0, \iota)\end{aligned}$$

Lemma B.23 (consistent forwarding).

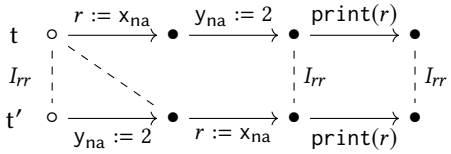
$$\begin{aligned}&\forall TS, \mathcal{S}, M, TS_0, n, \mathcal{S}', M', \iota.\\ &\quad \mathsf{consistent}(TS, \mathcal{S}, M, \iota) \wedge TS.P \# TS_0.P \wedge\\ &\quad \iota \vdash (TS_0, \mathcal{S}, M) \longrightarrow^n (\_, \mathcal{S}', M')\\ &\implies \mathsf{consistent}(TS, \mathcal{S}', M', \iota)\end{aligned}$$

Equivalence between the promising semantics and the non-preemptive semantics. We show the correctness proof of Lemma. 6.1 in the following.

Proof. We finish the proof by applying Lemma. B.3 and 6.1. □



$r := x_{na}; y_{na} := 2; \texttt{print}(r);$ $\overset{reorder}{\rightsquigarrow}$ $y_{na} := 2; r := x_{na}; \texttt{print}(r);$



where $I_{rr}(\varphi, \iota, (\mathcal{S}_t, M_t, \mathcal{S}_s, M_s))$
$\triangleq (M_t = M_s \wedge \mathcal{S}_t = \mathcal{S}_s)$

REFERENCES

[1] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Webe. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'11)*. 55–66.

[2] Soham Chakraborty and Viktor Vafeiadis. 2016. Validating optimizations of concurrent C/C++ programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO '16)*. 216–226.

[3] Soham Chakraborty and Viktor Vafeiadis. 2017. Formalizing the concurrency semantics of an LLVM fragment. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17)*. 100–110.

[4] Ševčík. 2011. Safe Optimisations for Shared-Memory Concurrent Programs. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*.

[5] Minki Cho, Sung-Hwan Lee, Chung-Kil Hur, and Ori Lahav. 2021. Promising 2.0: Global Optimizations in Relaxed Memory Concurrency. In *Proceedings of the 42nd annual ACM SIGPLAN conference on Programming Languages Design and Implementation (PLDI '21)*.

[6] CompCert Developers. 2020. CompCert-3.7. http://compcert.inria.fr/release/compcert-3.7.tgz

[7] Hanru Jiang, Hongjin Liang, Siyang Xiang, Junpeng Zha, and Xinyu Feng. 2019. Towards Certified Separate Compilation for Concurrent Programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*.

[8] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-Memory Concurrency. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '17)*.

[9] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17)*. 618–632.

[10] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* C-28, 9 (1979), 690–691.

[11] Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. 2020. Promising 2.0: Global Optimizations in Relaxed Memory Concurrency. In *Proceedings of the 41st annual ACM SIGPLAN conference on Programming Languages Design and Implementation (PLDI '20)*.

[12] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. 187–196.

[13] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2017. Promising Compilation to ARMv8 POP. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 74. 22:1–22:28.

[14] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2018. Bridging the gap between programming languages and hardware weak memory models. In *Proceedings of the ACM on Programming Languages (POPL'18)*, Vol. 3. 1–32.

[15] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3 (2013), 22.

[16] Steven S.Muchnick. 1997. *Advanced Compiler Design Implementation.* Academic Press.

[17] Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. 2020. CompCertM: CompCert with C-assembly linking and lightweight modular verification. *Proceedings of the ACM on Programming Languages* 4, 23 (2020), 1–31. Issue POPL.

[18] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. 209–220.

[19] Yuting Wang, Pierre Wilke, and Zhong Shao. 2019. An abstract stack based approach to verified compositional compilation to machine code. *Proceedings of the ACM on Programming Languages* 3, 62 (2019), 1–30. Issue POPL.