A Theory of Speculative Computation*

Gérard Boudol and Gustavo Petri

INRIA, 06902 Sophia Antipolis, France

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 computations. Regarding concurrent programs, 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.

1 Introduction

Speculative computation [8,16] is an implementation technique that aims at speeding up the execution of programs, by computing pieces of code in advance, possibly in parallel with the rest of the program, 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. By considering parallel composition of speculations, we also include relaxed memory models [1] into this picture – though not those that try to capture compiler optimizations, that transform the code on the basis of semantical reasoning (see [4,20,24]).

Let us see some examples of speculative computations. In these examples, we use ML's notation $!p$ for dereferencing a pointer, and $\Box$ to mean termination, and we present the speculation as a sequence of transitions, each labelled by an action to be performed. More specifically, $\mathsf{rd}_{p,v}$ is the action of reading the value v for pointer p, $\mathsf{wr}_{p,v}$ means the action of writing the value v for p in the

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

A.D. Gordon (Ed.): ESOP 2010, LNCS 6012, pp. 165–184, 2010.

memory, and $\swarrow$ is taking the first branch in a conditional branching. In our first example, the write to pointer q is reordered with respect to the read of p, which predicts the value tt – regardless of the actual value found in the memory, which is ignored at this stage:

$$(r := !\,p)\,;(q := tt) \xrightarrow{\mathsf{wr}_{q,tt}} (r := !\,p)\,;() \xrightarrow{\mathsf{rd}_{p,tt}} (r := tt)\,;() \tag{1}$$

In our second example

$$\begin{aligned}(\mathsf{if}\ !\,p\ \mathsf{then}\ q := tt\ \mathsf{else}\ ()) &\xrightarrow{\mathsf{wr}_{q,tt}} (\mathsf{if}\ !\,p\ \mathsf{then}\ ()\ \mathsf{else}\ ()) \\ &\xrightarrow{\mathsf{rd}_{p,tt}} (\mathsf{if}\ tt\ \mathsf{then}\ ()\ \mathsf{else}\ ()) \\ &\xrightarrow{\swarrow} ()\end{aligned} \tag{2}$$

the assignment in the first branch is issued speculatively, and the value tt is guessed for p. In both cases, the write to q could be issued to run in parallel with the rest of the code.

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, optimizing the execution of sequential code. These implementations are quite often complex, as speculations are not always correct, and need to be aborted or undone is some cases. For instance, the two speculations above are intuitively correct, provided that the predicted values coincide with the actual ones, but it would be wrong, in Example (2), to perform the write for q if the value eventually read for $!\,p$ is ff. Due to the complexity of implementing speculations perhaps, the notion of a *valid speculation* does not seem to have been formally defined before, 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.

Our first and main aim in this work is to design a semantical framework to formalize in a comprehensive manner the notion of a speculative computation, and to characterize the ones that are valid for sequential programs. We adopt and extend the approach presented in [7], 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 [9], and using *value prediction* [11,19] 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* [26], 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 [14]. Our model therefore encompasses many speculative techniques.

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. Our criterion here is that a thread's

speculation is only valid if it preserves the sequential semantics. The equivalence of computations we use is the *permutation of transitions equivalence* introduced, for a purely functional language, by Berry and Lévy in [5], 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 (1) and (2) above are valid, by executing the same operations in normal order. 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. One could therefore use our formal model to assess the correctness of an implementation, showing that the latter only allows valid speculations to be performed.

As we shall see, valid speculations indeed preserve the semantics of sequential programs. This is no longer the case for *multithreaded* applications running on multiprocessor architectures. This is not surprising, since most optimizations found in relaxed memory models do not preserve the standard interleaving semantics – also known as "sequential consistency" [17] in the area of memory models –, see the survey [1]. For instance, continuing with Example (1), 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 a state where $!\,r = tt = !\,r'$ as an outcome of a valid speculative computation, that first issues the writes to q and p. 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 **R→RW**, according to the terminology of [1]. 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 stressing the generality of our framework, which offers a very relaxed semantical model.

Since the interleaving semantics of thread systems is not preserved by optimizing platforms, such as parallelized hardware, and since the latter are unlikely to be changed for the purpose of running concurrent programs, some conditions must be found for multithreaded applications to be executed correctly on these platforms. For instance, most memory models support the well-known "DRF guarantee," that asserts that programs free of data races, with respect to the interleaving semantics, are correctly executed in the optimized semantics [2,12,20]. However, with speculative computations, this guarantee fails. For instance, extending the second example given above, one can see that with the thread system

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

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 [13] Section 17.4.8, and [6] for

similar examples). This example, though not very interesting from a programming point of view, exhibits the failure of the DRF guarantee. 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

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

if initially $!\mathsf{data} = 0$ and $!\mathsf{flag} = ff$, we can speculate that $!\mathsf{data}$ in the second thread returns 0, and therefore get an unexpected value for r (the other instructions being processed in the normal way). Since speculating ahead of synchronization (unlock) is permitted in our model, 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.

Now a question is: what kind of property should concurrent programs possess to be "robust" against aggressive optimizations – and more precisely: speculations – found in optimized execution platforms, and how to ensure such robustness? In this paper we address the first part of this question[1]. We have seen that data race free concurrent programs are not necessarily robust – where "robust" means that the speculative semantics does not introduce unexpected behaviors (w.r.t. the normal order semantics) for the program under consideration. In this paper we show that *speculatively* data race free programs *are* robust – this is our main technical 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 – according to the speculative semantics – concurrent accesses, one of them being a write, to the same memory location. Then sequential programs in particular are robust, that is, speculative computation is a correct implementation for these programs.

Related work

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 [10] of Flanagan and Felleisen. This was later extended by Moreau in [22] to deal with mutable state and continuations (extending the work in [15] as regards the latter). A similar work regarding Java has been done by Jagannathan and colleagues, dealing with mutable state [27] and exceptions [23]. However, all these works on the future construct aim at preserving the sequential semantics, but they are not concerned with shared memory concurrency. Moreover, they do not include branch prediction and value prediction.

[1] For an answer to the second part we refer to the forthcoming extended version of this paper.

2 The Language

The 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. The syntax is:

$$\begin{array}{rll} e ::= v \mid (e_0 e_1) \mid (\mathsf{if}\ e\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1) \mid (\mathsf{ref}\, e) & & \textit{expressions} \\ \quad \mid (\mathsf{!}\, e) \mid (e_0 := e_1) \mid (\mathsf{thread}\, e) \mid (\mathsf{with}\ \ell\ \mathsf{do}\ e) & & \\ v ::= x \mid \lambda x e \mid \mathit{tt} \mid \mathit{ff} \mid () & & \textit{values} \end{array}$$

where ℓ is a *lock*, that is a name from a given infinite set $\mathcal{L}ocks$. 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 $(\mathsf{let}\ x = e_0\ \mathsf{in}\ e_1)$ 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) $p, q, \ldots$ that are names from a given infinite set $\mathcal{R}ef$. These are (run-time) values. Then we use the construct $(e \backslash \ell)$ 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:

$$\begin{array}{rll} p, q \ldots \in \mathcal{R}ef & & \textit{references} \\ v ::= \cdots \mid p & & \textit{run-time values} \\ e ::= \cdots \mid (e \backslash \ell) & & \textit{run-time expressions} \\ u ::= (\lambda x e v) \mid (\mathsf{if}\ \mathit{tt}\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1) \mid (\mathsf{if}\ \mathit{ff}\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1) & & \textit{redexes} \\ \quad \mid (\mathsf{ref}\, v) \mid (\mathsf{!}\, p) \mid (p := v) \mid (\mathsf{thread}\, e) \mid (\mathsf{with}\ \ell\ \mathsf{do}\ e) \mid (v \backslash \ell) & & \\ \mathbf{E} ::= [\,] \mid \mathbf{E}[\mathbf{F}] & & \textit{evaluation contexts} \\ \mathbf{F} = (\,[\,]\, e) \mid (v\, [\,]) \mid (\mathsf{if}\ [\,]\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1) & & \textit{frames} \\ \quad \mid (\mathsf{ref}\, [\,]) \mid (\mathsf{!}\, [\,]) \mid ([\,] := e) \mid (v := [\,]) \mid ([\,] \backslash \ell) & & \end{array}$$

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:

- $(v e)$ where the value v is not a function $\lambda x e'$;
- $(\mathsf{if}\ v\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1)$ where the value v is not a boolean value, tt or ff;
- $(\mathsf{!}\, v)$ 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 $e = \mathbf{E}[e']$.*

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

$$\begin{array}{lll} \mathbf{\Sigma} ::= [] \mid \mathbf{\Sigma}[\mathbf{\Phi}] & & \textit{speculation contexts} \\ \mathbf{\Phi} ::= \mathbf{F} \mid (e\,[]) \mid (\lambda x[]\,e) \mid (\mathsf{if}\ e\ \mathsf{then}\ []\ \mathsf{else}\ e_1) & & \textit{speculation frames} \\ \quad \mid (\mathsf{if}\ e\ \mathsf{then}\ e_0\ \mathsf{else}\ []) \mid (e := []) & & \end{array}$$

Let us comment briefly on the speculation contexts. With frames of the shape $(\lambda x[]\,e)$, one can for instance compute e_1 in the expression $(\lambda x e_1 e_0)$ – hence in $(\mathsf{let}\ x = e_0\ \mathsf{in}\ e_1)$ 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 $(\mathsf{let}\ x = \mathsf{future}\ e_0\ \mathsf{in}\ e_1)$ [10], where e_1 is computed *in advance*, or *in parallel* with e_0. With the frames $(\mathsf{if}\ e\ \mathsf{then}\ []\ \mathsf{else}\ e_1)$ and $(\mathsf{if}\ e\ \mathsf{then}\ e_0\ \mathsf{else}\ [])$, 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*" [26]. Notice that, by contrast, the construct $(\mathsf{with}\ \ell\ \mathsf{do}\ e)$ acts as a "speculation barrier," that is, $(\mathsf{with}\ \ell\ \mathsf{do}\ [])$ 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 construct[2], 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 $\lceil\mathbf{\Sigma}\rceil$, of locks held in the context $\mathbf{\Sigma}$, defined as follows:

$$\begin{array}{r} \lceil[]\rceil = \emptyset \\ \lceil\mathbf{\Sigma}[\mathbf{\Phi}]\rceil = \lceil\mathbf{\Sigma}\rceil \cup \lceil\mathbf{\Phi}\rceil \end{array} \qquad \text{where} \qquad \lceil\mathbf{\Phi}\rceil = \begin{cases} \{\ell\} \text{ if } \mathbf{\Phi} = ([]\backslash\ell) \\ \emptyset \quad \text{otherwise} \end{cases}$$

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, namely performing a β-reduction, denoted β, choosing a branch in a conditional construct ($\swarrow$ and $\searrow$), creating a new reference p in the store with some initial (closed) value $(\nu_{p,v})$, reading $(\mathsf{rd}_{p,v})$ or writing $(\mathsf{wr}_{p,v})$ a reference, spawning a new thread (spw_e), acquiring $(\overset{\frown}{\ell})$ or releasing $(\overset{\smile}{\ell})$ a lock ℓ. Then the syntax of actions is as follows:

$$\begin{array}{l} a ::= \beta \mid \swarrow \mid \searrow \mid \nu_{p,v} \mid \mathsf{rd}_{p,v} \mid \mathsf{wr}_{p,v} \mid \mu \mid \overset{\frown}{\ell} \mid b \\ b := \mathsf{spw}_e \mid \overset{\smile}{\ell} \end{array}$$

[2] By enriching the conflict relation, see below.

where v and e are closed. The action μ stands for taking a lock that is already held. We denote by $\mathcal{A}ct$ 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 $(e\,[])$, 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. Then an occurrence is a sequence o over the set $\mathcal{SO}cc$ below:

$$\mathcal{O}cc = \{([]\,\_), (\bullet\,[]), (\mathsf{if}\ []\ \mathsf{then}\ \_\ \mathsf{else}\ \_), (\mathsf{ref}\,[]), (!\,[]), ([] := \_), (v := []), ([]\backslash\ell)\}$$
$$\mathcal{SO}cc = \mathcal{O}cc \cup \{(\_\,[]), (\lambda x[]\,\_), (\mathsf{if}\ \_\ \mathsf{then}\ []\ \mathsf{else}\ \_), (\mathsf{if}\ \_\ \mathsf{then}\ \_\ \mathsf{else}\ []), (\_ := [])\}$$

The occurrences $o \in \mathcal{O}cc^*$ are called *normal*. Notice that we do not consider $\lambda x[]$ 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 $@\mathbf{\Sigma}$, as the sequence of frames that points to the hole in $\mathbf{\Sigma}$, that is:

$$@[] = \varepsilon$$
$$@\mathbf{\Sigma}[\mathbf{\Phi}] = @\mathbf{\Sigma} \cdot @\mathbf{\Phi}$$

where

$$@([]\,e) = ([]\,\_)$$
$$@(e\,[]) = \begin{cases} (\bullet\,[]) \text{ if } e \in \mathcal{V}al \\ (\_\,[]) \text{ otherwise} \end{cases}$$
$$@(e := []) = \begin{cases} (v := []) \text{ if } e = v \in \mathcal{V}al \\ (\_ := []) \text{ otherwise} \end{cases}$$

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 $\mathbf{\Sigma}$, like with future computations or branch prediction. On the other hand, the

$$\mathbf{\Sigma}[(\lambda x e v)] \xrightarrow[@\mathbf{\Sigma}]{\beta} \mathbf{\Sigma}[\{x \mapsto v\}e] \qquad \mathbf{\Sigma}[(p := v)] \xrightarrow[@\mathbf{\Sigma}]{\mathsf{wr}_{p,v}} \mathbf{\Sigma}[()]$$

$$\mathbf{\Sigma}[(\mathsf{if}\ tt\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1)] \xrightarrow[@\mathbf{\Sigma}]{\swarrow} \mathbf{\Sigma}[e_0] \qquad \mathbf{E}[(\mathsf{thread}\, e)] \xrightarrow[@\mathbf{E}]{\mathsf{spw}_e} \mathbf{E}[()]$$

$$\mathbf{\Sigma}[(\mathsf{if}\ ff\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1)] \xrightarrow[@\mathbf{\Sigma}]{\searrow} \mathbf{\Sigma}[e_1] \qquad \mathbf{\Sigma}[(\mathsf{with}\ \ell\ \mathsf{do}\ e)] \xrightarrow[@\mathbf{\Sigma}]{\mu} \mathbf{\Sigma}[e] \qquad \ell \in \lceil\mathbf{\Sigma}\rceil$$

$$\mathbf{\Sigma}[(\mathsf{ref}\, v)] \xrightarrow[@\mathbf{\Sigma}]{\nu_{p,v}} \mathbf{\Sigma}[p] \qquad \mathbf{\Sigma}[(\mathsf{with}\ \ell\ \mathsf{do}\ e)] \xrightarrow[@\mathbf{\Sigma}]{\overset{\frown}{\ell}} \mathbf{\Sigma}[(e\backslash\ell)] \quad \ell \notin \lceil\mathbf{\Sigma}\rceil$$

$$\mathbf{\Sigma}[(!\, p)] \xrightarrow[@\mathbf{\Sigma}]{\mathsf{rd}_{p,v}} \mathbf{\Sigma}[v] \qquad \mathbf{E}[(v\backslash\ell)] \xrightarrow[@\mathbf{E}]{\overset{\frown}{\ell}} \mathbf{E}[v]$$

Fig. 1. Speculations

value resulting from a dereference operation $(!\, p)$, or the status of the lock in the case of a locking construct $(\mathsf{with}\ \ell\ \mathsf{do}\ e)$, is "guessed", or "predicted" – as regards loads from the memory, this is known as *value prediction*, and was introduced in [11,19]. 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 $([]\, e)$ and $(\lambda x e\, [])$ – hence in particular $(\mathsf{let}\ x = []\ \mathsf{in}\ e)$ and $[]\,;e$ –, where one is allowed to speculate the execution of e; this is also the case with $(\mathsf{if}\ []\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1)$ where one can speculate in a branch, that is in e_0 or e_1. Then for instance with an expression of the form $(e_0\backslash\ell)\,;e_1$, 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 [3]). 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 $\sigma = \left(e_i \xrightarrow[o_i]{a_i} e_{i+1}\right)_{0 \leqslant i \leqslant n}$ 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

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

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

$$(\mathsf{if}\ !p\ \mathsf{then}\ q := tt\ \mathsf{else}\ ()) \xrightarrow{\mathsf{wr}_{q,tt}} (\mathsf{if}\ !p\ \mathsf{then}\ ()\ \mathsf{else}\ ()) \xrightarrow{\mathsf{rd}_{p,tt}} (\mathsf{if}\ tt\ \mathsf{then}\ ()\ \mathsf{else}\ ()) \xrightarrow{\swarrow} ()$$

Here we speculate by predicting that we will have to compute in the first branch, while guessing that the value pointed to by p is tt. Obviously this guessed value may not be the correct one, and in this case the computation made in the "then" branch has to be invalidated. We shall define valid speculations in the next section.

The concurrent speculative semantics is again a small step semantics, consisting in transitions between *configurations* $C = (S, L, T)$ where the store S, also called here the memory, is a mapping from a finite set $\mathsf{dom}(S)$ of references to values, the lock context L is a finite set of locks, those that are currently held by some thread, and T, the thread system, is a mapping from a finite set $\mathsf{dom}(T)$ of thread names (or thread identifiers), subset of $\mathcal{N}ames$, to expressions. If $\mathsf{dom}(T) = \{t_1, \ldots, t_n\}$ and $T(t_i) = e_i$ we also write T as

$$(t_1, e_1) \parallel \cdots \parallel (t_n, e_n)$$

As usual, we shall assume we consider only *well-formed* configurations, meaning that any reference that occurs somewhere in the configuration belongs to the domain of the store, that is, it is bound to a value in the memory – we shall not define this property, which is preserved in the operational semantics, more formally. For instance, if e is an expression of the source language, any initial configuration $(\emptyset, \emptyset, (t, e))$ is well-formed. The speculative computations are made of transitions that have the form

$$C \xrightarrow[t,o]{a} C'$$

indicating the action a that is performed, the thread t that performs it, and the occurrence o where it is performed in the thread (these labels are just annotations, introduced for technical convenience, but they do not entail any constraint on the semantics). At each step, a speculation attempted by one thread is recorded, provided that *the global state agrees* with the action that is issued. That is, the value guessed by a thread for a pointer must be the value of that pointer in the memory (but notice that the store itself is speculative, being speculatively updated), and similarly acquiring a lock can only be done if the lock is free. We distinguish two cases, depending on whether the action spawns a new thread or not. The corresponding two rules are given in Figure 2, where

$$(*)\ \begin{cases} a = \beta, \swarrow, \searrow, \mu \Rightarrow S' = S\ \&\ L' = L \\ a = \nu_{p,v} \Rightarrow p \notin \mathsf{dom}(S)\ \&\ S' = S \cup \{p \mapsto v\} \\ \qquad\qquad \&\ L' = L \\ a = \mathsf{rd}_{p,v} \Rightarrow v = S(p)\ \&\ S' = S\ \&\ L' = L \\ a = \mathsf{wr}_{p,v} \Rightarrow S' = S[p := v]\ \&\ L' = L \\ a = \overset{\frown}{\ell} \Rightarrow S' = S\ \&\ \ell \notin L\ \&\ L' = L \cup \{\ell\} \\ a = \overset{\frown}{\ell} \Rightarrow S' = S\ \&\ L' = L - \{\ell\} \end{cases}$$

$$\frac{e \xrightarrow[o]{a} e' \qquad a \neq \mathsf{spw}_{e''}}{(S, L, (t, e) \parallel T) \xrightarrow[t,o]{a} (S', L', (t, e') \parallel T)} \ (*)$$

$$\frac{e \xrightarrow[o]{\mathsf{spw}_{e'}} e''}{(S, L, (t, e) \parallel T) \xrightarrow[t,o]{\mathsf{spw}_{e'}} (S, L, (t, e'') \parallel (t', e') \parallel T)} \ t' \notin \mathsf{dom}(T) \cup \{t\}$$

Fig. 2. Speculative Computations

Definition (Computations) 2.6. *A* speculative computation *from a configuration C to a configuration C' is a (possibly empty) sequence γ of steps $\left(C_i \xrightarrow[t_i,o_i]{a_i} C_{i+1}\right)_{0 \leqslant i \leqslant n}$ in the speculative operational semantics such that $C_0 = C$ and $C_n = C'$. This is written $\gamma : C \xrightarrow{*} C'$. The empty computation is denoted ε. The* concatenation *$\gamma \cdot \gamma' : C \xrightarrow{*} C'$ is only defined (in the obvious way) if γ ends on the configuration C'' where γ' originates, that is $\gamma : C \xrightarrow{*} C''$ and $\gamma' : C'' \xrightarrow{*} C'$. The computation $\gamma = \left(C_i \xrightarrow[t_i,o_i]{a_i} C_{i+1}\right)_{0 \leqslant i \leqslant n}$ is* normal *if for all i the occurrence o_i is normal.*

One can see that normal computations correspond to computations in the standard *interleaving semantics*, that we regard as the reference semantics from the programmer's point of view. Even though our definition of speculative computations ensures that the values read from the memory are correctly guessed, some speculation sequences are still wrong, like – omitting the occurrences

$$(\{p \mapsto \mathit{ff}\}, \emptyset, (t, (\mathsf{if}\ !p\ \mathsf{then}\ p := \mathit{tt}\ \mathsf{else}\ ()))) \xrightarrow{\mathsf{wr}_{p,\mathit{tt}}} (\{p \mapsto \mathit{tt}\}, \emptyset, (t, (\mathsf{if}\ !p\ \mathsf{then}\ ()\ \mathsf{else}\ ()))) \xrightarrow{\mathsf{rd}_{p,\mathit{tt}}} (\{p \mapsto \mathit{tt}\}, \emptyset, (t, (\mathsf{if}\ \mathit{tt}\ \mathsf{then}\ ()\ \mathsf{else}\ ())))$$

Here the normal *data dependency* between the read and write on p is broken, and the branch prediction is therefore wrong. Similarly, the computation

$$(\{p \mapsto \mathit{ff}\}, \emptyset, (t, (\mathsf{if}\ \mathit{ff}\ \mathsf{then}\ p := \mathit{tt}\ \mathsf{else}\ ()))) \xrightarrow{\mathsf{wr}_{p,\mathit{tt}}} (\{p \mapsto \mathit{tt}\}, \emptyset, (t, (\mathsf{if}\ \mathit{ff}\ \mathsf{then}\ ()\ \mathsf{else}\ ()))) \xrightarrow{\mathsf{rd}_{p,\mathit{tt}}} (\{p \mapsto \mathit{tt}\}, \emptyset, (t, ()))$$

is wrong because it violates the normal *control dependency* between the predicate and the branches of the conditional branching. In the next section we shall define which are the correct speculative computations. To this end, we shall need the following technical definition, which formalizes the contribution of each thread to a speculative computation:

Definition (Projection) 2.7. *Given a thread identifier t, the* projection *$\gamma|_t$ of a speculative computation γ on thread t is defined as follows, by induction on γ:*

$$\varepsilon|_t = \varepsilon$$

$$(C \xrightarrow[t',o]{a} C' \cdot \gamma)|_t = \begin{cases} e \xrightarrow[o]{a} e' \cdot (\gamma|_t) \text{ if } t' = t \;\&\; \\ \qquad\qquad C = (S, L, (t, e) \parallel T) \;\&\; \\ \qquad\qquad C' = (S', L', (t, e') \parallel T) \\ \gamma|_t \qquad\qquad \text{otherwise} \end{cases}$$

It is easy to check that this is indeed well-defined, that is:

Remark 2.8. *For any speculative computation γ and name t, the projection $\gamma|_t$ is a speculation.*

3 Valid Speculations

We shall qualify a speculative computation as *valid* in the case where each of its projections is *equivalent* in some sense to a normal evaluation. That is, a speculative computation is valid if it only involves thread speculations that correctly guess the values read from the memory, and preserves, up to some equivalence, the normal program order. In other words, the validity criterion is local to the threads, namely, each thread's speculation should be equivalent to a sequential execution.[3] The equivalence we use is the *permutation of transitions equivalence* introduced by Berry and Lévy [5,18], that we also used in our previous work on memory models [7]. Intuitively, this equivalence says that permuting independent steps in a speculation results in "the same" speculation, and that such independent steps could actually be performed in parallel. It is clear, for instance, that actions performed at disjoint occurrences can be done in any order, provided that they are not conflicting accesses to the same memory location (the conflict relation will be defined below). This applies for instance to

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

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

$$(\mathsf{if}\ tt\ \mathsf{then}\ q := tt\ \mathsf{else}\ ()) \xrightarrow{\mathsf{wr}_{q,tt}} (\mathsf{if}\ tt\ \mathsf{then}\ ()\ \mathsf{else}\ ()) \xrightarrow{\checkmark} ()$$

(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

$$(\mathsf{if}\ tt\ \mathsf{then}\ q := tt\ \mathsf{else}\ ()) \xrightarrow{\checkmark} q := tt \xrightarrow{\mathsf{wr}_{q,tt}} ()$$

To this end, given a speculation step $e \xrightarrow[o]{a} e'$ 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

$$(\mathsf{if}\ tt\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1) \xrightarrow[\varepsilon]{\checkmark} e_0$$

then for $o' = \varepsilon$ or $o' = (\mathsf{if}\ []\ \mathsf{then}\ \_\ \mathsf{else}\ \_)$ there is not residual, because the occurrence has been consumed in reducing the expression. The residual of any

[3] This appears to be the standard – though implicit – validity criterion in the literature on speculative execution of sequential programs.

occurrence pointing into e_0, i.e. of the form $(\mathsf{if}\ \_\ \mathsf{then}\ [\,]\ \mathsf{else}\ \_) \cdot o'$, is o', whereas an occurrence of the form $(\mathsf{if}\ \_\ \mathsf{then}\ \_\ \mathsf{else}\ [\,]) \cdot o'$, pointing into e_1, has no residual, since the subexpression e_1 is discarded by reducing to the first branch of the conditional expression. This is the way we deal with control dependencies. The notion of a residual here is much simpler than in the λ-calculus (see [18]), 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:

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

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. For lack of space, the proof of this Lemma is omitted (as well as the proofs of other statements below).

Lemma (Diamond Lemma) 3.1. *If* $e \xrightarrow[o_0]{a_0} e_0$ *and* $e \xrightarrow[o_1]{a_1} e_1$ *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* $e_0 \xrightarrow[o_1']{a_1} e'$ *and* $e_1 \xrightarrow[o_o']{a_0} e'$.

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:

$$r := !\,p\,; q := tt \xrightarrow[(\lambda x[\,]\,\_)]{\mathsf{wr}_{q,tt}} r := !\,p\,;() $$

and

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

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

$$e \xrightarrow[o_0]{a_0} e_0 \xrightarrow[o_1']{a_1} e' \quad \text{and} \quad e \xrightarrow[o_1]{a_1} e_1 \xrightarrow[o_0']{a_0} e'$$

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:

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

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 follows[4]:

Definition (Conflicting Actions) 3.2. *The* conflict *relation* $\#$ *between actions is given by*

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

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* $e \xrightarrow[o_0]{a_0} e_0$ *and* $e \xrightarrow[o_1]{a_1} e_1$ *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*

$$\sigma_0 \cdot e \xrightarrow[o_0]{a_0} e_0 \xrightarrow[o_1']{a_1} e' \cdot \sigma_1 \simeq \sigma_0 \cdot e \xrightarrow[o_1]{a_1} e_1 \xrightarrow[o_0']{a_0} e' \cdot \sigma_1$$

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

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

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

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

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.*

[4] We notice that in some (extremely, or even excessively) relaxed memory model (such as the one of the Alpha architecture, see [21]) 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).

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

$$(\mathsf{if}\ !p\ \mathsf{then}\ ()\ \mathsf{else}\ q := tt) \xrightarrow{\mathsf{wr}_{q,tt}} (\mathsf{if}\ !p\ \mathsf{then}\ ()\ \mathsf{else}\ ()) \xrightarrow{\mathsf{rd}_{p,tt}} (\mathsf{if}\ tt\ \mathsf{then}\ ()\ \mathsf{else}\ ()) \xrightarrow{\checkmark} ()$$

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 by permuted with it. We have already seen that the speculations from Examples 2.4 and 2.5 are valid. (Notice that, obviously, any normal computation is 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 **R→RW**, according to the terminology of [1], that was not captured in our previous work [7] 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 [7], 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 amd6 example (see [25]), that is

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

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

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

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 [2,12], 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 [6] 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 Robustness

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

Definition (Speculatively DRF Program) 4.1. *A configuration C* has a speculative data race *iff there exist t_i, o_i, a_i and C_i ($i = 0, 1$) such that $C' \xrightarrow[t_0, o_0]{a_0} C_0$ and $C' \xrightarrow[t_1, o_1]{a_1} C_1$ with $t_0 \neq t_1$ & $a_0 \# a_1$. A valid speculative computation $\left(C_i \xrightarrow[t_i, o_i]{a_i} C_{i+1}\right)_{0 \leqslant i \leqslant n}$ is* speculatively date race free *iff for all i, C_i has no speculative data race. A configuration C is* speculatively date race free *(speculatively DRF, or SDRF) iff any valid speculative computation originating in C is data race free. 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, in the sense that if C is speculatively DRF and C' is reachable from C by a normal computation, 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 [21]) 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 $p := \mathit{ff}\,; r := \mathbf{!}\,p$. We do not follow this way here, since such a model requires unnatural synchronizations from the programmer.

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

$$o/\varepsilon \equiv o$$

$$o/(e \xrightarrow[o']{a} e') \cdot \sigma \equiv (o/(a, o'))/\sigma$$

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 [5,18], and of steps being in the same *family*, a property introduced in [5].

DEFINITION (STEPS) 4.2. *A* step *is a pair* $[\sigma, (a, o)]$ *of a speculation* $\sigma : e \xrightarrow{*} e'$ *and an action* a *at occurrence* o *such that* $e' \xrightarrow[o]{a} e''$ *for some expression* e''. *Given a speculation* σ, *the set* $\mathsf{Step}(\sigma)$ *is the set of steps* $[\varsigma, (a, o)]$ *such that* $\varsigma \cdot \xrightarrow[o]{a} \leq \sigma$. *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 4.3. *If* $[\varsigma, (a, o)] \in \mathsf{Step}(\sigma)$ *and* $\sigma' \simeq \sigma$ *then there exists* $[\varsigma', (a, o')] \in \mathsf{Step}(\sigma')$ *such that* $[\varsigma, (a, o)] \sim [\varsigma', (a, o')]$.

Proof: by induction on the definition of $\sigma' \simeq \sigma$. ⬓

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 4.4. *Let* $\sigma = \sigma_0 \cdot \xrightarrow[o]{a} \cdot \sigma_1$ *be such that* $\sigma \simeq \xrightarrow[o']{a} \cdot \varsigma$ *with* $[\varepsilon, (a, o')] \sim [\sigma_0, (a, o)]$. *If* $\sigma_0 = \varsigma_0 \cdot (e \xrightarrow[\bar{o}]{\bar{a}} e') \cdot \varsigma_1$ *then there exist* o'', e'', $\bar{o'}$ *and* σ_1' *such that* $\varsigma_0 \cdot (e \xrightarrow[o'']{a} e'' \xrightarrow[\bar{o'}]{\bar{a}} \bar{e}) \cdot \sigma_1' \simeq \sigma$ *where* $o \equiv o''/\xrightarrow[\bar{o'}]{\bar{a}} \cdot \sigma_1'$ *and* $\bar{o'} \equiv \bar{o}/(a, o'')$.

Next, we can 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 4.5. *Let* $\sigma = \sigma_0 \cdot \xrightarrow[o]{a} \cdot \sigma_1$ *where* $a \in \mathcal{B}$, *and* $\sigma \simeq \sigma'$ *with* $\sigma' = \sigma_0' \cdot \xrightarrow[o]{a} \cdot \sigma_1'$ *where* $[\sigma_0, (a, o)] \sim [\sigma_0', (a, o)]$. *If* $[\varsigma, (a', o')] \in \mathsf{Step}(\sigma_0)$ *with* $o' \in \mathcal{O}cc^*$ *then there exist* ς' *and* o'' *such that* $[\varsigma', (a', o'')] \in \mathsf{Step}(\sigma_0')$ *and* $[\varsigma, (a', o')] \sim [\varsigma', (a', o'')]$.

An immediate consequence of this property is:

COROLLARY 4.6. *If* σ *is a valid speculation, that is* $\sigma \simeq \bar{\sigma}$ *for some normal speculation* $\bar{\sigma}$, *and if* $\bar{\sigma} = \bar{\sigma_0} \cdot \xrightarrow[o]{a} \cdot \bar{\sigma_1}$ *with* $a \in \mathcal{B}$, *then* $\sigma = \sigma_0 \cdot \xrightarrow[o]{a} \cdot \sigma_1$ *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.

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

Definition (Regular Configuration) 4.7. *A configuration $C = (S, L, T)$ is* regular *if and only if it satisfies*

(i) *if* $T = (t_i, \mathbf{\Sigma}_i[(e_i \backslash \ell)]) \parallel T_i$ *for* $i = 0, 1$ *then* $t_0 = t_1$ & $\mathbf{\Sigma}_0 = \mathbf{\Sigma}_1$ & $e_0 = e_1$ & $T_0 = T_1$

(ii) $T = (t, \mathbf{\Sigma}[(e \backslash \ell)]) \parallel T' \;\Rightarrow\; \ell \in L$

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

Remark 4.8. *If C is regular and $C \xrightarrow[t,o]{a} C'$ then C' is regular.*

The following lemma (for a different notion of computation) was called the "Asynchrony Lemma" in [7]. 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 4.9. *Let C be a (well-formed) regular configuration. If $C \xrightarrow[t_0,o_0]{a_0} C_0 \xrightarrow[t_1,o_1]{a_1} C'$ with $t_0 \neq t_1$, $\neg(a_0 \# a_1)$ and $a_0 = \overset{\curvearrowleft}{\ell} \;\Rightarrow\; a_1 \neq \overset{\curvearrowright}{\ell}$, then there exists C_1 such that $C \xrightarrow[t_1,o_1]{a_1} C_1 \xrightarrow[t_0,o_0]{a_0} C'$.*

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

Lemma 4.10. *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*

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

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.

Proposition 4.11. *Let C be a well-formed, closed, regular configuration. If $\gamma : C \xrightarrow{*} C'$ is a valid data race free speculative computation, then there exists a normal computation $\bar{\gamma}$ from C to C'.*

Proof: by induction on the length of γ. This is trivial if $\gamma = \varepsilon$. Otherwise, let $\gamma = \left(C_i \xrightarrow[t_i,o_i]{a_i} C_{i+1}\right)_{0 \leqslant i \leqslant n}$ with $n > 0$. Notice that for any i, the configuration C_i is well-formed, regular and has no data race. 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 ($0 \leqslant j < n$) 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 γ 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 $\gamma|_{t_i} = \varsigma_0 \cdot \xrightarrow[o_i]{a_i} \cdot \varsigma_1$, and by Lemma 4.3 we have $\sigma^{t_i} = \bar{\varsigma_0} \cdot \xrightarrow[o']{a_i} \cdot \bar{\varsigma_1}$ with $[\varsigma_0, (o_i, a_i)] \sim [\bar{\varsigma_0}, (o', a_i)]$. Then by Corollary 4.6 the first step of $\bar{\varsigma_0} \cdot \xrightarrow[o']{a_i}$ is in the family of a step in $\varsigma_0 \cdot \xrightarrow[o_i]{a_i}$, contradicting the minimality of j. We therefore have $a_{j-1} \neq \overset{\curvearrowleft}{\ell}$ in particular. By Lemma 4.9 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 $\sigma_0 = \varsigma_0 \cdot \xrightarrow[o_{j-1}]{a_{j-1}}$, and by Lemma 4.4 there exist o', o'' and σ_1' such that $\gamma|_{t_j} \simeq \varsigma_0 \cdot \xrightarrow[o']{a_j} \cdot \xrightarrow[o'']{a_{j-1}} \cdot \sigma_1'$ with $o \equiv o'/(a_{j-1}, o_{j-1})$. We conclude using Lemma 4.10 and the induction hypothesis (on j). ❏

Notice that we proved a property that is actually more precise than stated in the proposition, 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. An immediate consequence of this property is the announced robustness result:

Theorem (Robustness) 4.12. *Any speculatively data race free closed expression is robust.*

We 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 could have considered simpler memory barrier operations than the mutual exclusion construct (with ℓ do e), such as fence. This is a programming constant (but not a value), the semantics of which is given by

$$\mathbf{E}[\mathsf{fence}] \to \mathbf{E}[()]$$

with no side effect. Performing a fence should be categorized as a $\mathcal{B}$ action, so that the Corollary 4.6 holds for such an action, since it is only performed from within a normal evaluation context. Then our Theorem 4.12, which, as far as the $\mathcal{B}$ actions are concerned, relies on this property 4.6, 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 [6]), that is

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

starting from a configuration where the memory S is such that $S(p) = ff = S(q)$ we may, as with the amd6 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 $S'(r_1) = ff = S'(r_3)$. 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 the fence, thus checking that local normal order evaluations with the same actions is possible.

5 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, which are beyond the scope of our theory of speculations). 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 programming styles for safe multithreading. Our model could also be made less permissive, either by resticting the class of speculation contexts, or by extending the conflict relation, to forbid some commutations (regarding synchronization actions in particular), in order to capture more precisely actual optimized execution platforms. Obviously, our robustness result still holds, but in some cases one could hope to get a more liberal robustness property, like the DRF guarantee for instance. We plan to explore the variety of such speculation scenarios in future work.

References

1. Adve, S.A., Gharachorloo, K.: Shared memory consistency models: a tutorial. IEEE Computer 29(12), 66–76 (1996)
2. Adve, S., Hill, M.D.: Weak ordering – A new definition. In: ISCA 1990, pp. 2–14 (1990)
3. Aspinall, D., Ševčík, J.: Java memory model examples: good, bad and ugly. In: VAMP 2007 (2007)
4. Ševčík, J., Aspinall, D.: On validity of program transformations in the JAVA memory model. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 27–51. Springer, Heidelberg (2008)
5. Berry, G., Lévy, J.-J.: Minimal and optimal computations of recursive programs. J. of ACM 26, 148–175 (1979)

6. Boehm, H.-J., Adve, S.V.: Foundations of the C++ concurrency model. In: PLDI 2008, pp. 68–78 (2008)
7. Boudol, G., Petri, G.: Relaxed memory models: an operational approach. In: POPL 2009, pp. 392–403 (2009)
8. Burton, F.W.: Speculative computation, parallelism, and functional programming. IEEE Trans. on Computers C-34(12), 1190–1193 (1985)
9. Felleisen, M., Friedman, D.P.: Control operators, the SECD-machine and the λ-calculus. In: Wirsing, M. (ed.) Formal Description of Programming Concepts III, pp. 193–217. Elsevier, Amsterdam (1986)
10. Flanagan, C., Felleisen, M.: The semantics of future and its use in program optimization. In: POPL 1995, pp. 209–220 (1995)
11. Gabbay, F., Mendelson, A.: Using value prediction to increase the power of speculative execution hardware. ACM Trans. on Computer Systems 16(3), 234–270 (1998)
12. Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., Hennessy, J.: Memory consistency and event ordering in scalable sharedmemory multiprocessors. ACM SIGARCH Computer Architecture News 18(3a), 15–26 (1990)
13. Gosling, J., Joy, B., Steele, G.L., Bracha, G.: The JAVA Language Specification, 3rd edn. Prentice Hall, Englewood Cliffs (2005)
14. Halstead, R.H.: Multilisp: a language for concurrent symbolic computation. ACM TOPLAS 7(4), 501–538 (1985)
15. Katz, M., Weise, D.: Continuing into the future: on the interaction of futures and first-class continuations. In: ACM Conf. on Lisp and Functional Programming, pp. 176–184 (1990)
16. Knight, T.: An architecture for mostly functional languages. In: ACM Conf. on Lisp and Functional Programming, pp. 105–112 (1986)
17. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. on Computers 28(9), 690–691 (1979)
18. Lévy, J.-J.: Optimal reductions in the lambda calculus. In: To, H.B.C., Seldin, J.P., Hindley, J.R. (eds.) Essays on Combinatory Logic, Lambda Calculus and Formalism, pp. 159–191. Academic Press, London (1980)
19. Lipasti, M.H., Wilkerson, C.B., Shen, J.P.: Value locality and load value prediction. In: ASPLOS 1996, pp. 138–147 (1996)
20. Manson, J., Pugh, W., Adve, S.A.: The Java memory model. In: POPL 2005, pp. 378–391 (2005)
21. Martin, M.K., Sorin, D.J., Cain, H.W., Hill, M.D., Lipasti, M.H.: Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing. In: 34th International Symp. on Microarchitecture, pp. 328–337 (2001)
22. Moreau, L.: The semantics of Scheme with futures. In: ICFP 1996, pp. 146–156 (1996)
23. Navabi, A., Jagannathan, S.: Exceptionally safe futures. In: Field, J., Vasconcelos, V.T. (eds.) COORDINATION 2009. LNCS, vol. 5521, pp. 47–65. Springer, Heidelberg (2009)
24. Saraswat, V., Jagadeesan, R., Michael, M., von Praun, C.: A theory of memory-models. In: PPoPP 2007, pp. 161–172 (2007)
25. Sarkar, S., Sewell, P., Zappa Nardelli, F., Owens, S., Ridge, T., Braibant, T., Myreen, M.O., Alglave, J.: The semantics of x86-CC multiprocessor machine code. In: POPL 2009, pp. 379–391 (2009)
26. Smith, J.E.: A study of branch prediction strategies. In: ISCA 1981, pp. 135–148 (1981)
27. Welc, A., Jagannathan, S., Hosking, A.: Safe futures for JAVA. In: OOPSLA 2005, pp. 439–453 (2005)