

On Speculative Computation and Thread Safe Programming*

G erard Boudol Gustavo Petri

INRIA Sophia Antipolis

{Gerard.Boudol,Gustavo.Petri}@inria.fr

Abstract

We propose a formal definition for (valid) speculative computations, which is independent of any implementation technique. By speculative computations we mean optimization mechanisms that rely on relaxing the flow of execution in a given program, and on guessing the values read from pointers in the memory. Our framework for formalizing these computations is the standard operational one that is used to describe the semantics of programming languages. In particular, we introduce speculation contexts, that generalize classical evaluation contexts, and allow us to deal with out of order (or parallel) computations. We show that the standard DRF guarantee, asserting that data race free programs are correctly implemented in a relaxed semantics, fails with speculative computations, but that a similar guarantee holds for programs that are free of data races in the speculative semantics. We then introduce a language featuring an explicit distinction between shared and private variables, and show that there is a translation, guided by a type and effect system, that transforms a program written in this language into speculatively data race free code, which is therefore robust against aggressive optimizations.

1. Introduction

Speculative computation [12, 22] is an implementation technique that aims at speeding up the execution of programs, by computing pieces of code out of order or in parallel, without being sure that these computations are actually needed. We shall actually use the terminology “*speculative computation*” in a very broad sense here: we try to capture the optimization techniques that rely on executing the code as it is, but relaxing the flow of control, not necessarily following the order prescribed by the reference operational semantics. Some keywords here are: pipelining, instruction level parallelism, out-of-order execution, branch prediction, thread level speculation, etc. (we shall not cite any particular paper from the huge literature on these classical topics). We also include relaxed memory models [2] into this picture, though not those that try to capture compiler optimizations, that transform the code on the basis of semantical reasoning (see [5, 27, 31]).

The idea of optimizing by computing in parallel is quite old, but the work that has been done so far on this topic is almost exclusively concerned with implementation techniques, either from the hardware or the software point of view. These implementations are quite often complex, as speculations are not always correct, and need to be aborted or undone in some cases. Due to this complexity perhaps, the notion of a *valid speculation* does not seem to have been formally defined anywhere, except in some particular cases that we will mention below. Nevertheless, the various implementations of speculative techniques are generally considered correct as regards the semantics of sequential programs. This is no longer the case for multithreaded applications running on multicore architectures. It is well-known for instance (see the survey [2]) that

relaxed memory models do not preserve the standard interleaving semantics – also known as “sequential consistency” [23] in the area of memory models. We shall see that even worse phenomena arise from speculative computation, and in particular the failure of the “DRF guarantee” [3, 17, 27] (see below). Then a formal understanding of (valid) speculations would be useful, to guide the intuition of the programmer and help him writing safe multithreaded code, and to provide a basis for analyzing and verifying concurrent programs running on optimized multiprocessor architectures.

In this paper we follow and extend the approach presented in [10], that is, we define, using a pretty standard operational style, the speculative semantics of an expressive language, namely a call-by-value λ -calculus with mutable state and threads. Our formalization relies on extending the usual notion of an evaluation context [14], and using *value prediction* [16, 25] as regards the values read from the memory. By introducing *speculation contexts*, we are able to formalize out of order executions, as in relaxed memory models, and also *branch prediction* [33], allowing to compute in the alternatives of a conditional branching construct. A particular case of out of order computation is provided by the future construct of Multilisp [20]. Examples of speculations are the following sequences, where we use ML’s notation $!p$ for dereferencing a pointer:

$$\begin{aligned} r := !p; q := tt &\rightarrow r := !p; () \\ &\rightarrow r := tt; () \end{aligned}$$

where the write to q is reordered with respect to the read of p , which “guesses” the value tt , and

$$\begin{aligned} (\text{if } !p \text{ then } q := tt \text{ else } ()) &\rightarrow (\text{if } !p \text{ then } () \text{ else } ()) \\ &\rightarrow (\text{if } tt \text{ then } () \text{ else } ()) \\ &\rightarrow () \end{aligned}$$

where the assignment in the first branch is done speculatively, and the value tt is guessed for p . These two speculations are intuitively correct, but there are also incorrect speculations, such as the one we get from the expression of the second example, if $!p$ returns ff instead of tt : in this case the assignment $q := tt$ should not be done.

The central definition in this paper is the one of a *valid speculative computation*. Roughly speaking, a thread’s speculation is valid if it can be proved equivalent to a normal order computation. The equivalence we use here is the *permutation of transitions equivalence* introduced by Berry and L evy in [6], stating that independent steps can be performed in any order (or in parallel) without essentially changing the computation. One can see, for instance, that the two speculations above are valid, by executing the same operations in normal order (more details below). In an implementation setting we would say that a speculation is allowed to *commit* in the case it is valid, but one should notice that our formulation is fully independent from any implementation mechanism. In fact, we shall not in this paper study the problem of showing that an implementation

* Work partially supported by the ANR-SETI-06-010 grant.

is correct in the sense that it allows only valid speculations to be performed.

To the best of our knowledge, the notion of a (valid) speculation has not been previously stated in a formal way. In this respect, the work that is the closest to ours is the one on the mathematical semantics of Multilisp’s future construct, starting with the work [15] of Flanagan and Felleisen. This was later extended by Moreau in [29] to deal with mutable state and continuations (extending the work in [21] as regards the latter). A similar work regarding JAVA has been done by Jagannathan and colleagues, dealing with mutable state [34] and exceptions [30]. However, all these works on the future construct aim at preserving the sequential semantics, and therefore they are not concerned with shared memory concurrency. Moreover, they do not include branch prediction and value prediction.

Continuing with the two examples above, one can see that with the thread system

$$r := !p; q := tt \parallel r' := !q; p := tt$$

and starting with a state where $p = ff = q$, one can get as an outcome of a valid speculative computation a state where $r = tt = r'$. This cannot be obtained by a standard interleaving execution, but is allowed in memory models where reads can be reordered with respect to subsequent memory operations, a property symbolically called $\mathbf{R} \rightarrow \mathbf{RW}$, according to the terminology of [2]. One could check that most of the allowed behaviors (the so called “litmus tests”) in weak memory models can also be obtained by speculative computations, thus suggesting that the latter offers a very relaxed semantical model, supporting many optimization techniques. Extending the second example given above, one can see that with the thread system

$$\begin{array}{l} p := ff; \\ (\text{if } !p \text{ then } q := tt \text{ else } ()) \end{array} \parallel \begin{array}{l} q := ff; \\ (\text{if } !q \text{ then } p := tt \text{ else } ()) \end{array}$$

one can get the outcome $p = tt = q$, by speculatively performing, after the initial assignments, the two assignments $q := tt$ and $p := tt$, thus justifying the branch prediction made in the other thread (see [18] Section 17.4.8, and [7] for similar examples). This example, though not very interesting from a programming point of view, shows that the well-known “DRF guarantee” – asserting that programs free of data races, with respect to the interleaving semantics, are correctly executed in the optimized semantics –, does *not* hold in the case of speculative computations. Let us see another example, which looks more like a standard idiom, for a producer-consumer scenario. In this example, we use a construct (with ℓ do e) to ensure mutual exclusion, by acquiring a lock ℓ , computing e and, upon termination, releasing ℓ . Then with the two threads

$$\begin{array}{l} \text{data} := 1; \\ (\text{with } \ell \text{ do } \text{flag} := tt) \end{array} \parallel \begin{array}{l} \text{while not } (\text{with } \ell \text{ do } !\text{flag}) \text{ do skip}; \\ r := !\text{data} \end{array}$$

if initially $\text{data} = 0$ and $\text{flag} = ff$, we can speculate that $!\text{data}$ in the second thread returns 0, and therefore get an unexpected value for r (the other instructions being processed in the normal way). This is, according to our definition, a valid speculation, and this provides another example of the failure of the DRF guarantee in the speculative semantics.

From the programmer’s viewpoint, a question therefore arises: for which programs is the speculative semantics *correct*, not introducing unexpected behaviors? We shall qualify as *robust* such programs, and we will see in particular that any purely sequential program, that does not spawn any thread, is robust. That is, the speculative semantics is a correct implementation for sequential programs. As we have seen, data race free concurrent programs are not necessarily robust. In this paper we show that *speculatively* data race free programs *are* robust – this is our main result. Here

speculatively DRF means that there is no data race occurring in the speculative semantics, where a data race is, as usual, the possibility of performing concurrent accesses, one of them being a write, to the same memory location.

Again from the programmer’s viewpoint, this result is not fully satisfactory, because he/she should not have to think about speculative execution – only about the reference interleaving semantics. There are several possible ways out of this. One would be to see whether type (and effect) systems for data race prevention, such as [1, 11] for instance, also ensure the speculatively DRF property, and to adapt them if this is not the case (we conjecture however that this is indeed the case). In this paper we explore another way. The idea, which is not totally new (see [19] for instance) and not that surprising, is to introduce in the programming language an explicit distinction between shared and unshared variables. At this level, the programmer does not have to know about locks, but we provide a synchronization construct for turning a shared variable into a private one, with a delimited scope. That is, the thread using this construct temporarily *owns* the shared variable, having an exclusive access to it for a while, and releases it upon termination of this construct.

There is a simple translation from this language to a target language where all the pointers are implicitly shared, and where the semantics is speculative. In this translation, each access to a shared variable of the source language is protected with a lock univocally associated with it. Then a shared variable in our language is similar to a “*volatile*” variable [18]. The translation is guided by a novel type and effect system ([26]; see [13] for a different type system with similar purposes), which guarantees that a thread does not access a private memory location that it does not own, that is, any unshared variable is indeed used by at most one thread, and cannot be the subject of a data race. Accesses to such private references can therefore be subject to speculation. Our second main result is that any typable program in this source language is robust, its translation being speculatively DRF, and that the translation preserves the source semantics, that is the standard interleaving semantics. In [9] we have shown how to give a *deadlock-free* semantics to such a language, again relying on a type and effect system. Our language is therefore a step towards *thread safe*, or *modular* concurrent programming: by writing typable code, the programmer is guaranteed not to introduce dangerous data races nor deadlocks, even when his/her code is composed with other (typable) threads, or uses (typable) modules.

Obviously, this is a proposal that has to be assessed on an experimental ground, and as we said some other ways are likely to exist to write concurrent programs that are robust against aggressive optimizations. Whatever these other ways may be, we think that the formal definition of (valid) speculative computations we set up in this paper provides a sound basis for such investigations.

2. The (Target) Language

The intermediate, or target language, supporting speculative computations, is an imperative call-by-value λ -calculus, with boolean values and conditional branching, enriched with thread creation and a construct for ensuring mutual exclusion. In this language, all the references are implicitly shared. We shall later assume that the language also contains some record constructs which, for simplicity, are not yet included. The syntax is:

$$\begin{array}{ll} e ::= & v \mid (e_0 e_1) \qquad \text{expressions} \\ & \mid (\text{if } e \text{ then } e_0 \text{ else } e_1) \\ & \mid (\text{ref } e) \mid (!e) \mid (e_0 := e_1) \\ & \mid (\text{thread } e) \mid (\text{with } \ell \text{ do } e) \mid (\text{new } \ell \text{ in } e) \\ v ::= & x \mid \lambda x e \mid tt \mid ff \mid () \qquad \text{values} \end{array}$$

where ℓ is a *lock*, that is a name from a given infinite set \mathcal{Locks} . The construct $(\text{new } \ell \text{ in } e)$ is a binder for the name ℓ in e . As usual, λ is a binder for the variable x in $\lambda x e$, and we shall consider expressions up to α -conversion, that is up to the renaming of bound variables. The capture-avoiding substitution of e_0 for the free occurrences of x in e_1 is denoted $\{x \mapsto e_0\}e_1$. We shall use some standard abbreviations like $(\text{let } x = e_0 \text{ in } e_1)$ for $(\lambda x e_1 e_0)$, which is also denoted $e_0 ; e_1$ whenever x does not occur free in e_1 .

To state the operational semantics of the language, we have to extend it with run-time constructs, in two ways. First, we introduce *references* (sometimes also referred to as memory locations, memory addresses, or pointers) p, q, \dots that are names from a given infinite set \mathcal{Ref} . These are (run-time) values. Then we use the construct $(\text{holding } \ell \text{ do } e)$ to hold a lock for e . As it is standard with languages involving concurrency with shared variables, we follow a *small-step* style to describe the operational semantics, where an atomic transition consists in reducing a *redex* (reducible expression) within an *evaluation context*, while possibly performing a side effect. The syntax is then extended and enriched as follows:

$p, q \dots$	\in	\mathcal{Ref}	<i>references</i>
v	$::=$	$\dots \mid p$	<i>run-time values</i>
e	$::=$	$\dots \mid (\text{holding } \ell \text{ do } e)$	<i>run-time expressions</i>
u	$::=$	$(\lambda x e v)$	<i>redexes</i>
		$\mid (\text{if } tt \text{ then } e_0 \text{ else } e_1)$	
		$\mid (\text{if } ff \text{ then } e_0 \text{ else } e_1)$	
		$\mid (\text{ref } v) \mid (!p) \mid (p := v)$	
		$\mid (\text{thread } e) \mid (\text{with } \ell \text{ do } e)$	
		$\mid (\text{holding } \ell \text{ do } v) \mid (\text{new } \ell \text{ in } e)$	
\mathbf{E}	$::=$	$\square \mid \mathbf{E}[\mathbf{F}]$	<i>evaluation contexts</i>
\mathbf{F}	$=$	$(\square e) \mid (v \square)$	<i>frames</i>
		$\mid (\text{if } \square \text{ then } e_0 \text{ else } e_1)$	
		$\mid (\text{ref } \square) \mid (!\square) \mid (\square := e) \mid (v := \square)$	
		$\mid (\text{holding } \ell \text{ do } \square)$	

As usual, we denote by $\mathbf{E}[e]$ the expression resulting from filling the hole in \mathbf{E} by e . Every expression of the (run-time) language is either a value, or a redex in a position to be reduced, or faulty. More precisely, let us say that an expression is *faulty* if it has one of the following forms:

- (ve) where the value v is not a function $\lambda x e'$;
- $(\text{if } v \text{ then } e_0 \text{ else } e_1)$ where the value v is not a boolean value, tt or ff ;
- $(!v)$ or $v := v'$ where the value v is not a reference.

Then we have:

LEMMA 2.1. *For any expression e of the run-time language, either e is a value, or there is a unique evaluation context \mathbf{E} and a unique expression e' which either is a redex, or is faulty, such that $e = \mathbf{E}[e']$.*

(The proof, by induction on e , is immediate.)

To define speculative computations, we extend the class of standard evaluation contexts by introducing *speculation contexts*, defined as follows:

Σ	$::=$	$\square \mid \Sigma[\Phi]$	<i>speculation contexts</i>
Φ	$::=$	\mathbf{F}	<i>speculation frames</i>
		$\mid (e \square) \mid (\lambda x \square e)$	
		$\mid (\text{if } e \text{ then } \square \text{ else } e_1) \mid (\text{if } e \text{ then } e_0 \text{ else } \square)$	
		$\mid (e := \square)$	

Let us comment briefly on the speculation contexts. With frames of the shape $(\lambda x \square e)$, one can for instance compute e_1 in the expression $(\lambda x e_1 e_0)$ – hence in $(\text{let } x = e_0 \text{ in } e_1)$ and $e_0 ; e_1$ in particular – whereas in a normal order computation one has to compute e_0 first. This is similar to a *future* expression $(\text{let } x = \text{future } e_0 \text{ in } e_1)$ [15], where e_1 is computed *in advance*, or *in parallel* with e_0 . With the frames $(\text{if } e \text{ then } \square \text{ else } e_1)$ and $(\text{if } e \text{ then } e_0 \text{ else } \square)$, one is allowed to compute in a branch (or in both branches) of a conditional construct, without knowing the value of the condition, again computing in advance (or in parallel) with respect to the normal order. This is known as “*branch prediction*” [33]. Notice that, by contrast, the construct $(\text{with } \ell \text{ do } e)$ acts as a “speculation barrier,” that is, $(\text{with } \ell \text{ do } \square)$ is not a speculation frame. Indeed, the purpose of acquiring a lock is to separate side-effecting operations. We could allow pure (i.e. without side effect) speculations inside such a construct¹, but this would complicate the technical developments, with no added value, since, as we shall see, we can always speculatively acquire a lock (but not speculatively release it).

To define the semantics of locking, which allows for *reentrant* locks, we shall use the set, denoted $[\Sigma]$, of locks held in the context Σ , defined as follows:

$$\begin{aligned} [\square] &= \emptyset \\ [\Sigma[\Phi]] &= [\Sigma] \cup [\Phi] \end{aligned}$$

where

$$[\Phi] = \begin{cases} \{\ell\} & \text{if } \Phi = (\text{holding } \ell \text{ do } \square) \\ \emptyset & \text{otherwise} \end{cases}$$

Speculative computations are defined in two stages: first we define *speculations*, that are abstract computations of a given thread – abstract in the sense that the state, made of a memory, a set of busy locks, and a multiset of threads, is ignored at this stage. We can regard these as attempts to perform some computation, with no real side effect. Then we shall compose such speculations by interleaving them, now taking at this stage the global state into account. In order to do so, it is convenient to formalize speculations as labeled transitions, explicitly indicating *what* reduction occurs, that is what is the *action* performed at each step. There are several kinds of actions: performing a β -reduction, denoted β , choosing a branch in a conditional construct (\sphericalangle and \sphericalright), creating a new reference p in the store with some initial value $(\nu_{p,v})$, reading $(\text{rd}_{p,v})$ or writing $(\text{wr}_{p,v})$ a reference, spawning a new thread (spw_e) , acquiring $(\widehat{\ell})$ or releasing $(\widehat{\ell})$ a lock ℓ , and creating a new lock (ν_ℓ) . Then the syntax of actions is as follows:

$$\begin{aligned} a &::= \beta \mid \sphericalangle \mid \sphericalright \\ &\mid \nu_{p,v} \mid \text{rd}_{p,v} \mid \text{wr}_{p,v} \\ &\mid \mu \mid \widehat{\ell} \mid \nu_\ell \mid b \\ b &:= \text{spw}_e \mid \widehat{\ell} \end{aligned}$$

The action μ stands for taking a lock that is already held. We denote by \mathcal{Act} the set of actions, and by \mathcal{B} the subset of b actions.

In order to define *valid* speculations, we shall also need to explicitly indicate in the semantics *where* actions are performed. To this end, we introduce the notion of an *occurrence*, which is a sequence over a set of symbols, each associated with a frame, denoting a path from the root of the expression to the redex that is evaluated at some step. In the case of a frame $(e \square)$, it is convenient to distinguish the case where this is a “normal” frame, that is, when e is a value, from the case where this is a true speculation frame.

¹ by enriching the conflict relation, see below.

$$\begin{array}{lcl}
\Sigma[(\lambda x e v)] & \xrightarrow[\textcircled{\Sigma}]{\beta} & \Sigma[\{x \mapsto v\}e] & \mathbf{E}[(\text{thread } e)] & \xrightarrow[\textcircled{\mathbf{E}}]{\text{SPW}_e} & \mathbf{E}[\emptyset] \\
\Sigma[(\text{if } tt \text{ then } e_0 \text{ else } e_1)] & \xrightarrow[\textcircled{\Sigma}]{\checkmark} & \Sigma[e_0] & \Sigma[(\text{with } \ell \text{ do } e)] & \xrightarrow[\textcircled{\Sigma}]{\mu} & \Sigma[e] & \ell \in [\Sigma] \\
\Sigma[(\text{if } ff \text{ then } e_0 \text{ else } e_1)] & \xrightarrow[\textcircled{\Sigma}]{\searrow} & \Sigma[e_1] & \Sigma[(\text{with } \ell \text{ do } e)] & \xrightarrow[\textcircled{\Sigma}]{\widehat{\ell}} & \Sigma[(\text{holding } \ell \text{ do } e)] & \ell \notin [\Sigma] \\
\Sigma[(\text{ref } v)] & \xrightarrow[\textcircled{\Sigma}]{\nu_{p,v}} & \Sigma[p] & \mathbf{E}[(\text{holding } \ell \text{ do } v)] & \xrightarrow[\textcircled{\mathbf{E}}]{\widehat{\ell}} & \mathbf{E}[v] \\
\Sigma[(!p)] & \xrightarrow[\textcircled{\Sigma}]{\text{rd}_{p,v}} & \Sigma[v] & \Sigma[(\text{new } \ell \text{ in } e)] & \xrightarrow[\textcircled{\Sigma}]{\nu_{\ell'}} & \Sigma[\{\ell \mapsto \ell'\}e] \\
\Sigma[(p := v)] & \xrightarrow[\textcircled{\Sigma}]{\text{wr}_{p,v}} & \Sigma[\emptyset] & & & &
\end{array}$$

Figure 1: Speculations

Then an occurrence is a sequence o over the set \mathcal{SOcc} below:

$$\begin{aligned}
\mathcal{Occ} &= \{(\square _), (\bullet \square), (\text{if } \square \text{ then } _ \text{ else } _), (\text{ref } \square), (!\square), \\
&\quad (\square := _), (v := \square), (\text{holding } \ell \text{ do } \square)\} \\
\mathcal{SOcc} &= \mathcal{Occ} \cup \{(_ \square), (\lambda x \square _), (\text{if } _ \text{ then } \square \text{ else } _), \\
&\quad (\text{if } _ \text{ then } _ \text{ else } \square), (_ := \square)\}
\end{aligned}$$

The occurrences $o \in \mathcal{Occ}^*$ are called *normal*. Notice that we do not consider $\lambda x \square$ as an occurrence. This corresponds to the fact that speculating inside a value is forbidden, except in the case of a function applied to an argument, that is $(\lambda x e_1 e_0)$ where speculatively computing e_1 is allowed (again we could relax this as regards pure speculations, but this would involve heavy technical complications). One then defines the occurrence $\textcircled{\Sigma}$, as the sequence of frames that points to the hole in Σ , that is:

$$\begin{aligned}
\textcircled{\square} &= \varepsilon \\
\textcircled{\Sigma[\Phi]} &= \textcircled{\Sigma} \cdot \textcircled{\Phi}
\end{aligned}$$

where

$$\begin{aligned}
\textcircled{(\square _)} &= (\square _) \\
\textcircled{(e \square)} &= \begin{cases} (\bullet \square) & \text{if } e \in \mathcal{Val} \\ (_ \square) & \text{otherwise} \end{cases} \\
\textcircled{(e := \square)} &= \begin{cases} (v := \square) & \text{if } e = v \in \mathcal{Val} \\ (_ := \square) & \text{otherwise} \end{cases}
\end{aligned}$$

and so on. We denote by $o \cdot o'$ the concatenation of the two sequences o and o' , and we say that o is a prefix of o' , denoted $o \leq o'$, if $o' = o \cdot o''$ for some o'' . If $o \not\leq o'$ and $o' \not\leq o$ then we say that o and o' are disjoint occurrences.

We can now define the “local” speculations, for a given (sequential) thread. This is determined independently of any context (memory or other threads), and without any real side effect. Speculations are defined as a small step semantics, namely labeled transitions

$$e \xrightarrow[o]{a} e'$$

where a is the action performed at this step and o is the occurrence at which the action is performed (in the given thread). These are defined in Figure 1. Speculating here means not only computing “in advance” (or “out-of-order”), but also *guessing* the values from the global context (the memory and the lock context). More precisely, the speculative character of this semantics is twofold. On the one hand, some computations are allowed to occur in speculation contexts Σ , like with future computations or branch prediction. On the other hand, the value resulting from a dereference operation $(!p)$, or the status of the lock in the case of a locking construct $(\text{with } \ell \text{ do } e)$, is “guessed”, or “predicted” – as regards loads from

the memory, this is known as *value prediction*, and was introduced in [16, 25]. These guessed values may be written by other threads, which are ignored at this stage. One should notice that the b actions are only allowed to occur from within an evaluation context, not a speculation context. However, one should also observe that an evaluation context can be modified by a speculation, while still being an evaluation context. This is typically the case of $(\square e)$ and $(\lambda x e \square)$ – hence in particular $(\text{let } x = \square \text{ in } e)$ and $\square; e _$, where one is allowed to speculate the execution of e ; this is also the case with $(\text{if } \square \text{ then } e_0 \text{ else } e_1)$ where one can speculate in a branch, that is in e_0 or e_1 . Then for instance with an expression of the form $(\text{holding } \ell \text{ do } e_0); e_1$, one can speculatively compute in e_1 before trying to release the lock ℓ and proceed with e_0 (a special case of this is the so-called “roach motel semantics,” see [4]). The following is a standard property:

LEMMA 2.2. *If $e \xrightarrow[o]{a} e'$ then $\{x \mapsto v\}e \xrightarrow[o]{a} \{x \mapsto v\}e'$ for any v .*

DEFINITION (SPECULATIONS) 2.3. *A speculation from an expression e to an expression e' is a (possibly empty) sequence $\sigma = (e_i \xrightarrow[o_i]{a_i} e_{i+1})_{0 \leq i \leq n}$ of speculation steps such that $e_0 = e$ and $e_n = e'$. This is written $\sigma : e \xrightarrow{*} e'$. The empty speculation (with $e' = e$) is denoted ε . The sequence σ is normal iff for all i the occurrence o_i is normal. The concatenation $\sigma \cdot \sigma' : e \xrightarrow{*} e'$ of σ and σ' is only defined (in the obvious way) if σ ends on the expression e'' where σ' originates.*

Notice that a normal speculation proceeds in program order, evaluating redexes inside evaluation contexts – not speculation contexts; still it may involve guessing some values that have to be read from the memory. Let us see two examples of speculations – omitting some labels, just mentioning the actions:

EXAMPLE 2.4.

$$\begin{aligned}
r := !p; q := tt & \xrightarrow{\text{wr}_{q,tt}} r := !p; () \\
& \xrightarrow{\text{rd}_{p,tt}} r := tt; () \\
& \xrightarrow{\text{wr}_{r,tt}} (); () \xrightarrow{\beta} ()
\end{aligned}$$

Here we speculate in two ways: first, the assignment $q := tt$, which would normally take place after reading p and updating r , is performed, or rather, issued, out of order; second, we guess a value read from memory location p .

EXAMPLE 2.5.

$$\begin{aligned}
(\text{if } !p \text{ then } q := tt \text{ else } ()) & \xrightarrow{\text{wr}_{q,tt}} (\text{if } !p \text{ then } () \text{ else } ()) \\
& \xrightarrow{\text{rd}_{p,tt}} (\text{if } tt \text{ then } () \text{ else } ()) \\
& \xrightarrow{\checkmark} ()
\end{aligned}$$

from Example 2.4. Similarly, we can commute two steps such as

$$\begin{array}{ccc} (\text{if } tt \text{ then } q := tt \text{ else } ()) & \xrightarrow{\text{wr}_{q,tt}} & (\text{if } tt \text{ then } () \text{ else } ()) \\ & \swarrow & \\ & & () \end{array}$$

(see Example 2.5), although in this case we first need to say that the first step in this sequence is indeed “the same” as the second one in

$$\begin{array}{ccc} (\text{if } tt \text{ then } q := tt \text{ else } ()) & \swarrow & q := tt \\ & \xrightarrow{\text{wr}_{q,tt}} & () \end{array}$$

To this end, given a speculation step $e \xrightarrow{a} e'$ and an occurrence o' in e , we define the *residual* of o' after this step, that is the occurrence, if any, that points to the same subterm (if any) as o' pointed to in e . For instance, if the step is

$$(\text{if } tt \text{ then } e_0 \text{ else } e_1) \xrightarrow{\varepsilon} e_0$$

then for $o' = \varepsilon$ or $o' = (\text{if } [] \text{ then } _ \text{ else } _)$ there is not residual, because the occurrence has been consumed in reducing the expression. The residual of any occurrence pointing into e_0 , i.e. of the form $(\text{if } _ \text{ then } [] \text{ else } _) \cdot o'$, is o' , whereas an occurrence of the form $(\text{if } _ \text{ then } _ \text{ else } []) \cdot o'$, pointing into e_1 , has no residual, since the subexpression e_1 is discarded by reducing to the first branch of the conditional expression. The notion of a residual here is much simpler than in the λ -calculus (see [24]), because an occurrence is never duplicated, since we do not compute inside a value (except in a function applied to an argument). Here the residual of an occurrence after a speculation step will be either undefined, whenever it is discarded by a conditional branching, or a single occurrence. We actually only need to know the action a that is performed and the occurrence o where it is performed in order to define the residual of o' after such a step. We therefore define $o'/(a, o)$ as follows:

$$o'/(a, o) = \begin{cases} o' & \text{if } o \not\leq o' \\ o \cdot o'' & \text{if } o' = o \cdot (\lambda x [] _) \cdot o'' \ \& \ a = \beta \\ & \text{or } o' = o \cdot (\text{if } _ \text{ then } [] \text{ else } _) \cdot o'' \ \& \ a = \surd \\ & \text{or } o' = o \cdot (\text{if } _ \text{ then } _ \text{ else } []) \cdot o'' \ \& \ a = \searrow \\ \text{undefined} & \text{otherwise} \end{cases}$$

In the following we write $o'/(a, o) \equiv o''$ to mean that the residual of o' after (a, o) is defined, and is o'' . Notice that if $o'/(a, o) \equiv o''$ with $o' \in \mathcal{O}cc^*$ then $o'' = o'$ and $o \not\leq o'$.

Speculation enjoys a partial confluence property, namely that if an occurrence of an action has a residual after another one, then one can still perform the action from the resulting expression. This property is known as the Diamond Lemma.

LEMMA (DIAMOND LEMMA) 3.1. *If $e \xrightarrow{o_0} e_0$ and $e \xrightarrow{o_1} e_1$ with $o_1/(a_0, o_0) \equiv o'_1$ and $o_0/(a_1, o_1) \equiv o'_0$ then there exists e' such that $e_0 \xrightarrow{o'_1} e'$ and $e_1 \xrightarrow{o'_0} e'$.*

PROOF SKETCH: by case on the respective positions of o_0 and o_1 . If neither $o_0 \leq o_1$ nor $o_1 \leq o_0$ (that is, the two occurrences are disjoint) then $o_1/(a_0, o_0) \equiv o_1$ and $o_0/(a_1, o_1) \equiv o_0$, and it is easy to see that the two reductions can be done in any order.

Otherwise, let us assume for instance that $o_0 < o_1$ (we cannot have $o_0 = o_1$ since $o_0/(a_1, o_1)$ is defined, and the case where $o_1 < o_0$ is symmetric). A case analysis shows that we have either $o_1 = o_0 \cdot (\lambda x [] _) \cdot o'_1$ and $a_0 = \beta$, or $o_1 = o_0 \cdot (\text{if } _ \text{ then } [] \text{ else } _) \cdot o'_1$ and $a_0 = \surd$, or else $o_1 = o_0 \cdot (\text{if } _ \text{ then } _ \text{ else } []) \cdot o'_1$ and $a_0 = \searrow$, with $o'_1 = o_0 \cdot o'_1$ in all these cases.

In the first case we have $e = \Sigma_0[(\lambda x e''v)]$ with $e_0 = \Sigma_0[\{x \mapsto v\}e'']$ (and $@\Sigma_0 = o_0$), and $e = \Sigma_0[(\lambda x e''v)]$ with $e'' \xrightarrow{o'_1} e'_1$ and $e_1 = \Sigma_0[(\lambda x e'_1v)]$, and we conclude using Lemma

2.2, with $e' = \Sigma_0[\{x \mapsto v\}e'_1]$. In the case where $a_0 = \surd$ we have $e = \Sigma_0[(\text{if } tt \text{ then } e'_0 \text{ else } e'_1)]$ with $e_0 = \Sigma_0[e'_0]$, and $e'_0 \xrightarrow{o'_1} e'_1$, and we conclude with $e' = \Sigma_0[e'_1]$. The case where $a_0 = \searrow$ is similar. \square

One should notice that the e' , the existence of which is asserted in this lemma, is actually unique, up to α -conversion. Let us see an example: with the expression of Example 2.4, we have – recall that $e_0; e_1$ stands for $(\lambda x e_1 e_0)$ where x is not free in e_1 :

$$r := !p; q := tt \xrightarrow{\text{wr}_{q,tt}} r := !p; ()$$

and

$$r := !p; q := tt \xrightarrow{(\bullet, []): (r := [])} r := tt; q := tt$$

Then we can close the diagram, ending up with the expression $r := tt; ()$. This confluence property is the basis for the definition of the equivalence by permutation of computing steps: with the hypotheses of the Diamond Lemma, we shall regard the two speculations

$$e \xrightarrow{o_0} e_0 \xrightarrow{o_1} e' \quad \text{and} \quad e \xrightarrow{o_1} e_1 \xrightarrow{o_0} e'$$

as equivalent. However, this cannot be so simple, because we have to ensure that the program order is preserved as regards accesses to a given memory location (unless these accesses are all reads). For instance, the speculation – again, omitting the occurrences:

$$p := tt; r := !p \xrightarrow{\text{rd}_{p,ff}} p := tt; r := ff \xrightarrow{\text{wr}_{p,tt}} () ; r := ff$$

should not be considered as valid, because it breaks the data dependency between the write and the read on p . To take this into account, we introduce the *conflict* relation between actions, as follows²:

DEFINITION (CONFLICTING ACTIONS) 3.2. *The conflict relation $\#$ between actions is given by*

$$\# = \bigcup_{p \in \mathcal{R}ef, v, w \in \mathcal{V}al} \{(\text{wr}_{p,v}, \text{wr}_{p,w}), (\text{wr}_{p,v}, \text{rd}_{p,w}), (\text{rd}_{p,v}, \text{wr}_{p,w})\}$$

We can now define the permutation equivalence, which is the congruence (with respect to concatenation) on speculations generated by the conflict-free Diamond property.

DEFINITION (PERMUTATION EQUIVALENCE) 3.3. *The equivalence by permutation of transitions is the least equivalence \simeq on speculations such that if $e \xrightarrow{o_0} e_0$ and $e \xrightarrow{o_1} e_1$ with $o_1/(a_0, o_0) \equiv o'_1$ and $o_0/(a_1, o_1) \equiv o'_0$ and $\neg(a_0 \# a_1)$ then*

$$\sigma_0 \cdot e \xrightarrow{o_0} e_0 \xrightarrow{o_1} e' \cdot \sigma_1 \simeq \sigma_0 \cdot e \xrightarrow{o_1} e_1 \xrightarrow{o_0} e' \cdot \sigma_1$$

where e' is determined as in the Diamond Lemma.

Notice that two equivalent speculations have the same length. Let us see some examples. The speculation given in Example 2.4 is equivalent to the normal speculation

$$\begin{array}{ccc} r := !p; q := tt & \xrightarrow{\text{rd}_{p,tt}} & r := tt; q := tt \\ & \xrightarrow{\text{wr}_{r,tt}} & () ; q := tt \xrightarrow{\beta} q := tt \\ & \xrightarrow{\text{wr}_{q,tt}} & () \end{array}$$

²We notice that in some (extremely, or even excessively) relaxed memory model (such as the one of the Alpha architecture, see [28]) the data dependencies are not maintained. To deal with such models, we would adopt an empty conflict relation, and a different notion of data race free program (see below).

Similarly, the speculation given in Example 2.5 is equivalent to the normal speculation

$$\begin{aligned} (\text{if } !p \text{ then } q := tt \text{ else } ()) &\xrightarrow{rd_{p,tt}} (\text{if } tt \text{ then } q := tt \text{ else } ()) \\ &\swarrow \\ & q := tt \xrightarrow{wr_{q,tt}} () \end{aligned}$$

We are now ready to give the definition that is central to our work, characterizing what is a *valid* speculative computation.

DEFINITION (VALID SPECULATIVE COMPUTATION) 3.4. *A speculation is valid if it is equivalent by permutation to a normal speculation. A speculative computation γ is valid if all its thread projections $\gamma|_t$ are valid speculations.*

It is clear for instance that the speculations given above that do not preserve the normal data dependencies are not valid. Similarly, one can see that the following speculation

$$\begin{aligned} (\text{if } !p \text{ then } () \text{ else } q := tt) &\xrightarrow{wr_{q,tt}} (\text{if } !p \text{ then } () \text{ else } ()) \\ &\xrightarrow{rd_{p,tt}} (\text{if } tt \text{ then } () \text{ else } ()) \\ &\swarrow \\ & () \end{aligned}$$

which is an example of wrong branch prediction, is invalid, since the occurrence of the first action has no residual after the last one, and cannot therefore be permuted with it. We have already seen that the speculations from Examples 2.4 and 2.5 are valid. Then the reader can observe that from the thread system – where we omit the thread identifiers

$$r := !p; q := tt \parallel r' := !q; p := tt$$

and an initial memory S such that $S(p) = ff = S(q)$, we can, by a valid speculative computation, get as an outcome a state where the memory S' is such that $S'(r) = tt = S'(r')$, something that cannot be obtained with the standard, non-speculative interleaving semantics. This is typical of a memory model where the reads can be reordered with respect to subsequent memory operations – a property symbolically called $\mathbf{R} \rightarrow \mathbf{RW}$, according to the terminology of [2], that was not captured in our previous work [10] on write-buffering memory models. We conjecture that our operational model of speculative computations is more general (for static thread systems) than the weak memory model of [10], in the sense that for any configuration, there are more outcomes following (valid) speculative computations than with write buffering. We also believe, although this would have to be more formally stated, that speculative computations are more general than most hardware memory models, which deal with access memory, but do not transform programs using some semantical reasoning as optimizing compilers do. For instance, let us examine the case of the amd64 example (see [32]), that is

$$p := tt \parallel q := tt \parallel \begin{array}{ll} r_0 := !p; & r_2 := !q; \\ r_1 := !q & r_3 := !p \end{array}$$

If we start from a configuration where the memory S is such that $S(p) = ff = S(q)$, we may speculate in the third thread that $!q$ returns ff (which is indeed the initial value of q), and similarly in the fourth thread that $!p$ returns ff , and then proceed with the assignments $p := tt$ and $q := tt$, and so on. Then we can reach, by a valid speculative computation, a state where the memory S' is such that $S'(r_0) = tt = S'(r_2)$ and $S'(r_1) = ff = S'(r_3)$, an outcome which cannot be obtained with the interleaving semantics.

Another unusual example is based on Example 2.5. Let us consider the following system made of two threads

$$\begin{array}{ll} p := ff; & \parallel q := ff; \\ (\text{if } !p \text{ then } q := tt \text{ else } ()) & (\text{if } !q \text{ then } p := tt \text{ else } ()) \end{array}$$

Then by a valid speculative computation we can reach, after having performed the two initial assignments, a state where $S(p) = tt = S(q)$. What is unusual with this example, with respect to what is generally expected from relaxed memory models for instance [3, 17], is that this is, with respect to the interleaving semantics, a data race free thread system, which still has an “unwanted” outcome in the optimizing framework of speculative computations (see [7] for a similar example). This indicates that we have to assume a stronger property than DRF (data-race freeness) to ensure that a program is “robust” with respect to speculations.

DEFINITION (ROBUST PROGRAMS) 3.5. *A closed expression e is robust iff for any t and γ such that $\gamma : (\emptyset, \emptyset, (t, e)) \xrightarrow{*} (S, L, T)$ there exists a normal computation $\bar{\gamma}$ such that $\bar{\gamma} : (\emptyset, \emptyset, (t, e)) \xrightarrow{*} (S, L, T)$.*

In other words, for a robust expression the speculative and interleaving semantics coincide, or: the robust programs are the ones for which the speculative semantics is correct (with respect to the interleaving semantics).

4. Main Result

Our main result is that *speculatively* data-race free programs are robust.

DEFINITION (SPECULATIVELY DRF PROGRAM) 4.1. *A configuration C is speculatively data race free (speculatively DRF) iff for any configuration C' reachable from C , that is $\gamma : C \xrightarrow{*} C'$ for some speculative computation γ , if $C' \xrightarrow[t_0, \sigma_0]{a_0} C_0$ and $C' \xrightarrow[t_1, \sigma_1]{a_1} C_1$*

we have $t_0 \neq t_1 \Rightarrow \neg(a_0 \# a_1)$. An expression e is speculatively DRF iff for any t the configuration $(\emptyset, \emptyset, (t, e))$ is speculatively DRF.

It is obvious that this is a safety property, that is if C is speculatively DRF and $C \xrightarrow{*} C'$, then C' is speculatively DRF. We could have formulated this property directly, without resorting to the conflict relation, saying that there are no reachable concurrent accesses to the same location in the memory. In this way we could deal with optimizing architectures (such as the Alpha memory model, see [28]) that allow to reorder such accesses, by including the case where these concurrent accesses can occur (in the speculative semantics) from within the same thread, like for instance in $p := ff; r := !p$. We do not follow this way here, since such a model requires unnatural synchronizations from the programmer.

In order to establish our result, we need a number of preliminary lemmas, regarding both speculations and speculative computations.

4.1 Properties of Speculations

We extend the notion of residual by defining o/σ where o is an occurrence and σ a speculation. This is defined by induction on the length of σ , where the notation $o' \equiv o/\sigma$ means that o/σ is defined and is o' .

$$\begin{aligned} o/\varepsilon &\equiv o \\ o/(e \xrightarrow[o']{a} e') \cdot \sigma &\equiv (o/(a, o'))/\sigma \end{aligned}$$

The following lemma states that the residual of a given occurrence along equivalent speculations are the same. This property was called the “cube lemma” in [8].

LEMMA (THE CUBE LEMMA) 4.2. *Let $\sigma = e \xrightarrow[\sigma_0]{a_0} \cdot \xrightarrow[\sigma'_1]{a_1} e'$ and $\sigma' = e \xrightarrow[\sigma'_1]{a_1} \cdot \xrightarrow[\sigma_0]{a_0} e'$ be such that $\sigma \simeq \sigma'$. Then $o/\sigma \equiv o/\sigma'$ for any o .*

PROOF: straightforward (but tedious) case analysis. \square

In the following we shall often omit the expressions in a speculation, writing $\sigma_0 \cdot \xrightarrow[o]{a} \cdot \sigma_1$ instead of $\sigma_0 \cdot (e_0 \xrightarrow[o]{a} e_1) \cdot \sigma_1$. Indeed, e_0

is determined by σ_0 , and, given e_0 , the expression e_1 is determined by the pair (a, o) . Now we introduce the notion of a *step*, called “redex-with-history” in [6, 24], and of steps being in the same *family*, a property introduced in [6].

DEFINITION (STEPS) 4.3. A step is a pair $[\sigma, (a, o)]$ of a speculation $\sigma : e \xrightarrow{*} e'$ and an action a at occurrence o such that $e' \xrightarrow{a/o} e''$ for some expression e'' . Given a speculation σ , the set $\text{Step}(\sigma)$ is the set of steps $[\varsigma, (a, o)]$ such that $\varsigma \cdot \frac{a}{o} \leq \sigma$. The binary relation \sim on steps, meaning that two steps are in the same family, is the equivalence relation generated by the rule

$$\frac{\exists \sigma'' . \sigma' \simeq \sigma \cdot \sigma'' \ \& \ o' \equiv o/\sigma''}{[\sigma, (a, o)] \sim [\sigma', (a, o')]}$$

Equivalent speculations have similar steps:

LEMMA 4.4. If $[\varsigma, (a, o)] \in \text{Step}(\sigma)$ and $\sigma' \simeq \sigma$ then there exists $[\varsigma', (a, o')] \in \text{Step}(\sigma')$ such that $[\varsigma, (a, o)] \sim [\varsigma', (a, o')]$.

PROOF: by induction on the definition of $\sigma' \simeq \sigma$. \square

A property that should be intuitively clear is that if a step in a speculation is in the same family as the initial step of an equivalent speculation, then it can be commuted with all the steps that precede it:

LEMMA 4.5. Let $\sigma = \sigma_0 \cdot \frac{a}{o} \cdot \sigma_1$ be such that $\sigma \simeq \frac{a}{o'} \cdot \varsigma$ with $[\varepsilon, (a, o')] \sim [\sigma_0, (a, o)]$. If $\sigma_0 = \varsigma_0 \cdot (e \xrightarrow{\bar{a}/\bar{o}} e') \cdot \varsigma_1$ then there exist o'', e'', \bar{o}' and σ'_1 such that $\varsigma_0 \cdot (e \xrightarrow{a/o''} e'' \xrightarrow{\bar{a}/\bar{o}'} \bar{e}) \cdot \sigma'_1 \simeq \sigma$ where $o \equiv o''/\frac{\bar{a}}{\bar{o}'}$ and $\bar{o}' \equiv \bar{o}/(a, o'')$.

PROOF SKETCH: by induction on the inference of $\sigma \simeq \frac{a}{o'} \cdot \varsigma$.

This is trivial if $\sigma = \frac{a}{o'} \cdot \varsigma$, that is $\sigma_0 = \varepsilon$, $o' = o$ and $\varsigma = \sigma_1$. Otherwise, there exists σ' such that $\sigma' \simeq \frac{a}{o'} \cdot \varsigma$ (with a shorter inference) and σ' results from σ by a transposition of two consecutive steps. If this transposition occurs in σ_1 , or commutes the last step of $\sigma_0 \cdot \frac{a}{o}$ with the first step of σ_1 , we simply use the induction hypothesis. If the transposition commutes the last step of σ_0 with the last one of $\sigma_0 \cdot \frac{a}{o}$, we get the lemma in the case where $\varsigma_1 = \varepsilon$, and we use the induction hypothesis for the other cases. If the transposition commutes two steps in σ_0 , we use the induction hypothesis and the Cube Lemma 4.2 to conclude. \square

LEMMA 4.6. If $e \xrightarrow{o_0} e_0$ and $e \xrightarrow{o_1} e_1$ with $o_1/(a_0, o_0) \equiv o'_1$ and $o_0/(a_1, o_1) \equiv o'_0$ then $o_0 \in \mathcal{O}cc^*$ implies $o'_0 = o_0$, $o_1 \not\leq o_0$ and $o_1 \notin \mathcal{O}cc^*$. Moreover if $a_0 \in \mathcal{B}$ then $o_0 \not\leq o_1$, hence $o'_1 = o_1$.

PROOF: we already observed that, with the hypothesis of this lemma, we have $o'_0 = o_0$ and $o_1 \not\leq o_0$. It cannot be the case that $o_1 \in \mathcal{O}cc^*$, since otherwise we would have $o_0 = o_1$ by Lemma 2.1. If $a_0 = \mu$ or $a_0 = \hat{\ell}$ then $e = \Sigma[(\text{with } \ell \text{ do } e')]$ (with $o_0 = @\Sigma$), and therefore $o_0 \not\leq o_1$ since there is no frame of the form $(\text{with } \ell \text{ do } \square)$. The argument is similar with $a_0 = \text{spw}_e$. If $a_0 = \hat{\ell}$ then $e = \mathbf{E}[(\text{holding } \ell \text{ do } v)]$ and again $o_0 \not\leq o_1$ since there is no way to compute inside a value (except in the function part of an application). \square

This allows us to show that, in a speculation, the unlock actions, and also spawning a new thread, act as *barriers* with respect to other actions that occur in an evaluation context: these actions cannot be permuted with unlock (or spawn) actions. This is expressed by the following lemma:

LEMMA 4.7. Let $\sigma = \sigma_0 \cdot \frac{a}{o} \cdot \sigma_1$ where $a \in \mathcal{B}$, and $\sigma \simeq \sigma'$ with $\sigma' = \sigma'_0 \cdot \frac{a}{o'} \cdot \sigma'_1$ where $[\sigma_0, (a, o)] \sim [\sigma'_0, (a, o)]$. If $[\varsigma, (a', o')] \in \text{Step}(\sigma_0)$ with $o' \in \mathcal{O}cc^*$ then there exist ς' and o'' such that $[\varsigma', (a', o'')] \in \text{Step}(\sigma'_0)$ and $[\varsigma, (a', o')] \sim [\varsigma', (a', o'')]$.

PROOF: by induction on the definition of $\sigma \simeq \sigma'$. The lemma is obvious if $\sigma' = \sigma$. Otherwise, there exists σ'' such that σ'' results from σ by a transposition of two steps, and $\sigma'' \simeq \sigma'$ (with a shorter proof). There are four cases: if the transposition takes place in σ_0 or σ_1 , the proof is routine (using the induction hypothesis). Otherwise, the transposition concerns the step $[\sigma_0, (a, o)]$, and either the last step of σ_0 , or the first step of σ_1 .

If $\sigma_0 = \sigma'_0 \cdot (e \xrightarrow{a'/o'} e_0)$ and $e \xrightarrow{a/o} \bar{e}$ with $o \equiv \bar{o}/(a', o')$, $o'' \equiv o'/(a, \bar{o})$ and

$$\sigma'' = \sigma'_0 \cdot (e \xrightarrow{a/o} \bar{e}) \cdot (\bar{e} \xrightarrow{a'/o''} e_1) \cdot \sigma_1$$

then $\bar{o} \in \mathcal{O}cc^*$ by Lemma 4.6, since $a \in \mathcal{B}$, and therefore $o' \notin \mathcal{O}cc^*$ and $o'' = o'$. Then we easily conclude using the induction hypothesis.

The case where $\sigma_1 = (e_1 \xrightarrow{a'/o'} e) \cdot \sigma'_1$ and $e_0 \xrightarrow{a/o} \bar{e}$ with $o' \equiv o''/(a, o)$, $\bar{o} \equiv o/(a', o'')$ and

$$\sigma'' = \sigma_0 \cdot (e_0 \xrightarrow{a'/o''} \bar{e}) \cdot (\bar{e} \xrightarrow{a/o} e) \cdot \sigma'_1$$

is similar (notice that it cannot be the case that $o'' \in \mathcal{O}cc^*$). \square

An immediate consequence of this property is:

COROLLARY 4.8. If σ is a valid speculation, that is $\sigma \simeq \bar{\sigma}$ for some normal speculation $\bar{\sigma}$, and if $\bar{\sigma} = \bar{\sigma}_0 \cdot \frac{a}{o} \cdot \bar{\sigma}_1$ with $a \in \mathcal{B}$, then $\sigma = \sigma_0 \cdot \frac{a}{o} \cdot \sigma_1$ with $[\sigma_0, (a, o)] \sim [\bar{\sigma}_0, (a, o)]$, such that for any step $[\bar{\varsigma}, (a', o')]$ of $\bar{\sigma}_0$ there exists a step $[\varsigma, (a', o'')]$ in the same family which is in σ_0 .

This is to say that, in order for a speculation to be valid, all the operations that normally precede a \mathcal{B} action, and in particular an unlocking action, must be performed before this action in the speculation.

4.2 Properties of Speculative Computations

From now on, we shall consider *regular* configurations, where at most one thread can hold a lock, and where a lock held by some thread is indeed in the lock context. This is defined as follows:

DEFINITION (REGULAR CONFIGURATION) 4.9. A configuration $C = (S, L, T)$ is regular if and only if it satisfies

- (i) if $T = (t_i, \Sigma_i[(\text{holding } \ell \text{ do } e_i)]) \parallel T_i$ for $i = 0, 1$ then $t_0 = t_1$ & $\Sigma_0 = \Sigma_1$ & $e_0 = e_1$ & $T_0 = T_1$
- (ii) $T = (t, \Sigma[(\text{holding } \ell \text{ do } e)]) \parallel T' \Rightarrow \ell \in L$

For instance, any configuration of the form $(\emptyset, \emptyset, (t, e))$ where e is an expression is regular. The following should be obvious:

REMARK 4.10. If C is regular and $C \xrightarrow{a/o} C'$ then C' is regular.

The following lemma (for a different notion of computation) was called the “Asynchrony Lemma” in [10]. There it was used as the basis to define the equivalence by permutation of computations. We could also introduce such an equivalence here, generalizing the one for speculations, but this is actually not necessary.

LEMMA 4.11. Let C be a (well-formed) regular configuration. If $C \xrightarrow{a_0/t_0, o_0} C_0 \xrightarrow{a_1/t_1, o_1} C'$ with $t_0 \neq t_1$, $\neg(a_0 \# a_1)$ and $a_0 = \hat{\ell} \Rightarrow a_1 \neq \hat{\ell}$, then there exists C_1 such that $C \xrightarrow{a_1/t_1, o_1} C_1 \xrightarrow{a_0/t_0, o_0} C'$.

PROOF (SKETCH): by a case analysis on the actions a_0 and a_1 .

- If $a_0 \in \{\beta, \sphericalangle, \searrow\}$ then this action has no side effect (i.e., it does not modify the components S, L and $\square \parallel T$ of the configuration), and therefore such an action commutes with any other a_1 .
- The case where $a_0 = \nu_\ell$ is similar; one may observe that it can be commuted with $a_1 = \nu_{\ell'}$, since in this case we have $\ell' \neq \ell$.
- If $a_0 = \nu_{p,v}$ then it cannot be the case that a_1 is $\text{rd}_{p,w}$ or $\text{wr}_{p,w}$, by the well-formedness of the configurations. Also, $a_1 \neq \nu_{p,w}$, and it is therefore easy to see that a_1 commutes with a_0 in this case.
- If $a_0 = \text{rd}_{p,v}$ then we have $a_1 \neq \text{wr}_{p,w}$ (otherwise $a_0 \# a_1$) and $a_1 \neq \nu_{p,w}$ by well-formedness. Again it is easy to see that in any possible case for a_1 , the two actions commute, producing the same resulting configuration.
- If $a_0 = \text{wr}_{p,v}$ then we have $a_1 \neq \text{rd}_{p,w}$, $a_1 \neq \text{wr}_{p,w}$ and $a_1 \neq \nu_{p,w}$. As in the previous case, we easily conclude.
- If $a_0 = \widehat{\ell}$ or $a = \mu$ then $a_1 \neq \widehat{\ell}$ since two different threads cannot acquire the same lock (we are using the regularity of C when $a_0 = \mu$). Also, $a_1 \neq \widehat{\ell}$ because this would mean that C_0 is not regular. If $a_0 = \widehat{\ell}$ then $a_1 \neq \widehat{\ell}$, since otherwise C_0 would not be regular, and $a_1 \neq \widehat{\ell}$ by hypothesis. Again in these cases it is easy to conclude that the lemma holds.
- The case where $a_0 = \text{spw}_e$ is immediate. \square

We have a similar property regarding “local” computations, that occur in the same thread:

LEMMA 4.12. *Let C be a (well-formed) regular configuration.*

If $C \xrightarrow[t, o_0]{a_0} C_0 \xrightarrow[t, o'_1]{a_1} C'$ with $C = (S, L, (t, e) \parallel T)$, $C_0 = (S_0, L_0, (t, e_0) \parallel T_0)$, $C' = (S', L', (t, e') \parallel T')$ and

$$e \xrightarrow[o_0]{a_0} e_0 \xrightarrow[o'_1]{a_1} e' \simeq e \xrightarrow[o_1]{a_1} e_1 \xrightarrow[o'_0]{a_0} e'$$

then $C \xrightarrow[t, o_1]{a_1} (S_1, L_1, (t, e_1) \parallel T_1) \xrightarrow[t, o'_0]{a_0} C'$ for some S_1, L_1 and T_1 .

PROOF SKETCH: we distinguish three cases, according to the respective position of the occurrences o_0 and o_1 .

- $o_0 \leq o_1$. In this case, we can only have $a_0 \in \{\beta, \sphericalangle, \searrow\}$, and therefore $S_0 = S$, $L_0 = L$ and $T_0 = T$, and it is easy to conclude with $S_1 = S'$, $L_1 = L'$ and $T_1 = T'$.
- $o_1 < o_0$. Then $a_1 \in \{\beta, \sphericalangle, \searrow\}$, hence $S' = S_0$, $L' = L_0$ and $T' = T_0$, and we conclude, as in the previous case, with $S_1 = S$, $L_1 = L$ and $T_1 = T$.

- If o_0 and o_1 are disjoint, that is $o_0 \not\leq o_1$ and $o_1 \not\leq o_0$, we proceed by case on (a_0, o_0) and (a_1, o_1) , as in the previous proof. Notice that $\{a_0, a_1\} \subseteq \mathcal{B}$ is impossible, by Lemma 2.1, since these actions must occur in an evaluation context. \square

We can now prove our main result, stating in particular that speculatively race free programs are robust against speculations:

THEOREM (MAIN RESULT) 4.13. *Let C be a well-formed regular configuration that is speculatively race free. If $\gamma : C \xrightarrow{*} C'$ is a valid speculative computation, then there exists a normal computation $\bar{\gamma}$ from C to C' . In particular, any speculatively data race free closed expression is robust.*

PROOF: by induction on the length of γ . This is trivial if $\gamma = \varepsilon$. Otherwise, let $\gamma = (C_i \xrightarrow[t_i, o_i]{a_i} C_{i+1})_{0 \leq i \leq n}$ with $n > 0$. Notice that C_i is well-formed, regular and speculatively DRF for any i . The set $\{t \mid \gamma|_t \neq \varepsilon\}$ is non-empty. For any t there exists a normal speculation σ^t such that $\sigma^t \simeq \gamma|_t$. Let j be the first index

($0 \leq j < n$) such that $\gamma|_{t_j} = \sigma_0 \cdot \xrightarrow[o_j]{a_j} \cdot \sigma_1$ and $\sigma^{t_j} = \xrightarrow{o}{a_j} \cdot \sigma'$ with $[\varepsilon, (o, a_j)] \sim [\sigma_0, (a_j, o_j)]$. Now we proceed by induction on j . If $j = 0$ then $o = o_j \in \mathcal{O}cc^*$, and we use the induction hypothesis (on the length n) to conclude. Otherwise, we have $C_{j-1} \xrightarrow[t_{j-1}, o_{j-1}]{a_{j-1}} C_j \xrightarrow[t_j, o_j]{a_j} C_{j+1}$. We distinguish two cases.

- If $t_{j-1} \neq t_j$ then we have $\neg(a_{j-1} \# a_j)$ since C is speculatively data-race free. We show that $i < j \Rightarrow a_i \notin \mathcal{B}$. Assume the contrary, that is $a_i \in \mathcal{B}$ for some $i < j$. Then $\gamma|_{t_i} = \zeta_0 \cdot \xrightarrow[o_i]{a_i} \cdot \zeta_1$, and by Lemma 4.4 we have $\sigma^{t_i} = \bar{\zeta}_0 \cdot \xrightarrow{o'}{a_i} \cdot \bar{\zeta}_1$ with $[\zeta_0, (o_i, a_i)] \sim [\bar{\zeta}_0, (o', a_i)]$. Then by Corollary 4.8 the first step of $\bar{\zeta}_0 \cdot \xrightarrow{o'}{a_i}$ is in the family of a step in $\zeta_0 \cdot \xrightarrow[o_i]{a_i}$, contradicting the minimality of j .

We therefore have $a_{j-1} \neq \widehat{\ell}$ in particular. By Lemma 4.11 we can commute the two steps $\xrightarrow[o_{j-1}]{a_{j-1}}$ and $\xrightarrow[o_j]{a_j}$, and we conclude using the induction hypothesis (on j).

- If $t_{j-1} = t_j$, we have $\sigma_0 = \zeta_0 \cdot \xrightarrow[o_{j-1}]{a_{j-1}}$, and by Lemma 4.5

there exist o', o'' and σ'_1 such that $\gamma|_{t_j} \simeq \zeta_0 \cdot \xrightarrow{o'}{a_j} \cdot \xrightarrow{o''}{a_{j-1}} \cdot \sigma'_1$ with $o \equiv o'/(a_{j-1}, o_{j-1})$. We conclude using Lemma 4.12 and the induction hypothesis (on j). \square

Notice that we proved a property that is actually more precise than stated in the theorem, since the $\bar{\gamma}$ that is constructed is equivalent, by permutations, to γ – but we decided not to introduce explicitly this equivalence as regards speculative computations. We also observe that if an expression is purely sequential, that is, it does not spawn any thread, then it is speculatively data race free, and therefore robust, that is, all the valid speculations for it are correct with respect to its standard semantics.

Our result holds with synchronization mechanisms other than acquiring and releasing locks. We shall use the mutual exclusion construct (with ℓ do e) in the sequel, but we could have considered simpler memory barrier operations, such as fence. This is a programming constant (but not a value), the semantics of which is given by

$$\mathbf{E}[\text{fence}] \rightarrow \mathbf{E}\{0\}$$

with no side effect. Performing a fence should be categorized as a \mathcal{B} action, so that the Corollary 4.8 holds for such an action, since it is only performed from within a normal evaluation context. Then our Theorem 4.13, which, as far as the \mathcal{B} actions are concerned, relies on this property 4.8, still holds with this construct. However when speculation is allowed this construct is rather weak, and in particular it does not help very much in preventing data races, or even to separate the accesses to the memory from a given thread. We let the reader check for instance that with the IRIW example (see [7]), that is

$$\begin{array}{llll} p := tt & \parallel & q := tt & \parallel & r_0 := !p; & \parallel & r_2 := !q; \\ & & & & \text{fence;} & & \text{fence;} \\ & & & & r_1 := !q & & r_3 := !p \end{array}$$

starting from a configuration where the memory S is such that $S(p) = \text{ff} = S(q)$ we may, as with the amd64 example above, get by a valid speculative computation a state where the memory S' is such that $S'(r_0) = tt = S'(r_2)$ and $S'(r_1) = \text{ff} = S'(r_3)$. This is because the assignments to r_1 and r_3 can be speculatively performed first (after having read pointers p and q), and, in the projections over their respective threads, be commuted with the assignments to r_0 and r_2 (since there is no data dependency), and

$(S, \Omega, (t, \mathbf{E}[(\omega xep)]) \parallel T) \rightarrow (S, \Omega, (t, \mathbf{E}[\{x \mapsto p\}e]) \parallel T)$	$p \in [\mathbf{E}]$
$(S, \Omega, (t, \mathbf{E}[(\omega xep)]) \parallel T) \rightarrow (S, \Omega \cup \{p \mapsto t\}, \{(t, \mathbf{E}[\{x \mapsto p\}e \setminus p])\} \parallel T)$	$p \notin [\mathbf{E}] \cup \text{dom}(\Omega)$
$(S, \Omega, (t, \mathbf{E}[(v \setminus p)]) \parallel T) \rightarrow (S, \Omega \setminus p, \{(t, \mathbf{E}[v])\} \parallel T)$	
$(S, \Omega, (t, \mathbf{E}[(\text{ref } v)]) \parallel T) \rightarrow (S \cup \{p \mapsto v\}, \Omega \cup \{p \mapsto t\}, (t, \mathbf{E}[p]) \cup T)$	$p \notin \text{dom}(S)$
$(S, \Omega, (t, \mathbf{E}[(\text{sref } v)]) \parallel T) \rightarrow (S \cup \{p \mapsto v\}, \Omega, (t, \mathbf{E}[p]) \parallel T)$	$p \notin \text{dom}(S)$
$(S, \Omega, (t, \mathbf{E}[(!p)]) \parallel T) \rightarrow (S, \Omega, (t, \mathbf{E}[v]) \parallel T)$	$(p \notin \text{dom}(\Omega) \text{ or } \Omega(p) = t) \text{ and } S(p) = v$
$(S, \Omega, (t, \mathbf{E}[(p := v)]) \parallel T) \rightarrow (S[p := v], \Omega, (t, \mathbf{E}[\emptyset]) \parallel T)$	$p \notin \text{dom}(\Omega) \text{ or } \Omega(p) = t$

Figure 3: Operational Semantics (Source Language)

the fence, thus checking that local normal order evaluations with the same actions is possible.

5. Towards Thread Safe Programming

In this section we introduce another language, that we will call the *source* language, intended to provide the programmer with a *safe* concurrent programming style, where the semantics is guaranteed to be sequentially consistent³. The differences with the (target) language we considered up to now are as follows. We split the reference creation construct into two: one, which we still denote $(\text{ref } e)$, for creating a *private* reference, that only one thread can use, and $(\text{sref } e)$ for creating a potentially shared reference. We may say that a thread *owns* the private references it creates. We also replace the locking construct (with ℓ do e) by a functional construct $\omega x e$, read “own x in e ” that, when applied to a (shared) pointer, acquires that pointer for exclusive use in e , and releases it upon termination. Then in this language synchronization is only concerned with acquiring/releasing pointers, not locks. The synchronization construct $\omega x e$ is needed to write atomic operations, such as incrementing an account a by some amount x , written $\omega a \lambda x (a := !a + x)$. The syntax is as follows

$e ::= v \mid (e_0 e_1)$	<i>expressions</i>
$\mid (\text{if } e \text{ then } e_0 \text{ else } e_1)$	
$\mid (\text{ref } e) \mid (\text{sref } e) \mid (!e) \mid (e_0 := e_1)$	
$\mid (\text{thread } e)$	
$v ::= x \mid \lambda x e \mid \omega x e \mid tt \mid ff \mid ()$	<i>values</i>

The variable x is bound in $\omega x e$, exactly as in $\lambda x e$.

To describe the semantics of this language, we have to introduce, as with the target language, an extension with constructs that appear at run-time. These are the pointers p, q, \dots , as above, and a construct $(e \setminus p)$, similar to (holding ℓ do e), meaning that the pointer p has been acquired, and is currently owned by e . The evaluation contexts in the source language, that we still denote by \mathbf{E} , are the same as in the target language, except that we add the frame $(\text{sref } \square)$, and that (holding ℓ do \square) is replaced by $(\square \setminus p)$. We reuse the notation $[\mathbf{E}]$, this time to mean the set of pointers that are owned in the context \mathbf{E} , that is the set of p such that $(\square \setminus p)$ occurs in \mathbf{E} . The semantics is specified as small steps transitions between configurations of the form (S, Ω, T) where S and T are respectively the memory and the thread system, as in Section 2 (but built with expressions of the source language), and Ω is a mapping from $\text{dom}(\Omega)$, a subset of $\text{dom}(S)$, into Names . This represents *ownership*: if the pointer p has been created using ref by a thread named t , we have $\Omega(p) = t$. A pointer created by means of sref is not owned by any thread, but it can be temporarily acquired as a private reference by a thread applying an $\omega x e$ function. The private references

can only be read or written by their owner thread. This is formalized in Figure 3, where, in order to save some space, we omit the cases of the redexes $(\lambda x e v)$, (if tt then e_0 else e_1), (if ff then e_0 else e_1) and (thread e). In the rule for reducing $(v \setminus p)$, $\Omega \setminus p$ means Ω restricted to $\text{dom}(\Omega) - \{p\}$.

One can see in Figure 3 that a thread is blocked when it tries to acquire, by means of (ωxep) , an exclusive access to a reference p that is currently private – i.e. $p \in \text{dom}(\Omega)$. This is a synchronization operation. By contrast, there is a run-time error when a thread t tries to access (i.e. read or write) a private reference p that it does not own, i.e. $p \in \text{dom}(\Omega)$, but the condition $\Omega(p) = t$ is not met in the corresponding rules. We shall design a type and effect system [26] to prevent such errors. In this system, *effects* are finite sets of threads names, that is, if we let φ, ψ, \dots range over effects, $\varphi \subseteq \text{Names}$. As we shall see, effects that can be assigned to expressions in the type system are either empty, or a singleton. The meaning is that a closed expression has an empty effect whenever the thread it represents does not access (read or write) any private reference, whereas the effect $\{t\}$ is assigned to a thread named t that uses its own private references. The types are given by

$$\tau, \sigma, \theta \dots ::= \text{unit} \mid \text{bool} \mid \theta \text{ref}_t \mid \theta \text{sref} \mid (\tau \xrightarrow{\varphi} \sigma)$$

where t is any thread name. As usual, a functional type $(\tau \xrightarrow{\varphi} \sigma)$ records the latent effect of a function of that type, that is the effect the function may have when applied to an argument. The type θref_t is the type of a reference which is created by a thread named t , and which contains values of type θ .

The judgements of the type and effect system for the source language have the form $\Gamma \vdash_t e : \varphi, \tau$ where, as usual, Γ is a typing context, that is a mapping from a finite set of variables to types. The index t means that e is typed as a part of a thread supposedly named t . This name is only used to ensure that a thread does not access private references that it does not create. We shall use the type system to define a translation $e \Rightarrow \bar{e}$ from the source language to the target language, and we therefore introduce judgements of the form $\Gamma \vdash_t e : \varphi, \tau \Rightarrow \bar{e}$ meaning that, in the context of Γ , the source expression, as part of a thread named t , has type τ and effect φ , and translates to the expression \bar{e} of the target language. To define this translation, we actually extend the target language with a record construct $\{\text{lock} = \ell, \text{val} = e\}$, together with the field selection operations $e.\text{lock}$ and $e.\text{val}$. For lack of space, we do not give the definition of the operational semantics (which is standard) for these constructs. The idea of the translation is to transform an expression of the source language of type θsref into a very simple monitor, namely a record where the val field is a reference and the lock field is a lock protecting the access to the reference. A private reference, that is an expression of type θref , is translated simply as a reference (hence the use of ref for private references), without any protection. The accesses to an sref are then translated into synchronized access, preventing any data race, whereas the

³The semantics may also be guaranteed to be *deadlock-free*, by using the “prudent semantics” of [9].

$\Gamma, x : \tau \vdash_t x : \emptyset, \tau \Rightarrow x$	$\Gamma \vdash_t tt : \emptyset, \text{bool} \Rightarrow tt$	$\Gamma \vdash_t ff : \emptyset, \text{bool} \Rightarrow ff$	$\Gamma \vdash_t () : \emptyset, \text{unit} \Rightarrow ()$
$\Gamma, x : \tau \vdash_t e : \varphi, \sigma \Rightarrow \bar{e}$	$\varphi \subseteq \psi$	$\Gamma, x : \theta \text{ref}_t \vdash_t e : \varphi', \tau \Rightarrow \bar{e}$	$\Gamma, x : \theta \text{sref}_t \vdash_t e : \varphi, \sigma \Rightarrow \bar{e}'$
$\Gamma \vdash_t \lambda x e : \emptyset, (\tau \xrightarrow{\psi} \sigma) \Rightarrow \lambda x \bar{e}$	$\Gamma \vdash_t \omega x e : \emptyset, (\theta \text{sref} \xrightarrow{\psi} \sigma) \Rightarrow \lambda y (\text{with } y.\text{lock do } (\lambda x \bar{e} y.\text{val}))$		
$\Gamma \vdash_t e_0 : \varphi_0, (\tau \xrightarrow{\varphi_2} \sigma) \Rightarrow \bar{e}_0$	$\Gamma \vdash_t e_1 : \varphi_1, \tau \Rightarrow \bar{e}_1$	$\Gamma \vdash_t e_0 : \varphi_0, \text{bool} \Rightarrow \bar{e}_0$	$\Gamma \vdash_t e_1 : \varphi_1, \tau \Rightarrow \bar{e}_1$
$\Gamma \vdash_t (e_0 e_1) : \varphi_0 \cup \varphi_1 \cup \varphi_2, \sigma \Rightarrow (\bar{e}_0 \bar{e}_1)$	$\Gamma \vdash_t (\text{if } e_0 \text{ then } e_1 \text{ else } e_2) : \varphi_0 \cup \varphi_1 \cup \varphi_2, \tau \Rightarrow (\text{if } \bar{e}_0 \text{ then } \bar{e}_1 \text{ else } \bar{e}_2)$		
$\Gamma \vdash_t e : \varphi, \theta \Rightarrow \bar{e}$	$\Gamma \vdash_t e : \varphi, \theta \Rightarrow \bar{e}$		
$\Gamma \vdash_t (\text{ref } e) : \varphi, \theta \text{ref}_t \Rightarrow (\text{ref } \bar{e})$	$\Gamma \vdash_t (\text{sref } e) : \varphi, \theta \text{sref} \Rightarrow (\text{new } \ell \text{ in } \{\text{lock} = \ell, \text{val} = (\text{ref } \bar{e})\})$		
$\Gamma \vdash_t e : \varphi, \theta \text{ref}_t \Rightarrow \bar{e}$	$\Gamma \vdash_t e : \varphi, \theta \text{sref} \Rightarrow \bar{e}$		
$\Gamma \vdash_t (!e) : \varphi \cup \{t\}, \theta \Rightarrow (!\bar{e})$	$\Gamma \vdash_t (!e) : \varphi, \theta \Rightarrow (\text{let } x = \bar{e} \text{ in } (\text{with } x.\text{lock do } !x.\text{val}))$		
$\Gamma \vdash_t e_0 : \varphi_0, \theta \text{ref}_t \Rightarrow \bar{e}_0$	$\Gamma \vdash_t e_1 : \varphi_1, \theta \Rightarrow \bar{e}_1$	$\Gamma \vdash_t e : \varphi, \text{unit} \Rightarrow \bar{e}$	$t \notin \Gamma, \varphi \subseteq \{t\}$
$\Gamma \vdash_t (e_0 := e_1) : \varphi_0 \cup \varphi_1 \cup \{t\}, \text{unit} \Rightarrow (\bar{e}_0 := \bar{e}_1)$	$\Gamma \vdash_{t'} (\text{thread } e) : \emptyset, \text{unit} \Rightarrow (\text{thread } \bar{e})$		
$\Gamma \vdash_t e_0 : \varphi_0, \theta \text{sref} \Rightarrow \bar{e}_0$	$\Gamma \vdash_t e_1 : \varphi_1, \theta \Rightarrow \bar{e}_1$		
$\Gamma \vdash_t (e_0 := e_1) : \varphi_0 \cup \varphi_1, \text{unit} \Rightarrow (\text{let } x = \bar{e}_0 \text{ in } (\text{let } y = \bar{e}_1 \text{ in } (\text{with } x.\text{lock do } x.\text{val} := y)))$			

Figure 4: Type-Directed Translation

accesses to private references are left unprotected, and therefore subject to speculations.

The rules for inferring the judgements $\Gamma \vdash_t e : \varphi, \tau \Rightarrow \bar{e}$ are given in Figure 4. The type and effect system for the source language is obtained by omitting the $\Rightarrow \bar{e}$ parts. As one can see, an effect is introduced by reading or updating a private reference: the name of the thread that owns the reference is added to the effect. At creation, a reference's type gets the name of the current thread as the "region" in which the reference is created. The only constraint (apart from the standard unification constraints) we have in the system regards the typing of thread creation: to type $(\text{thread } e)$ one has to assume that the thread only reads and writes the private references that it has created, which is expressed with $\varphi \subseteq \{t\}$, and that it does not import such references from the context, that is, its name t does not appear in the typing context Γ (i.e. in the types assigned by this context to variables). In other words, to be typable, a thread must create ($t \notin \Gamma$) the references that it is using ($\varphi \subseteq \{t\}$), apart obviously from the shared ones.

In order to prove a type safety result for the source language, namely that in a typable configuration a thread does not attempt to access a private reference that it does not own, we need to extend the typing to configurations. In the extended typing $\Gamma \vdash (S, \Omega, T)$, the context Γ not only assigns types to variables, but also to references. The typing of the memory is as follows:

$$\Gamma \vdash S \Leftrightarrow_{\text{def}} \begin{cases} \text{dom}(S) \subseteq \text{dom}(\Gamma) \ \& \ \text{for any } p \in \text{dom}(S) \\ \Gamma(p) = \theta \text{sref} \Rightarrow \exists t. \Gamma \vdash_t S(p) : \emptyset, \theta \\ \Gamma(p) = \theta \text{ref}_t \Rightarrow \Gamma \vdash_t S(p) : \emptyset, \theta \end{cases}$$

Then for thread systems

$$\Gamma \vdash T \Leftrightarrow_{\text{def}} \forall t \in \text{dom}(T) \exists \varphi, \tau. \varphi \subseteq \{t\} \ \& \ \Gamma \vdash_t T(t) : \varphi, \tau$$

Finally $\Gamma \vdash (S, \Omega, T)$ means $\Gamma \vdash S$ and $\Gamma \vdash T$.

The following is a standard property of type derivations that will be needed to prove the safety result for the type system.

REMARK 5.1. *For any typable expression $\Gamma \vdash_t \mathbf{E}[e] : \varphi, \theta$ there exist ψ, σ and t' such that $\Gamma \vdash_{t'} e : \psi, \sigma$.*

It is easy to see that values have an empty effect, formalized in the following remark.

REMARK 5.2. *For all $v \in \text{Val}$, with $\Gamma \vdash_t v : \varphi, \theta$ we have $\varphi = \emptyset$.*

The proof of the type safety result follows the classical steps (see [35]) of type safety proofs. In particular, we show a Subject Reduction property, using a Substitution and a Replacement lemmas.

LEMMA (SUBSTITUTION) 5.3. *Let $\Gamma, x : \sigma \vdash_t e : \varphi, \tau$ be a valid type derivation with $\varphi \subseteq \{t\}$ and $x \notin \Gamma$. Let also $\Gamma \vdash_t v : \emptyset, \sigma$ be a valid type derivation for some value v , then $\Gamma \vdash_t \{x \mapsto v\}e : \varphi, \tau$.*

PROOF SKETCH: The proof is by induction on the derivation of $\Gamma, x : \sigma \vdash_t e : \varphi, \tau$. We only examine some cases:

- $e = (!e')$. We have $\Gamma, x : \sigma \vdash_{t'} e' : \varphi', \theta$ for some φ', t' and θ . By the typing rules we have two cases according to the type θ . If $\theta = \tau \text{ref}_{t'}$, we have that $t = t'$ for $t' \in \varphi'$ and $\varphi' \subseteq \{t\}$. By the induction hypotheses we have that $\Gamma \vdash_t \{x \mapsto v\}e' : \varphi', \theta$. Thus, by the typing rule for dereferencing we obtain $\Gamma \vdash_t \{x \mapsto v\}(!e') : \varphi' \cup \{t\}, \tau$ for $(!e')$ has $\{t\}$ as an immediate effect. The case where $\theta = \tau \text{sref}$ is similar with $\varphi' = \varphi$, disregarding the remark about $t = t'$.
- $e = (e_0 := e_1)$. We apply the induction hypothesis directly for e_1 and perform a similar case analysis to that of the case $e = (!e')$ for e_0 . We conclude by applying the typing rule for assignment.

□

LEMMA (REPLACEMENT) 5.4. *Let $\Gamma \vdash_t \mathbf{E}[e] : \varphi, \tau$ with $\varphi \subseteq \{t\}$. Let also $\Gamma \vdash_t e : \psi, \sigma$ and $\Gamma' \vdash_t e' : \psi', \sigma$ with $\Gamma \subseteq \Gamma'$ and $\psi' \subseteq \psi$. We can conclude that there exists φ' such that $\varphi' \subseteq \varphi$ and $\Gamma' \vdash_t \mathbf{E}[e'] : \varphi', \tau$.*

PROOF SKETCH: The proof proceeds by induction on the evaluation context \mathbf{E} and by cases on the corresponding frame \mathbf{F} where $\mathbf{E} = \mathbf{E}'[\mathbf{F}]$. We develop only some cases, the others being direct from the induction hypothesis:

- $\mathbf{F} = (! \square)$. As in the Lemma 5.3, we have cases according to the type of σ . If $\sigma = \theta \text{ref}_{t'}$ we have that $t = t'$ by $\Gamma \vdash_t (!e) : \psi \cup \{t\}, \theta$ and $\psi \cup \{t'\} \subseteq \{t\}$. It follows from the hypothesis that $\psi' \cup \{t\} \subseteq \psi \cup \{t\}$, and thus we can conclude by the induction hypothesis. The case where $\sigma = \theta \text{sref}$ is trivial applying the induction hypothesis.
- $\mathbf{F} = (\square := e_1)$. By a similar argument to the previous case.

□

Next we prove a substitution property on thread names that we will need to prove the Subject Reduction lemma.

LEMMA 5.5. *Given a valid typing judgement $\Gamma \vdash_t e : \varphi, \tau$ with $\varphi \subseteq \{t\}$ and let t'' be a thread name that does not occur in Γ nor τ . We conclude that $\{t \mapsto t''\}(\Gamma \vdash_t e : \varphi, \tau)$ is a valid typing judgement.*

PROOF SKETCH: The proof is by induction on the derivation of $\Gamma \vdash_t e : \varphi, \tau$. We only examine the case $e = (\text{thread } e')$, all the other cases being a direct consequence of the induction hypothesis. By the typing of the thread expression, we have that $\Gamma \vdash_{t'} e' : \varphi, \text{unit}$. There are two cases to consider: if $t' \neq t''$ the conclusion is direct by the induction hypothesis; if $t' = t''$ we have to avoid the capture of t' by the substitution. Let \hat{t} be a thread name that does not appear in the derivation of $\Gamma \vdash_t e : \varphi, \tau$. We have, by the induction hypothesis, a valid typing judgment $\{t' \mapsto \hat{t}\}(\Gamma \vdash_{t'} e' : \varphi, \text{unit})$, and hence we can apply the induction hypothesis a second time to conclude. □

We can now prove a Subject Reduction property that allows to preserve the typing hypothesis along computations.

LEMMA (SUBJECT REDUCTION) 5.6. *If $\Gamma \vdash (S, \Omega, T)$ and $(S, \Omega, T) \rightarrow (S', \Omega', T')$ then $\Gamma' \vdash (S', \Omega', T')$ for some Γ' with $\Gamma \subseteq \Gamma'$.*

PROOF: By cases on the redex being reduced in the configuration (S, Ω, T) with $T = ((t, \mathbf{E}[r]) \parallel T'')$ and r the redex being reduced. We only examine some cases:

- $r = (\lambda xev)$. Then we have $S = S', \Omega = \Omega'$ and $T' = ((t, \mathbf{E}[\{x \mapsto v\}e]) \parallel T'')$. We need to prove that if $\Gamma \vdash_t \mathbf{E}[\{x \mapsto v\}e] : \varphi, \tau$ then $\Gamma \vdash \mathbf{E}[(\lambda xev)] : \varphi, \tau$. By the Remark 5.1 we have that there are ψ and σ such that $\Gamma \vdash_t (\lambda xev) : \psi, \sigma$. By the Substitution Lemma 5.3 we get that $\Gamma \vdash_t \{x \mapsto v\}e : \psi, \sigma$, and we conclude the case applying the Replacement Lemma 5.4.
- $r = (\text{ref } e)$. By Remark 5.1 we have $\Gamma \vdash_t (\text{ref } e) : \varphi, \theta \text{ref}_t$ for some θ and $\varphi = \emptyset$. By the semantics we have that $(\text{ref } e)$ evaluates to p for some $p \in \text{Ref}$ with $p \notin \text{dom}(S)$; with $S' = S \cup \{p \mapsto v\}$ and $\Omega' = (\Omega \cup \{p \mapsto t\})$. Clearly by typing we have $\Gamma \vdash_t v : \theta$ and therefore $\Gamma, p : \theta \text{ref}_t \vdash S'$ with $S' = S \cup \{p \mapsto v\}$. Hence we have $\Gamma, p : \theta \text{ref}_t \vdash_t p : \emptyset, \theta \text{ref}_t$. We conclude applying Lemma 5.4.
- $r = (!p)$. Notice first that $S' = S$ and $\Omega' = \Omega$. There are two cases depending on the type σ in $\Gamma \vdash_t p : \emptyset, \sigma$: if $\sigma = \theta \text{sref}$ we conclude by Lemma 5.4 observing that if v is the value read by the store typing we have $\Gamma \vdash_t v : \emptyset, \theta$; if $\sigma = \theta \text{ref}_{t'}$ we have $t' = t$ for $t' \in \varphi$ by typing and $\varphi \subseteq \{t\}$ by the typing of configurations. Finally, by typing of the store we have that if v is the value read from S we have $\Gamma \vdash_t v : \emptyset, \theta$, and we reach the conclusion by Lemma 5.4.
Notice that the case $r = (p := w)$ is similar to this one, with $S' = S[p := w]$ and the store typing is preserved by the typing in the expression.
- $r = (\text{thread } e)$. We have that $\Gamma \vdash_{t'} e : \varphi, \text{unit}$ with $\varphi \subseteq \{t'\}$ by Remark 5.1 and the typing of the thread expression. The conclusion is immediate once we see by Lemma 5.5 that $\{t' \mapsto t''\}(\Gamma \vdash_{t'} e : \varphi, \text{unit})$ with t'' a fresh thread name not present

in Γ nor the type or effect of any of the other threads in T'' (i.e. t'' does not occur in $\Gamma \vdash (S, \Omega, T)$).

□

Finally we can prove that non-shared references can only be used by a single thread, in particular, the one that creates it.

PROPOSITION 5.7. *Let e be a typable closed expression of the source language, with $\Gamma \vdash_t e : \varphi, \tau$ where $\varphi \subseteq \{t\}$. Let $(C_i \rightarrow C_{i+1})_{0 \leq i \leq n}$ be a sequence of transitions $C_i = (S_i, \Omega_i, T_i)$ with $S_0 = \emptyset = \Omega_0$ and $T_0 = \{(t, e)\}$, where the pointer p has been created as a ref. If $T_n(t') = \mathbf{E}[(!p)]$ or $T_n(t') = \mathbf{E}[(p := v)]$ then $\Omega_n(p) = t'$.*

PROOF: By the hypothesis the reference p has been created as a ref. Let C_j with $0 \leq j \leq n$, be the first configuration in which p appears. Notice that if p is created as ref for all $j \leq i \leq n$ we have $\Omega_j(p) = \Omega_i(p)$ for the semantics has no means to change Ω for pointers created by ref. By the Subject Reduction Lemma 5.6 we have that there is a typing $\Gamma_n \vdash C_n$ and thus $\Gamma_n \vdash_{t'} \mathbf{E}[(!p)] : \varphi, \tau$ with $\varphi \subseteq \{t'\}$. Then by Remark 5.1 we have $\Gamma_n \vdash_{t'} (!p) : \varphi', \sigma$ for some φ' and σ . Suppose that $\Gamma(p) = \theta \text{ref}_{t''}$, then we have that $t'' = t'$, for $t'' \in \varphi'$ and $\varphi' \subseteq \varphi \subseteq \{t'\}$. This allows us to conclude that the reference p has been created by thread t' , and hence $\Omega_n(p) = t'$. The case $\mathbf{E}[(p := v)]$ is similar. □

To prove the correctness of the translation we show an operational correspondence between the semantics of an expression of the source language and the interleaving – non speculative – semantics of its translation. Later we will prove that typable translated expressions are *speculatively* data race free, hence proving the correspondence between the source language semantics and valid speculative executions of the target semantics.

To relate source language run-time expressions with their target semantic counterparts we will extend the translation to run-time expressions. Notice that shared references in the source language semantics are mapped to records containing a unique lock and a reference in the target semantics. Therefore we parameterize our run-time compiles-to relation with the lock assignment chosen by the semantics so far. We will denote by \mathcal{L} such injection from the set of references Ref to the set of locks Locks . The run-time compiles-to relation is almost identical to that of Figure 4, except for the addition of references (both shared and private) and the translation of shared variable accesses. An excerpt, including the translation of private and shared references as well as the assignment expression, is given in Figure 5; dereferencing a shared reference is similar to its mutation. The translation is guided by typing, which is justified by Lemma 5.6.

The compiles-to relation can now be extended to the store and to configurations (with $C = (S, \Omega, T)$ and $\bar{C} = (\bar{S}, L, \bar{T})$):

$$\begin{aligned}
S &\stackrel{\mathcal{L}}{\rightsquigarrow} \bar{S} \iff \begin{cases} \text{dom}(S) = \text{dom}(\bar{S}) & \& \\ \bar{S}(p) = \bar{S}(p) & \text{for all } p \in \text{dom}(S) \end{cases} \\
T &\stackrel{\mathcal{L}}{\rightsquigarrow} \bar{T} \iff \begin{cases} \text{dom}(T) = \text{dom}(\bar{T}) & \& \\ \bar{T}(t) = \bar{T}(t) & \text{for all } t \in \text{dom}(T) \end{cases} \\
C &\stackrel{\mathcal{L}}{\rightsquigarrow} \bar{C} \iff \begin{cases} S \stackrel{\mathcal{L}}{\rightsquigarrow} \bar{S} & \& \\ L = \{\mathcal{L}(p) \mid p \in \text{dom}(\mathcal{L}) \cap \text{dom}(\Omega)\} & \& \\ T \stackrel{\mathcal{L}}{\rightsquigarrow} \bar{T} & \end{cases}
\end{aligned}$$

Now we introduce a relation between configurations of the source language and configurations of the target language to prove their operational correspondence. The relation is parameterized by an initial typable expression e :

$$CR_e \bar{C} \iff \begin{cases} \vdash_t e : \varphi, \tau & \text{with } \varphi \subseteq \{t\} \& \\ (\emptyset, \emptyset, (t, e)) \xrightarrow{*} C & \& \\ (\emptyset, \emptyset, (t, \bar{e})) \xrightarrow{*} \bar{C} & \& \\ C \stackrel{\mathcal{L}}{\rightsquigarrow} \bar{C} & \text{for some } \mathcal{L} \end{cases}$$

$$\begin{array}{c}
\frac{}{\Gamma, p : \theta \text{ref}_{t'} \vdash_t p : \emptyset, \theta \text{ref}_{t'} \xrightarrow{\mathcal{L}} p} \quad \frac{}{\Gamma, p : \theta \text{sref} \vdash_t p : \emptyset, \theta \text{sref} \xrightarrow{\mathcal{L}} \{\text{lock} = \mathcal{L}(p), \text{val} = y\}} \quad p \in \text{dom}(\mathcal{L}) \\
\frac{}{\Gamma \vdash_t e_0 : \varphi_0, \theta \text{sref} \xrightarrow{\mathcal{L}} \bar{e}_0} \quad \frac{}{\Gamma \vdash_t e_1 : \varphi_1, \theta \xrightarrow{\mathcal{L}} \bar{e}_1} \\
\frac{}{\Gamma \vdash_t (e_0 := e_1) : \varphi_0 \cup \varphi_1, \text{unit} \xrightarrow{\mathcal{L}} (\text{let } x = \bar{e}_0 \text{ in } (\text{let } y = \bar{e}_1 \text{ in } (\text{with } x.\text{lock} \text{ do } x.\text{val} := y)))} \\
\frac{}{\Gamma \vdash_t p : \emptyset, \theta \text{sref} \xrightarrow{\mathcal{L}} \bar{p}} \quad \frac{}{\Gamma \vdash_t e_1 : \varphi_1, \theta \xrightarrow{\mathcal{L}} \bar{e}_1} \quad p \in \text{Val} \\
\frac{}{\Gamma \vdash_t (p := e_1) : \varphi_1, \text{unit} \xrightarrow{\mathcal{L}} (\text{let } y = \bar{e}_1 \text{ in } (\text{with } \bar{p}.\text{lock} \text{ do } \bar{p}.\text{val} := y))} \\
\frac{}{\Gamma \vdash_t p : \emptyset, \theta \text{sref} \xrightarrow{\mathcal{L}} \bar{p}} \quad \frac{}{\Gamma \vdash_t v : \emptyset, \theta \xrightarrow{\mathcal{L}} \bar{w}} \quad p, v \in \text{Val} \\
\frac{}{\Gamma \vdash_t (p := v) : \emptyset, \text{unit} \xrightarrow{\mathcal{L}} (\text{with } \bar{p}.\text{lock} \text{ do } \bar{p}.\text{val} := \bar{w})}
\end{array}$$

Figure 5: Run-time compiles-to relation (excerpt)

LEMMA 5.8. *If $CR_e \bar{C}$ and $C \rightarrow C'$, then there is \bar{C}' such that $\bar{C} \xrightarrow{*} \bar{C}'$ with $C' R_e \bar{C}'$.*

(The proof is routine by cases on the redex being reduced in the source language semantics.)

In what follows we shall use the notation \bar{C} to mean $C \xrightarrow{\mathcal{L}} \bar{C}$ where the lock assignment \mathcal{L} is implicit when clear from the context.

To prove the simulation of the target language semantics by the source language we shall use the following result stating that there are no data races in normal executions of translated typable expressions:

REMARK 5.9. *If $\vdash_t e : \varphi, \tau$ with $\varphi \subseteq \{t\}$, and $(\emptyset, \emptyset, (t, \bar{e})) \xrightarrow{*} \hat{C}$ is a normal execution, then \hat{C} contains no races.*

PROOF: The result is a consequence of the Lemma 5.7 for references created by ref, and it is enforced by the compilation for references created by sref. \square

Notice that the translation introduces redexes to the target language computation that are not present at the source level. To refer to these steps we introduce the *administrative steps* relation. This relation inductively pairs a source configuration C to a target configuration \hat{C} (denoted $C \uparrow \hat{C}$) if whenever $C = (S, \Omega, T) \rightarrow (S', \Omega', T')$ by reducing a redex in thread t in the source semantics we have

- if $\bar{C} \rightarrow \hat{C}$ by reducing t in the target, then $\overline{T'(t)} \neq \hat{T}(t)$, and
- if $C \uparrow \hat{C}_0$ and $\hat{C}_0 \rightarrow \hat{C}$ by reducing t then $\overline{T'(t)} \neq \hat{T}(t)$.

Thus, a target configuration is administratively related to a source configuration if in the latter there is one or more threads performing administrative steps corresponding to redexes that can immediately be reduced in the source configuration.

To prove the operational correspondence from the target semantics to the source semantics we first prove that if $CR_e \bar{C}$, then configurations reachable from \bar{C} are administratively related to some configuration reachable from C in the source semantics. Next we prove that administratively related configurations converge to configurations related by R_e , concluding our operational correctness result.

LEMMA 5.10. *If $CR_e \bar{C}$ and $\bar{C} \xrightarrow{*} \hat{C}$ we have one of the following:*

- $C \uparrow \hat{C}$, or
- exists C' a source configuration with $C \xrightarrow{*} C'$ and $C' \uparrow \hat{C}$, or
- exists C' a source configuration with $C \xrightarrow{*} C'$ and $\bar{C}' = \hat{C}$.

PROOF SKETCH: The proof is by induction on the number of steps required to reach \hat{C} from \bar{C} . We only examine the inductive case: Suppose $\hat{C}_0 \rightarrow \hat{C}$ by reducing the thread t . If the step is just another administrative step we obtain the conclusion (i). If, on the contrary, $C \uparrow \hat{C}$ does not hold, we have that thread t finished performing its administrative steps. The proof proceeds by case analysis on the next redex to be reduced by thread t in the source configuration C (i.e. r if $C = (S, \Omega, T' \parallel (t, \mathbf{E}[r]))$). We only examine some cases:

- $r = (\text{sref } v)$. If the thread t is the only one performing administrative steps in \hat{C}_0 we have that $\hat{C} = \bar{C}'$ (as it has been proved in Lemma 5.8), thus obtaining conclusion (iii). Otherwise, if there are other threads performing administrative steps in \hat{C}_0 we have that $C' \uparrow \hat{C}$, for threads other than t remain unchanged. Hence we obtain the conclusion (ii).
- $r = (!p)$. The only case to consider is when $p \in \text{dom}(\mathcal{L})$, else the configuration would not be administratively related. Notice that since it is the last administrative step, the thread t held the lock $\mathcal{L}(p)$ and by Remark 5.9 no other thread could have modified the store for the reference p ; which implies the read value is exactly $\overline{S(p)}$ if $C = (S, \Omega, T)$. We use the same case analysis as in the previous case to obtain conclusions (iii) or (ii). Then we use a similar argument for the case $r = (p := w)$ with $p \in \text{dom}(\mathcal{L})$.

\square

LEMMA 5.11. *If $C \uparrow \hat{C}$, then there exists C' , a source semantics configuration, with $C \xrightarrow{*} C'$ in the source semantics and $\hat{C} \xrightarrow{*} \bar{C}'$ in the target semantics.*

PROOF SKETCH: The proof is by induction on the number of steps required to reach \hat{C} from \bar{C} . We only examine the inductive case; suppose $\hat{C}_0 \rightarrow \hat{C}$ by reducing the thread t . We proceed by case analysis on the next redex to be reduced in thread t (i.e. r if $C = (S, \Omega, T' \parallel (t, \mathbf{E}[r]))$) in the source configuration C (we only examine some cases):

- $r = (!p)$. Then we have that $p \in \text{dom}(\mathcal{L})$. By the translation we have $\overline{\mathbf{E}[(!p)]} = \overline{\mathbf{E}[(\text{with } \bar{p}.\text{lock} \text{ do } (!\bar{p}.\text{val})]}$. Notice that the first action of t (i.e. reducing $\bar{p}.\text{lock}$ to $\mathcal{L}(p)$) is non-conflicting with any other action, therefore we can perform exactly the steps given by the induction hypothesis for the other threads, and finally reduce $\overline{\mathbf{E}[(\text{with } \mathcal{L}(p) \text{ do } (!\bar{p}.\text{val})]}$ reaching a configuration \bar{C}'' related by R_e with C'' , where C'' results from C'_0 (the resulting configuration from the induction hypotheses) by reducing the thread t .

$$\begin{array}{c}
\frac{\Gamma \vdash_t e_0 : \varphi_0, \text{lock} \quad \Gamma \vdash_t e_1 : \varphi_1, \theta \text{ sref}}{\Gamma \vdash_t \{\text{lock} = e_0, \text{val} = e_1\} : \varphi_0 \cup \varphi_1, \{\text{lock} : \text{lock}, \text{val} : \theta\}} \quad \frac{\Gamma \vdash_t e : \varphi, \{\text{lock} : \text{lock}, \text{val} : \theta\}}{\Gamma \vdash_t e.\text{val} : \varphi, \theta \text{ sref}} \\
\frac{\Gamma \vdash_t e_0 : \varphi_0, \text{lock} \quad \Gamma \vdash_t e_1 : \varphi_1, \tau}{\Gamma \vdash_t (\text{with } e_0 \text{ do } e_1) : \varphi_0 \cup \varphi_1, \tau} \quad \frac{\Gamma, \ell : \text{lock} \vdash_t e : \varphi, \tau}{\Gamma \vdash_t (\text{new } \ell \text{ in } e) : \varphi, \tau} \quad \frac{\Gamma \vdash_t e : \varphi, \theta}{\Gamma \vdash_t (\text{ref } e) : \varphi, \theta \text{ ref}_t} \quad \frac{\Gamma \vdash_t e : \varphi, \theta}{\Gamma \vdash_t (\text{ref}^s e) : \varphi, \theta \text{ sref}}
\end{array}$$

Figure 6: Type and effect system for the target language (excerpt)

If the step taken is not the first of $\bar{\mathbf{E}}[(\text{with } \bar{p}.\text{lock do } (!\bar{p}.\text{val}))]$, the thread t acquires (or holds already) the lock $\mathcal{L}(p)$ in which case we can reduce it until we reach $\bar{\mathbf{E}}[\bar{S}(p)]$ and then perform the exact steps given by the induction hypothesis for the other threads. This procedure reaches a configuration \bar{C}'' where \bar{C}'' is obtained from \bar{C} by reducing first the thread t and then performing the steps in $\bar{C} \xrightarrow{*} \bar{C}'_0$, other than those by thread t , to that configuration.

□

LEMMA 5.12. *If $CR_e\bar{C}$ and $\bar{C} \xrightarrow{*} \hat{C}$, then there is C' , a source language configuration, such that $C \xrightarrow{*} C'$ in the source semantics and $\hat{C} \xrightarrow{*} C'$ in the target semantics.*

(The proof is a direct consequence of Lemmas 5.10 and 5.11.)

THEOREM (OPERATIONAL CORRESPONDENCE) 5.13. *Let e be a closed typable expression of the source language. If $(\emptyset, \emptyset, (t, e)) \xrightarrow{*} C$ is a valid computation of the source language semantics, then there exist normal computation of the target language semantics $(\emptyset, \emptyset, (t, \bar{e})) \xrightarrow{*} \bar{C}$ such that $CR_e\bar{C}$. Conversely, if $(\emptyset, \emptyset, (t, \bar{e})) \xrightarrow{*} \bar{C}$ is a normal computation of the target language semantics, then there exists C' a configuration of the source language semantics such that $\bar{C} \xrightarrow{*} C'$ in the target language semantics, $C \xrightarrow{*} C'$ in the source language semantics and $C' R_e \bar{C}$.*

(The proof is direct by Lemmas 5.8 and 5.12.)

In the rest of the section we prove the correspondence between computations of the source language semantics and valid speculative computations of the target language semantics.

We now prove that translations of typable source language expressions are speculatively data race free. To prove that private references in the source language are not shared in valid speculative computations we introduce a type and effect system that mimics the one of Figure 4 in the target language. To simplify the type system we will assume that the translation of the sref construct annotates the ref expression as (ref^s) to differentiate shared and private references in the target language. Thus, the translation of $(\text{sref } e)$ becomes $(\text{new } \ell \text{ in } \{\text{lock} = \ell, \text{val} = (\text{ref}^s \bar{e})\})$, where the annotation s in ref^s has no semantical meaning whatsoever. The typing rules that differ from those of Figure 4 are given in Figure 6.

We can easily extend the type and effect system of Figure 6 to run-time expressions, by adding references and locks to the typing context as we did before. Moreover, we can extend the typing to the store and target language semantics configurations as we did for the source language type system. Also, the Subject Reduction result of the source language (Lemma 5.6) can be easily reproduced for the type system of Figure 6, where we modify the Replacement Lemma 5.4 to account for speculative contexts (Σ).

LEMMA (REPLACEMENT - SPECULATIONS) 5.14. *Let $\Gamma \vdash_t \Sigma[e] : \varphi, \tau$ with $\varphi \subseteq \{t\}$. Let also $\Gamma \vdash_t e : \psi, \sigma$ and $\Gamma' \vdash_t e' : \psi', \sigma$ with $\Gamma \subseteq \Gamma'$ and $\psi' \subseteq \psi$. We can conclude that there exists φ' such that $\varphi' \subseteq \varphi$ and $\Gamma' \vdash_t \Sigma[e'] : \varphi', \tau$.*

(The proof is a simple extension to that of Lemma 5.4.)

LEMMA (SUBJECT REDUCTION - TARGET) 5.15. *If $\Gamma \vdash (S, L, T)$ and $(S, L, T) \xrightarrow{*} (S', L', T')$, by reducing a redex in a speculative context, then $\Gamma' \vdash (S', L', T')$ for some Γ' with $\Gamma \subseteq \Gamma'$.*

(The proof is a simple extension to that Lemma 5.6.)

One can easily see that references in the target language that correspond to private references of the source semantics are never shared.

LEMMA 5.16. *Let e be a typable closed expression of the source language with $\Gamma \vdash_t e : \varphi, \tau \Rightarrow \bar{e}$ and $\varphi \subseteq \{t\}$. Let $(\emptyset, \emptyset, \bar{e}) \xrightarrow{*} C$ be a valid speculative computation, with $\Gamma' \vdash C$ and $C = (S, L, T' \parallel (t, \Sigma[(!p)]))$. If $\Gamma'(p) = \theta \text{ ref}_{t'}$ for some θ and t' , then $t = t'$.*

PROOF SKETCH: By the typing of configurations we have that $\Gamma' \vdash_t \Sigma[(!p)] : \psi, \sigma$ for some σ and ψ with $\psi \subseteq \{t\}$. Moreover if $\Gamma'(p) = \theta \text{ ref}_{t'}$ we have that $t' \in \psi$ and therefore $t' = t$. □

Now we have the property that typable source expressions are translated into robust programs. This is our second main result.

THEOREM 5.17. *For any closed typable expression e of the source language, with $\Gamma \vdash_t e : \varphi, \tau$ where $\varphi \subseteq \{t\}$, the translation \bar{e} of e is speculatively data race free.*

(The proof is a simple consequence of the translation of shared references which instruments synchronization, and the Lemma 5.16 for private references of the source language.)

Finally we can show that there is an operational correspondence between the typable expressions of the source language and valid speculative computations of the target language.

THEOREM 5.18. *Let e be a closed typable expression of the source language. If $(\emptyset, \emptyset, (t, e)) \xrightarrow{*} (S, \Omega, T)$ then there exist \bar{S}, \bar{T}, L and a valid speculative computation $\gamma : (\emptyset, \emptyset, (t, \bar{e})) \xrightarrow{*} (\bar{S}, L, \bar{T})$ such that $\text{dom}(\bar{S}) = \text{dom}(S)$ and $\bar{S}(p) = \bar{S}(p)$ for any p . Conversely, if $\gamma : (\emptyset, \emptyset, (t, \bar{e})) \xrightarrow{*} (S', L', T')$ is a valid speculative computation, then there exists (S, Ω, T) such that $(S', L', T') \xrightarrow{*} (\bar{S}, L, \bar{T})$ for some L and $(\emptyset, \emptyset, (t, e)) \xrightarrow{*} (S, \Omega, T)$ with $\text{dom}(S) = \text{dom}(\bar{S})$ and $\bar{S}(p) = \bar{S}(p)$ for any p .*

PROOF SKETCH: The result is a direct consequence of Theorems 5.13, 4.13 and 5.17. □

6. Conclusion

We have given a formal definition for speculative computations which, we believe, is quite general. We have, in particular, checked the classical ‘‘litmus tests’’ that are considered when dealing with memory models, and we have seen that most of these are correctly described in our setting (except in the cases relying on code transformations). This means that our semantics is quite permissive as regards the allowed optimizations, while being correct for sequential programs, but also that it is very easy to use for justifying that a particular outcome is allowed or forbidden. This is clearly a benefit from using a standard operational style. We think that our model of speculative computation could be used to justify implementation

techniques, and to design formal analysis and verification methods for checking concurrent programs, as well as developing, as we did in the last section, programming styles for safe multithreading.

References

- [1] M. ABADI, C. FLANAGAN, S.N. FREUND, *Types for safe locking: static race detection for Java*, ACM TOPLAS Vol. 28 No. 2 (2006) 207-255.
- [2] S.A. ADVE, K. GHARACHORLOO, *Shared memory consistency models: a tutorial*, IEEE Computer Vol. 29 No. 12 (1996) 66-76.
- [3] S. ADVE, M.D. HILL, *Weak ordering – A new definition*, ISCA'90 (1990) 2-14.
- [4] D. ASPINALL, J. ŠEVČÍK, *Java memory model examples: good, bad and ugly*, VAMP'07 (2007).
- [5] D. ASPINALL, J. ŠEVČÍK, *On validity of program transformations in the JAVA memory model*, ECOOP'08, Lecture Notes in Comput. Sci. 5142 (2008) 27-51.
- [6] G. BERRY, J.-J. LÉVY, *Minimal and optimal computations of recursive programs*, J. of ACM 26 (1979) 148-175.
- [7] H.-J. BOEHM, S.V. ADVE, *Foundations of the C++ concurrency model*, PLDI'08 (2008) 68-78.
- [8] G. BOUDOL, *Computational semantics of term rewriting systems*, in Algebraic Methods in Semantics (M. Nivat & J.C. Reynolds Eds), Cambridge University Press (1985) 169-236.
- [9] G. BOUDOL, *A deadlock-free semantics for shared memory concurrency*, ICTAC'09, Lecture Notes in Comput. Sci. 5684 (2009) 140-154.
- [10] G. BOUDOL, G. PETRI, *Relaxed memory models: an operational approach*, POPL'09 (2009) 392-403.
- [11] C. BOYAPATI, R. LEE, M. RINARD, *Ownership types for safe programming: preventing data-races and deadlocks*, OOPSLA'02 (2002) 211-230.
- [12] F.W. BURTON, *Speculative computation, parallelism, and functional programming*, IEEE Trans. on Computers, Vol. C-34, No. 12 (1985) 1190-1193.
- [13] F. DABROWSKI, F. BOUSSINOT, *Cooperative threads and preemptive computations*, Proceeding of TV'06, Workshop on Multithreading in Hardware and Software: Formal Approaches to Design and Verification, FLoC'06 (2006) 40-51.
- [14] M. FELLEISEN, D.P. FRIEDMAN, *Control operators, the SECD-machine and the λ -calculus*, in Formal Description of Programming Concepts III (Elsevier, M. Wirsing Ed.) (1986) 193-217.
- [15] C. FLANAGAN, M. FELLEISEN, *The semantics of future and its use in program optimization*, POPL'95 (1995) 209-220.
- [16] F. GABBAY, A. MENDELSON, *Using value prediction to increase the power of speculative execution hardware*, ACM Trans. on Computer Systems Vol. 16 No. 3 (1998) 234-270.
- [17] K. GHARACHORLOO, D. LENOSKI, J. LAUDON, P. GIBBONS, A. GUPTA, J. HENNESSY, *Memory consistency and event ordering in scalable shared-memory multiprocessors*, ACM SIGARCH Computer Architecture News Vol. 18 No. 3a (1990) 15-26.
- [18] J. GOSLING, B. JOY, G.L. STEELE, G. BRACHA, *The JAVA Language Specification*, 3rd edition, Prentice Hall (2005).
- [19] D. GROSSMAN, *Type-safe multithreading in Cyclone*, TLDI'03 (2003) 13-25.
- [20] R.H. HALSTEAD, *Multilisp: a language for concurrent symbolic computation*, ACM TOPLAS Vol. 7 No. 4 (1985) 501-538.
- [21] M. KATZ, D. WEISE, *Continuing into the future: on the interaction of futures and first-class continuations*, ACM Conf. on Lisp and Functional Programming (1990) 176-184.
- [22] T. KNIGHT, *An architecture for mostly functional languages*, ACM Conf. on Lisp and Functional Programming (1986) 105-112.
- [23] L. LAMPORT, *How to make a multiprocessor computer that correctly executes multiprocess programs*, IEEE Trans. on Computers Vol. 28 No. 9 (1979) 690-691.
- [24] J.-J. LÉVY, *Optimal reductions in the lambda calculus*, in To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism (J.P. Seldin, J.R. Hindley, Eds), Academic Press (1980) 159-191.
- [25] M.H. LIPASTI, C.B. WILKERSON, J.P. SHEN, *Value locality and load value prediction*, ASPLOS'96 (1996) 138-147.
- [26] J.M. LUCASSEN, D.K. GIFFORD, *Polymorphic effect systems*, POPL'88 (1988) 47-57.
- [27] J. MANSON, W. PUGH, S.A. ADVE, *The Java memory model*, POPL'05 (2005) 378-391.
- [28] M.K. MARTIN, D.J. SORIN, H.W. CAIN, M.D. HILL, M.H. LIPASTI, *Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing*, 34th International Symp. on Microarchitecture (2001) 328-337.
- [29] L. MOREAU, *The semantics of Scheme with futures*, ICFP'96 (1996) 146-156.
- [30] A. NAVABI, S. JAGANNATHAN, *Exceptionally safe futures*, COORDINATION'09, Lecture Notes in Comput. Sci. 5521 (2009) 47-65.
- [31] V. SARASWAT, R. JAGADEESAN, M. MICHAEL, C. von PRAUN, *A theory of memory models*, PPoPP'07 (2007) 161-172.
- [32] S. SARKAR, P. SEWELL, F. ZAPPA NARDELLI, S. OWENS, T. RIDGE, T. BRAIBANT, M.O. MYREEN, J. ALGLAVE, *The semantics of x86-CC multiprocessor machine code*, POPL'09 (2009) 379-391.
- [33] J.E. SMITH, *A study of branch prediction strategies*, ISCA'81 (1981) 135-148.
- [34] A. WELC, S. JAGANNATHAN, A. HOSKING, *Safe futures for JAVA*, OOPSLA'05 (2005) 439-453.
- [35] A. WRIGHT, M. FELLEISEN, *A syntactic approach to type soundness*, Information and Computation Vol. 115 No. 1 (1994) 38-94.