A Deadlock-Free Semantics for Shared Memory Concurrency*

Gérard Boudol

INRIA, 06902 Sophia Antipolis, France

Abstract. We design a deadlock-free semantics for a concurrent, functional and imperative programming language where locks are implicitly and univocally associated with pointers. The semantics avoids unsafe states by relying on a static analysis of programs, by means of a type and effect system. The system uses singleton reference types, which allow us to have a precise information about the pointers that are anticipated to be locked by an expression.

1 Introduction

In this paper we revisit, from a programming language perspective, one of the most annoying problems with concurrent systems, namely the risk of entering into a deadlocked situation. Deadlocks arise in particular from synchronization mechanisms like locking, when several threads of computation are circularly blocked, each waiting for a resource that is locked by another thread. As is well-known, locking is sometimes necessary. To illustrate this, as well as some other points, we shall use an example which is often considered as regards synchronization problems. This is the example of manipulating bank accounts. In our setting, a bank account will simply be a memory location containing an integer value.[1] Now suppose that we want to define a function to deposit some amount x on the account y. Using ML's notation $(!\, y)$ to get the contents of the memory location y (i.e. to dereference it, in ML's jargon where memory locations are called references – we shall also use the word "pointer"), this function can be defined as $\lambda x \lambda y(y := !\, y + x)$. There is a problem however with this definition, which is that two concurrent deposits may have the effect of only one of them, if both read the current amount before it has been updated by the other thread.

To solve this problem, it is enough to make the deposit function taking, for the update operation $y := !\, y + x$, an exclusive access to the bank account to update, that is y. In this paper we shall assume that there is in the programming language a construct, say $(\mathsf{lock}\ y\ \mathsf{in}\ e)$, to lock the reference y for the purpose of performing the operation e with an exclusive access to y. Indeed, we think that the programmer should be offered constructs to control access to memory locations (as they appear in the language), rather than having to explicitly

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

[1] The operations we shall consider dealing with accounts should actually be packaged into a module.

M. Leucker and C. Morgan (Eds.): ICTAC 2009, LNCS 5684, pp. 140–154, 2009.

manipulate locks. In other words, we are assuming here that the locks are, transparently for the programmer, associated with the resources, as in JAVA. Then we can conveniently define the deposit function as follows:

$$\mathsf{deposit} \;=\; \lambda x \lambda y(\mathsf{lock}\ y\ \mathsf{in}\ y := !\, y + x)$$

Similarly, we can define a function to withdraw some amount from an account:

$$\mathsf{withdraw} \;=\; \lambda x \lambda y(\mathsf{lock}\ y\ \mathsf{in}\ (\mathsf{if}\ !\, y \geq x\ \mathsf{then}\ (y := !\, y - x)\ \mathsf{else}\ \mathsf{error}))$$

From this we can define another function, to transfer some amount x from an account y to another one z, as $\lambda x \lambda y \lambda z((\mathsf{withdraw}\, xy)\,;(\mathsf{deposit}\, xz))$. It has been argued (see [9]) that this function should ensure the property that another thread cannot see the intermediate state where y has decreased, but z has not yet been credited. This can be achieved by defining

$$\mathsf{transfer} \;=\; \lambda x \lambda y \lambda z(\mathsf{lock}\ y\ \mathsf{in}\ (\mathsf{withdraw}\, xy)\,;(\mathsf{deposit}\, xz))$$

We are assuming here that the locks are reentrant: a thread that temporarily "possesses" a reference, like y in this example, is not blocked in locking it twice. Now suppose that two transfers are performed concurrently, from account a to account b, and in the converse direction. That is, we have to execute something like

$$(\mathsf{transfer}\, 100\, a\, b) \;\|\; (\mathsf{transfer}\, 10\, b\, a) \tag{1}$$

Clearly there is a danger of deadlock here: if both operations first perform the withdrawals, locking respectively a and b, they are then blocked in trying to lock the other account in order to perform the deposits.

There are three ways out of deadlocks, that have been identified long ago in the area of operating systems development (see [2]):

(i) deadlock *prevention* aims at only accepting for execution concurrent systems that are determined to be deadlock-free, in the sense that none of their interleaved executions runs into a deadlock;
(ii) deadlock *avoidance* aims at ensuring, by monitoring the execution at runtime, that unsafe states that could lead to a deadlocked situation are avoided;
(iii) deadlock *detection* and *recovery* uses run-time monitoring and rollback mechanisms to analyse the current state, and undo some computations[2] in case there is a deadlock.

Despite the existence of the well-known Dijkstra's Banker's algorithm, solutions (i) and (iii) are, by far, the most popular. Deadlock detection and recovery is similar to optimistic concurrency control in database transactions implementation. By contrast, deadlock avoidance may be qualified as pessimistic concurrency. (See [7] for a recent use of this technique).

Solution (i), deadlock prevention, lends itself to using static analysis techniques. Indeed, a lot of work has been done in this direction – see [1,3,5,10,11], to mention just a few recent works on this topic. One has to notice that, with

[2] Provided these are not irrevocable, such as I/O operations.

no exception, all these works (also including [7]) use the standard approach to precluding deadlocks, which is to assume an ordering on locking to prevent circularities. This is an assumption we would like to avoid: in our bank account example, where one can do concurrent transfers from an account to another in any direction, like in Example (1), such an assumption would entail that there should be a unique lock associated with all the accounts, which obviously limits the concurrency in a drastic, and sometimes unjustified way. In this paper we shall explore a different direction, namely (ii), deadlock avoidance.

To implement solution (ii), one has to know in advance what are the resources that are needed, in an exclusive way, by a thread. Then this also seems amenable to static analysis techniques. This is what this paper is proposing: we define, for a standard multithreaded programming style, a *type and effect system* that allows us to design a *prudent* semantics, that is then proved to be deadlock-free. The idea is quite simple: one should not lock a pointer whenever one anticipates, by typing, to take some other pointer that is currently held by another thread. As one can see, this is much lighter than implementing optimistic concurrency, and the proof of correctness is not very complicated. Surprisingly enough, I could not find in the literature any reference to a similar work – except [12], which however uses a Petri net model, and Discrete Control Theory –, so ours appears to be the first one to define a deadlock-free semantics, following the deadlock avoidance approach, based on a type and effect system for standard multithreading.

To conclude this introduction, let us discuss some more technical points of our contribution. In analysing an expression such as $(\mathsf{lock}\ e_0\ \mathsf{in}\ e_1)$, we need a way to get, statically, an approximative idea of what will be the value of e_0, the pointer to be locked, in order to assign it as an effect to the locking expression, and then use it in the types. An idea could be to use dependent types, but dependent types for imperative, call-by-value languages is a topic which largely remains unexplored, and the existing proposals (see [8] for instance) seem to be over-elaborate for our purpose. A standard approach to statically get information about pointer accesses is to use *regions* in a type and effect system [6]: in an ML-like language, one assigns (distinct) region names to the subexpressions $(\mathsf{ref}\ e)$ creating a reference, and one can then record as an effect the region where a reference that has to be locked resides. In this way, locks are actually associated with regions, rather than with references. However, this is too coarse grained for our purpose: again using the bank account example, we could define a (very simplified) function for creating accounts with an initial value as $\lambda x(\mathsf{ref}\ x)$, but then, this would mean that all accounts would be assigned the same lock, and we already rejected such a scenario.

To solve this problem, we shall introduce in the programming language a new construct $(\mathsf{cref}\ e)$ which is a function that, when applied to some (dummy) argument, then creates a reference with initial value the one of e. Typically, $(\mathsf{cref}\ e)()$ has the same meaning as $(\mathsf{ref}\ e)$. We shall then restrict, by typing, the use of such a function f to a particular form, namely $(\mathsf{let}\ x = (f())\ \mathsf{in}\ e)$ where e does not export x. In this way, we shall be able to know exactly the name of the

pointer denoted by e_0 in $(\mathsf{lock}\ e_0\ \mathsf{in}\ e_1)$[3], using *singleton types* [4], which are both dependent types of a very simple kind, as well as types with (singleton) regions. This provides us with a fine grained locking policy, where locks are univocally associated with references.

Note. For lack of space, the proofs are omitted, or only sketched.

2 Source and Target Languages

Our source language is an extension of CoreML, that is a functional (embedding the call-by-value λ-calculus) and imperative language, enriched with concurrent programming primitives, namely a thread spawning construct $(\mathsf{thread}\, e)$ and a locking construct $(\mathsf{lock}\ e_0\ \mathsf{in}\ e_1)$. The main feature of ML we are interested in here is not the polymorphic let, but rather the explicit distinction between values and references to values. Typically, in ML – as opposed to SCHEME or JAVA for instance –, one cannot write $x := x+1$, because x cannot be both an integer, as in $x+1$, and a reference to an integer, as in $x := 1$. As explained in the Introduction, we refine the reference creation construct $(\mathsf{ref}\, e)$ of ML into $(\mathsf{cref}\, e)$, which is a function that needs to be applied (to a dummy argument) to actually create a mutable reference, with the value of e as initial value. Then $(\mathsf{ref}\, e)$ is here an abbreviation for $((\mathsf{cref}\, e)())$. For simplicity, we omit from the language the constructs relying on basic types such as the booleans or integers. Considering these constructs (and recursion) does not cause any technical difficulty, and we shall use them in the examples. The syntax of our source language is as follows:

$$
\begin{array}{rcll}
v,\ w \ldots & ::= & x \mid \lambda x e & \textit{values} \\
e & ::= & v \mid (e_1 e_0) & \textit{expressions (functional)} \\
& & \mid (\mathsf{cref}\, e) \mid (\mathsf{!}\, e) \mid (e_0 := e_1) & \textit{(imperative)} \\
& & \mid (\mathsf{thread}\, e) \mid (\mathsf{lock}\ e_0\ \mathsf{in}\ e_1) & \textit{(concurrent)}
\end{array}
$$

The abstraction $\lambda x e$ is the only binder in this language. We denote by $\{x \mapsto v\}e$ the capture-avoiding substitution of the variable x by the value v in its free occurrences in e, and we shall always consider expressions up to α-conversion, that is up to the renaming of bound variables. We shall use the standard abbreviation $(\mathsf{let}\ x = e_0\ \mathsf{in}\ e_1)$ for $(\lambda x e_1 e_0)$, also denoted $e_0\,;e_1$ whenever x is not free in e_1. The use of expressions e reducing to values of the form $(\mathsf{cref}\, v)$ will be restricted, by typing, to a particular form, namely $(\mathsf{let}\ x = (e())\ \mathsf{in}\ e')$. A particular case of this is $(\mathsf{let}\ x = (\mathsf{ref}\, e)\ \mathsf{in}\ e')$.

In order to be evaluated (or executed), the expressions of the source language will be first translated into a slightly different language. This run-time language, or more appropriately *target language* differs from the source one on the following points:

(i) the construct $(\mathsf{cref}\, e)$ is removed, as well as the values $(\mathsf{cref}\, v)$;

[3] This does not mean that one can statically predict which pointers will be created at run-time, since an expression such as $(\mathsf{let}\ x = ((\mathsf{cref}\, e)())\ \mathsf{in}\ e')$ can be passed as an argument, and duplicated.

(ii) *references* (or pointers), ranged over by $p,\ q\ldots$ are introduced. These are run-time values;
(iii) the locking construct $(\mathsf{lock}\ e_0\ \mathsf{in}\ e_1)$ is replaced by the family of constructs $(\mathsf{lock}_\varphi\ e_0\ \mathsf{in}\ e_1)$ where φ is any *effect*, that is any finite set of pointer names (either constant or variable);
(iv) a family of constructs $(e\backslash p)_{\psi,P}$ is introduced, to represent the fact that the pointer p is currently held, and will be released upon termination of e. In this construct ψ and P are finite sets of pointers (they are there for technical convenience only);
(v) a construct $(\mathsf{new}\ x\ \mathsf{in}\ e)$, also written simply $\nu x\, e$, is introduced for creating new pointers. This is a binder for x.

An expression $(\mathsf{cref}\, e)$ of the source language will be represented as

$$(\mathsf{let}\ x = e\ \mathsf{in}\ \lambda y((y := x)\,;y))$$

in the target language, where it will take (and return) a pointer as argument (see the next Section). The (pointer) variables occurring in the effect φ in $(\mathsf{lock}_\varphi\ e_0\ \mathsf{in}\ e_1)$ are free in this expression. An expression of the target language is called *pure* if it does not contain any pointer (which does not mean that its evaluation does not produce side effects). In particular, a pure expression does not contain any subexpression of the form $(e\backslash p)_{\psi,P}$.

3 Translation

In this section we define a translation, guided by a type and effect system, from the source language into the target language. The purpose of this translation is twofold:

(i) we compute the effect φ of an expression e, which is the set of pointers that this expression may have to lock during its execution. This effect is then used to annotate, by translating them, the expressions $(\mathsf{lock}\ e'\ \mathsf{in}\ e)$ (which are also the ones which produce an effect, namely of locking the pointer denoted by e'), in order to guide the evaluation, avoiding deadlocks. This is the main purpose of the type and effect system.
(ii) we restrict the use of expressions of the form $(\mathsf{cref}\, e)$ in a way that allows us to have, in the types, a precise information about the pointer names.

The types for the source language are as follows:

$$\tau,\ \sigma,\ \theta\ldots\ ::=\ \mathsf{unit}\ \mid\ \theta\,\mathsf{ref}_x\ \mid\ \theta\,\mathsf{cref}\ \mid\ (\tau \xrightarrow{\varphi} \sigma)$$

Here $\theta\,\mathsf{ref}_x$ is a *singleton* type [4], meaning that the only value of this type is the pointer name x. (This is a very primitive form of dependent type.) We abbreviate $(\tau \xrightarrow{\emptyset} \sigma)$ into $(\tau \to \sigma)$. In $(\theta\,\mathsf{ref}_x \xrightarrow{\varphi} \sigma)$ the (pointer) variable x is *universally quantified*, with scope φ and σ, and will be instantiated when applying a function of this type. The capture-avoiding substitution $\{x \mapsto y\}\tau$ is defined in the standard way, and we always consider types up to α-conversion,

$$\frac{}{\Gamma, x:\tau \vdash_s x : \emptyset, \tau \;\Rightarrow\; x} \qquad \frac{\Gamma, x:\tau \vdash_s e : \varphi, \sigma \;\Rightarrow\; \bar{e}}{\Gamma \vdash_s \lambda x e : \emptyset, (\tau \xrightarrow{\varphi} \sigma) \;\Rightarrow\; \lambda x \bar{e}} \qquad \frac{}{\Gamma \vdash_s () : \emptyset, \mathsf{unit} \;\Rightarrow\; ()}$$

$$\frac{\Gamma \vdash_s e_0 : \varphi_0, (\tau \xrightarrow{\varphi_2} \sigma) \;\Rightarrow\; \overline{e_0} \qquad \Gamma \vdash_s e_1 : \varphi_1, \tau \;\Rightarrow\; \overline{e_1}}{\Gamma \vdash_s (e_0 e_1) : \varphi_0 \cup \varphi_1 \cup \varphi_2, \sigma \;\Rightarrow\; (\overline{e_0}\,\overline{e_1})} \; \tau \neq \theta\,\mathsf{ref}_x$$

$$\frac{\Gamma \vdash_s e_0 : \varphi_0, (\theta\,\mathsf{ref}_x \xrightarrow{\varphi_2} \sigma) \;\Rightarrow\; \overline{e_0} \qquad \Gamma \vdash_s e_1 : \varphi_1, \theta\,\mathsf{ref}_y \;\Rightarrow\; \overline{e_1}}{\Gamma \vdash_s (e_0 e_1) : \varphi_0 \cup \varphi_1 \cup \{x \mapsto y\}\varphi_2, \{x \mapsto y\}\sigma \;\Rightarrow\; (\overline{e_0}\,\overline{e_1})}$$

$$\frac{\Gamma \vdash_s e_0 : \varphi_0, \theta\,\mathsf{cref} \;\Rightarrow\; \overline{e_0} \qquad \Gamma, x : \theta\,\mathsf{ref}_x \vdash_s e_1 : \varphi_1, \tau \;\Rightarrow\; \overline{e_1}}{\Gamma \vdash_s (\lambda x e_1 (e_0 ())) : \varphi_0 \cup (\varphi_1 - \{x\}), \tau \;\Rightarrow\; \nu y\, (\lambda x \overline{e_1} (\overline{e_0} y))} \; y \text{ fresh},\; x \notin \Gamma, \varphi_0, \tau$$

$$\frac{\Gamma \vdash_s e : \varphi, \theta \;\Rightarrow\; \bar{e}}{\Gamma \vdash_s (\mathsf{cref}\, e) : \varphi, \theta\,\mathsf{cref} \;\Rightarrow\; (\lambda x \lambda y ((y := x)\,; y)\bar{e})}$$

$$\frac{\Gamma \vdash_s e : \varphi, \theta\,\mathsf{ref}_x \;\Rightarrow\; \bar{e}}{\Gamma \vdash_s (!\, e) : \varphi, \theta \;\Rightarrow\; (!\, \bar{e})} \qquad \frac{\Gamma \vdash_s e_0 : \varphi_0, \theta\,\mathsf{ref}_x \;\Rightarrow\; \overline{e_0} \quad \Gamma \vdash_s e_1 : \varphi_1, \theta \;\Rightarrow\; \overline{e_1}}{\Gamma \vdash_s (e_0 := e_1) : \varphi_0 \cup \varphi_1, \mathsf{unit} \;\Rightarrow\; (\overline{e_0} := \overline{e_1})}$$

$$\frac{\Gamma \vdash_s e : \varphi, \mathsf{unit} \;\Rightarrow\; \bar{e}}{\Gamma \vdash_s (\mathsf{thread}\, e) : \emptyset, \mathsf{unit} \;\Rightarrow\; (\mathsf{thread}\, \bar{e})}$$

$$\frac{\Gamma \vdash_s e_0 : \varphi_0, \theta\,\mathsf{ref}_x \;\Rightarrow\; \overline{e_0} \qquad \Gamma \vdash_s e_1 : \varphi_1, \tau \;\Rightarrow\; \overline{e_1}}{\Gamma \vdash_s (\mathsf{lock}\; e_0\; \mathsf{in}\; e_1) : \{x\} \cup \varphi_0 \cup \varphi_1, \tau \;\Rightarrow\; (\mathsf{lock}_{\varphi_1}\; \overline{e_0}\; \mathsf{in}\; \overline{e_1})}$$

Figure 1: Type and Effect System (Source Language)

that is, up to the renaming of bound variables. The typing judgements for the source language are as follows:

$$\Gamma \vdash_s e : \varphi, \tau$$

where Γ is a typing context, that is a mapping from a finite set $\mathsf{dom}(\Gamma)$ of variables to types. In this judgement the effect φ is the set of pointer names that the expression e may have to lock during its evaluation. In the following we shall only consider *well-formed* judgements, meaning that if a type $\theta\,\mathsf{ref}_x$ occurs in the judgement then x does not occur in θ, and if $\Gamma(y) = \theta\,\mathsf{ref}_x$ then $y = x$. This assumption is left implicit in the following.

We shall give a simultaneous definition for both the type and effect system and the translation from the source to the target languages. That is, we define inductively the predicate

$$\Gamma \vdash_s e : \varphi, \tau \;\Rightarrow\; \bar{e}$$

meaning that the source expression e is well-typed in the typing context Γ, with effect φ and type τ, and translates into the target expression $\bar{e}$. The rules are given in Figure 1, where, when we write $x \notin \Gamma, \varphi, \tau$, we mean that x does not occur in Γ (neither in the domain, nor in the types assigned by this typing context), nor in φ, nor in τ. By forgetting the "$\Rightarrow \bar{e}$" parts one obtains the rules of the

type system for the source language. One should notice that if $\Gamma \vdash_s e : \varphi, \tau \Rightarrow \bar{e}$ then $\bar{e}$ is pure. One should also notice that the type and effect system only *builds* effects, but does not (other than by implicit type unification, as usual) use them to constrain the typing.

The most interesting rule is the one for the $(\mathsf{lock}\ e_0\ \mathsf{in}\ e_1)$ constructs. This expression is the only one introducing an effect, which is the name of the pointer that is intended to be locked, that is the reference resulting from the evaluation of e_0. In the translation of this expression, that is $(\mathsf{lock}_{\varphi_1}\ \overline{e_0}\ \mathsf{in}\ \overline{e_1})$, one records the anticipated effect φ_1 of e_1. Indeed, the operational semantics will rely on the idea that, in order to avoid deadlocks, one should not lock the pointer which is denoted by $\overline{e_0}$ if a pointer from φ_1 is already held by another thread. Notice that the use of a singleton type for e_0, namely $\theta\,\mathsf{ref}_x$, allows us to build the effect as a set of names (i.e. variables, in the source language), and not expressions (or regions).

As announced, reference creation is restricted to the form $(\mathsf{let}\ x = (e_0()) \ \mathsf{in}\ e_1)$, where the name (that is, x) of the reference in known in e_1. In the translation of this expression, namely $\nu y(\mathsf{let}\ x = (\overline{e_0}y)\ \mathsf{in}\ \overline{e_1})$, one first creates, by means of νy, a fresh pointer name (see the following Section), which is passed as an argument, and then bound to the value v "handled" by $\overline{e_0}$ (as one can see, $\overline{e_0}$ is constrained, by typing, to reduce to an expression of the form $\lambda z((z := v)\,;\,z)$). By reduction the name y will be substituted for x in $\overline{e_1}$, and in particular in the effects involving the name x, in subexpressions of the form $(\mathsf{lock}_\varphi\ \_\ \mathsf{in}\ \_)$.

One can see that, assuming that we have the obvious typing rules for boolean and integer constructs, the deposit and transfer functions considered in the Introduction can be typed as follows, using polymorphic types, where y and z are universally quantified:

$$\Gamma \vdash_s \mathsf{deposit} : \emptyset, \mathsf{int} \to (\mathsf{int}\,\mathsf{ref}_y \xrightarrow{\{y\}} \mathsf{unit})$$
$$\Gamma \vdash_s \mathsf{transfer} : \emptyset, \mathsf{int} \to (\mathsf{int}\,\mathsf{ref}_y \to (\mathsf{int}\,\mathsf{ref}_z \xrightarrow{\{y,z\}} \mathsf{unit}))$$

and their definitions are translated as follows:

$$\lambda x \lambda y(\mathsf{lock}_\emptyset\ y\ \mathsf{in}\ y := !\,y + x)$$
$$\lambda x \lambda y \lambda z(\mathsf{lock}_{\{y,z\}}\ y\ \mathsf{in}\ (\mathsf{withdraw}\,xy)\,;(\mathsf{deposit}\,xz))$$

Then (assuming that error has any type) one can check that the following is typable, in a context where the functions deposit, withdraw and transfer have been defined, as above:

$$\begin{array}{l}\mathsf{let\ create\_account}\ = \lambda x(\mathsf{cref}\,x)\ \mathsf{in}\\ \quad \mathsf{let}\ a = (\mathsf{create\_account}\ 100)()\ \mathsf{in}\\ \quad \mathsf{let}\ b = (\mathsf{create\_account}\ 10)()\ \mathsf{in}\\ \qquad ((\mathsf{thread}\,(\mathsf{transfer}\,50\,ab)))\,;(\mathsf{deposit}\,10\,b)\end{array} \tag{2}$$

and the translation is, with some optimization in the translation of $\mathsf{create\_account}$:

$$\begin{array}{l} \mathsf{let\ create\_account}\ = \lambda x \lambda y((y := x)\,;y)\ \mathsf{in} \\ \mathsf{new}\ y\ \mathsf{in\ let}\ a = (\mathsf{create\_account}\ 100)y\ \mathsf{in} \\ \mathsf{new}\ z\ \mathsf{in\ let}\ b = (\mathsf{create\_account}\ 10)z\ \mathsf{in} \\ \qquad ((\mathsf{thread}\,(\mathsf{transfer}\,50\,ab)))\ ;(\mathsf{deposit}\,10\,b) \end{array}$$

4 Prudent Operational Semantics

As usual, evaluation consists in reducing a *redex* (reducible expression) in an *evaluation context*, possibly performing a side effect. In our (run-time) language, redexes and evaluation contexts are defined as follows:

$$\begin{array}{lll} r ::= (\lambda xev) \mid (!\,p) \mid (p := v) & & \textit{redexes} \\ \quad \mid (\mathsf{thread}\,e) \mid (\mathsf{lock}_\psi\ p\ \mathsf{in}\ e) \mid (v\backslash p)_{\psi,P} \mid \nu x\,e & & \\ \mathbf{E} ::= [] \mid \mathbf{E}[\mathbf{F}] & & \textit{evaluation contexts} \\ \mathbf{F} := ([]\,e) \mid (v\,[]) & & \textit{frames} \\ \quad \mid (\mathsf{cref}\,[]) \mid (!\,[]) \mid ([] := e) \mid (v := []) & & \\ \quad \mid (\mathsf{lock}_\psi\ []\ \mathsf{in}\ e) \mid ([]\backslash p)_{\psi,P} & & \end{array}$$

To define the semantics of reentrant locks, we shall use the set $\lceil \mathbf{E} \rceil$ of pointers held in the context $\mathbf{E}$, computed by a kind of "stack inspection" mechanism, as follows:

$$\begin{array}{r} \lceil [] \rceil = \emptyset \\ \lceil \mathbf{E}[\mathbf{F}] \rceil = \lceil \mathbf{E} \rceil \cup \lceil \mathbf{F} \rceil \end{array} \quad \text{where} \quad \lceil \mathbf{F} \rceil \; = \begin{cases} \{p\} \text{ if } \mathbf{F} = ([]\backslash p)_{\psi,P} \\ \emptyset \quad \text{otherwise} \end{cases}$$

We now describe our operational semantics for expressions of the target language, defined as a small-step transition system between *configurations* (S, L, T) where S is the *store*, that is a partial mapping from a finite set $\mathsf{dom}(S)$ of pointers to values, L is a finite set of *locked pointers*, and T is a multiset of *threads*, which are simply expressions. The store is only partial because in some state, some pointers may have been created but not yet initialized. As regards the store, we shall use the following notations: $S + p$, where $p \notin \mathsf{dom}(S)$, is the store obtained by adding p to $\mathsf{dom}(S)$, but not providing a value for p; $S[p := v]$, where p is supposed to be in $\mathsf{dom}(S)$, is the store obtained by initializing or updating the value of p to be v. The set L is the set of pointers that are currently held by some thread. As regards multisets, our notations are as follows. Given a set X, a *multiset* over X is a mapping E from X to the set $\mathbb{N}$ of non-negative integers, indicating the multiplicity $E(x)$ of an element. We denote by x the singleton multiset such that $x(y) = (\mathsf{if}\ y = x\ \mathsf{then}\ 1\ \mathsf{else}\ 0)$. Multiset union $E \parallel E'$ is given by $(E \parallel E')(x) = E(x) + E'(x)$. In the following we only consider multisets of expressions, ranged over by T.

The semantics is given in Figure 2, that we now comment. The general form of the rules is

$$(S, L, \mathbf{E}[r] \parallel T) \to (S', L', \mathbf{E}[e] \parallel T')$$

$$
\begin{array}{rll}
(S, L, \mathbf{E}[(\lambda x e v)] \parallel T) & \to (S, L, \mathbf{E}[\{x \mapsto v\}e] \parallel T) & \\
(S, L, \mathbf{E}[(!\, p)] \parallel T) & \to (S, L, \mathbf{E}[v] \parallel T) & S(p) = v \\
(S, L, \mathbf{E}[(p := v)] \parallel T) & \to (S[p := v], L, \mathbf{E}[()] \parallel T) & \\
(S, L, \mathbf{E}[(\mathsf{thread}\, e)] \parallel T) & \to (S, L, \mathbf{E}[()] \parallel T \parallel e) & \\
(S, L, \mathbf{E}[(\mathsf{lock}_\psi\ p\ \mathsf{in}\ e)] \parallel T) & \to (S, L, \mathbf{E}[e] \parallel T) & p \in \lceil \mathbf{E} \rceil \\
(S, L, \mathbf{E}[(\mathsf{lock}_\psi\ p\ \mathsf{in}\ e)] \parallel T) & \to (S, L', \mathbf{E}[(e \backslash p)_{\psi,P}] \parallel T) & p \notin \lceil \mathbf{E} \rceil\ \&\ (\spadesuit)\ \& \\
 & & P = \mathsf{dom}(S) \\
(S, L, \mathbf{E}[(v \backslash p)_{\psi,P}] \parallel T) & \to (S, L - \{p\}, \mathbf{E}[v] \parallel T) & \\
(S, L, \mathbf{E}[\nu x\, e] \parallel T) & \to (S + p, L, \mathbf{E}[\{x \mapsto p\}e] \parallel T) & p \notin \mathsf{dom}(S)
\end{array}
$$

$(\spadesuit)\quad L \cap (\{p\} \cup (\psi - \lceil \mathbf{E} \rceil)) = \emptyset,\ L' = L \cup \{p\}$

Figure 2: Prudent Operational Semantics

meaning that any thread ready to be reduced can be non-deterministically chosen for evaluation. Again, the most interesting case is the one of expressions $(\mathsf{lock}_\psi\ e_0\ \mathsf{in}\ e_1)$. To evaluate such an expression, one first has to evaluate e_0, since $(\mathsf{lock}_\psi\ []\ \mathsf{in}\ e_1)$ is [part of] an evaluation context. The expected result is a pointer p. Then, to reduce $(\mathsf{lock}_\psi\ p\ \mathsf{in}\ e_1)$, one first looks in the evaluation context $\mathbf{E}$ to see if the thread has already locked p, that is $p \in \lceil \mathbf{E} \rceil$. If this is the case, the locking instruction is ignored, that is $(\mathsf{lock}_\psi\ p\ \mathsf{in}\ e_1)$ is reduced to e_1, with no effect. Otherwise, one consults the set L to see if p, *or any pointer in* ψ, is locked by another thread. If this is the case, the expression $(\mathsf{lock}_\psi\ p\ \mathsf{in}\ e_1)$ is blocked, waiting for this condition to become false. Otherwise, the pointer p is locked,[4] and one proceeds executing e_1 in a context where the fact that p is currently held is recorded, namely $([]\backslash p)_{\psi,P}$. On termination of e_1, the pointer p is released. One should compare the precondition in $(\spadesuit)$ for taking a lock with the usual one, which is $L \cap \{p\} = \emptyset$. It is then obvious that our prudent semantics avoids some paths explored in the standard interleaving semantics.

Notice that the sets ψ and P in the context $([]\backslash p)_{\psi,P}$ are actually not used in the operational semantics, and could therefore be removed from the syntax. We include them for the sole purpose of proving our safety result. Here ψ is the set of pointers that are anticipated, by the $(\mathsf{lock}_\psi\ p\ \mathsf{in}\ e_1)$ instruction, as possibly locked in the future, before p is released. The set P is the one of known pointers at the time where p is locked.

In the following we shall only consider *well-formed* configurations, which are triples (S, L, T) such that if a pointer p occurs in the configuration, either in some thread or in some value in the store, or in L, then $p \in \mathsf{dom}(S)$. It is easy to check that well-formedness is preserved by reduction, since references are allocated in the store when they are created.

[4] The computations expressed by $(\spadesuit)$ must be performed in an atomic way. This means that in an implementation one would use a global lock on the set L.

In the rest of this section we establish some results about the operational semantics, and discuss it on an example. Let us say that a configuration (S, L, T) is *regular* if it satisfies

(i) $T = \mathbf{E}[(e\backslash p)_{\psi,P}] \parallel T' \;\Rightarrow\; p \notin \lceil \mathbf{E} \rceil$ & $(T' = \mathbf{E}'[e'] \parallel T'' \;\Rightarrow\; p \notin \lceil \mathbf{E}' \rceil)$

(ii) $p \in L \;\Leftrightarrow\; \exists \mathbf{E}, e, \psi, P, T'.\ T = \mathbf{E}[(e\backslash p)_{\psi,P}] \parallel T'$

Clearly, if e is a pure expression, the initial configuration $(\emptyset, \emptyset, e)$ is regular. Moreover, this property is preserved by reduction:

Lemma 4.1. *If (S, L, T) is regular and $(S, L, T) \to (S', L', T')$ then (S', L', T') is regular.*

The following notion of a *safe* expression is central to our safety result:

Definition (Safe Expression) 4.2. *A closed pure expression e of the target language is* safe *if $(\emptyset, \emptyset, e) \xrightarrow{*} (S, L, T)$ implies*

$$T = \mathbf{E}[(\mathbf{E}'[(\mathsf{lock}_{\psi_1}\ p_1\ \mathsf{in}\ e)]\backslash p_0)_{\psi_0,P_0}] \parallel T' \;\&\; p_1 \in P_0 \;\Rightarrow\; p_1 \in \psi_0$$

That is, when a reference p_1 is about to be locked while some other pointer p_0 was previously locked by the same thread, with p_1 known to exist at that point, then the possibility of locking p_1 was anticipated when locking p_0. (this is where we need ψ and P in $(e\backslash p)_{\psi,P}$).

Definition (Deadlock) 4.3. *A configuration (S, L, T) is* deadlocked *if*

$$T = \mathbf{E}_0[(\mathsf{lock}_{\psi_0}\ p_0\ \mathsf{in}\ e_1)] \parallel \cdots \parallel \mathbf{E}_n[(\mathsf{lock}_{\psi_n}\ p_n\ \mathsf{in}\ e_n)] \parallel T'$$

with $n > 0$ and $p_{i+1} \in \lceil \mathbf{E}_i \rceil$ (mod $n+1$). A pure expression e is deadlock-free *if no configuration reachable from $(\emptyset, \emptyset, e)$ is deadlocked.*

The main property of our operational semantics is the following:

Proposition 4.4. *Any safe expression is deadlock-free.*

Proof Sketch: let us assume the contrary, that is $(\emptyset, \emptyset, e) \xrightarrow{*} (S, L, T)$ where (S, L, T) is deadlocked. For simplicity, let us assume that there are two threads in T that block each other, that is

$$T = \mathbf{E}_0^0[(\mathbf{E}_1^0[(\mathsf{lock}_{\varphi_0}\ p_1\ \mathsf{in}\ e_0)]\backslash p_0)_{\psi_0,P_0}] \parallel \mathbf{E}_0^1[(\mathbf{E}_1^1[(\mathsf{lock}_{\varphi_1}\ p_0\ \mathsf{in}\ e_1)]\backslash p_1)_{\psi_1,P_1}] \parallel T'$$

(the general case where there is a cycle of blocked threads of length greater than 2 is just notationally more cumbersome). Since e is safe, we have $p_1 \in \psi_0$ if $p_1 \in P_0$, and $p_0 \in \psi_1$ if $p_0 \in P_1$. Assume for instance that p_0 is the pointer that is locked the first (and then not released), that is:

$$\begin{aligned}
(\emptyset, \emptyset, e) &\xrightarrow{*} (S_0, L_0, \mathbf{E}_0^0[(\mathsf{lock}_{\psi_0}\ p_0\ \mathsf{in}\ e_0')] \parallel T_0) \\
&\to (S_0, L_0', \mathbf{E}_0^0[(e_0'\backslash p_0)_{\psi_0,P_0}] \parallel T_0) && P_0 = \mathsf{dom}(S_0) \\
&\xrightarrow{*} (S_1, L_1, \mathbf{E}_0^0[(e_0''\backslash p_0)_{\psi_0,P_0}] \parallel \mathbf{E}_0^1[(\mathsf{lock}_{\psi_1}\ p_1\ \mathsf{in}\ e_1')] \parallel T_1) \\
&\to (S_1, L_1', \mathbf{E}_0^0[(e_0''\backslash p_0)_{\psi_0,P_0}] \parallel \mathbf{E}_0^1[(e_1'\backslash p_1)_{\psi_1,P_1}] \parallel T_1) && P_1 = \mathsf{dom}(S_1) \\
&\xrightarrow{*} (S, L, T)
\end{aligned}$$

Since e is pure, the configurations reachable from $(\emptyset, \emptyset, e)$ are regular (Lemma 4.1), and therefore $p_0 \in L_1 \subseteq P_1$, but then reducing $(\mathsf{lock}_{\psi_1}\ p_1\ \mathsf{in}\ e_1')$ in the context of L_1 is not possible – a contradiction. ⬭

To conclude this section, let us revisit and discuss Example (2), where the multiset of threads is

$$(\mathsf{transfer}\,50\,a\,b)\,\|(\mathsf{deposit}\,10\,b)$$

Assuming that the pointers a and b contain some integers in the store, with $S(a) \geq 50$, and both of them are free (i.e. not locked), one can see that a reachable state is $(S, \{a\}, ((a := !\,a - 50)\backslash a)\,\|(\mathsf{deposit}\,10\,b))$, where we omit the ψ and P components annotating the context $(\_\backslash a)$. Then, from this state one can reach for instance the state

$$(S, \{a, b\}, ((a := !\,a - 50)\backslash a)\,\|((b := !\,b + 10)\backslash b))$$

This means that there is some real concurrency in executing a transfer from a to b and a deposit to b in parallel, even though both these operations need to lock b at some point. However, if $(\mathsf{deposit}\,10\,b)$ starts executing, this blocks $(\mathsf{transfer}\,50\,a\,b)$, because the latter cannot lock a, while anticipating to lock b, since b is already locked. Then the condition (♠) is sometimes too strong in preventing deadlocks, precluding some harmless interleavings, and one may wonder how we could relax it, adopting for instance a more informative structure than L for locked pointers. However, one must be careful with pointer creation, as the following example shows:

$$\mathsf{new}\ x\ \mathsf{in}\ \mathsf{lock}_\emptyset\ x\ \mathsf{in}\ \mathsf{new}\ y\ \mathsf{in}\ (\mathsf{thread}\,(\mathsf{lock}_{\{x\}}\ y\ \mathsf{in}\ (\mathsf{lock}_\emptyset\ x\ \mathsf{in}\ ()))); \\ (\mathsf{lock}_\emptyset\ y\ \mathsf{in}\ ())$$

Starting with $S = \emptyset = L$, this expression reduces to

$$(\{p \mapsto , q \mapsto\}, \{p\}, ((\mathsf{lock}_\emptyset\ q\ \mathsf{in}\ ())\backslash p)_{\emptyset,\{p\}}\,\|(\mathsf{lock}_{\{p\}}\ q\ \mathsf{in}\ (\mathsf{lock}_\emptyset\ p\ \mathsf{in}\ ())))$$

where the second thread is (as it should be) not allowed to lock q. Notice that to detect a potential cycle out of the static information contained in this expression one has to look into the evaluation context.

5 Safety

In this section we establish our main result (Type Safety, Theorem 5.8 below), stating that typable expressions are safe. The types for the target language are as follows:

$$\rho ::= x \mid p \qquad \textit{pointer names}$$
$$\tau,\ \sigma,\ \theta \ldots ::= \mathsf{unit} \mid \theta\,\mathsf{ref}_\rho \mid (\tau \xrightarrow{\varphi} \sigma) \qquad \textit{types}$$

We define a translation $\tau \Rightarrow \overline{\tau}$ from the types of the source language to the types of the target language by

$$\theta\,\mathsf{cref} \ \Rightarrow\ (\theta\,\mathsf{ref}_x \to \theta\,\mathsf{ref}_x)$$

$$\frac{}{\Gamma, x : \tau \vdash_t x : \emptyset, \tau} \qquad \frac{}{\Gamma, p : \theta\,\mathsf{ref}_p \vdash_t p : \emptyset, \theta\,\mathsf{ref}_p} \qquad \frac{\Gamma, x : \tau \vdash_t e : \varphi, \sigma}{\Gamma \vdash_t \lambda x e : \emptyset, (\tau \xrightarrow{\varphi} \sigma)}\ x \notin \Gamma$$

$$\frac{}{\Gamma \vdash_t () : \emptyset, \mathsf{unit}} \qquad \frac{\Gamma \vdash_t e_0 : \varphi_0, (\tau \xrightarrow{\varphi_2} \sigma) \qquad \Gamma \vdash_t e_1 : \varphi_1, \tau}{\Gamma \vdash_t (e_0 e_1) : \varphi_0 \cup \varphi_1 \cup \varphi_2, \sigma}\ \tau \neq \theta\,\mathsf{ref}_x$$

$$\frac{\Gamma \vdash_t e_0 : \varphi_0, (\theta\,\mathsf{ref}_x \xrightarrow{\varphi_2} \sigma) \qquad \Gamma \vdash_t e_1 : \varphi_1, \theta\,\mathsf{ref}_\rho}{\Gamma \vdash_t (e_0 e_1) : \varphi_0 \cup \varphi_1 \cup \{x \mapsto \rho\}\varphi_2, \{x \mapsto \rho\}\sigma}$$

$$\frac{\Gamma \vdash_t e : \varphi, \theta\,\mathsf{ref}_\rho}{\Gamma \vdash_t (!\, e) : \varphi, \theta} \qquad \frac{\Gamma \vdash_t e_0 : \varphi_0, \theta\,\mathsf{ref}_\rho \quad \Gamma \vdash_t e_1 : \varphi_1, \theta}{\Gamma \vdash_t (e_0 := e_1) : \varphi_0 \cup \varphi_1, \mathsf{unit}}$$

$$\frac{\Gamma \vdash_t e : \varphi, \mathsf{unit}}{\Gamma \vdash_t (\mathsf{thread}\, e) : \emptyset, \mathsf{unit}} \qquad \frac{\Gamma \vdash_t e_0 : \varphi_0, \theta\,\mathsf{ref}_\rho \quad \Gamma \vdash_t e_1 : \varphi_1, \tau}{\Gamma \vdash_t (\mathsf{lock}_{\varphi_1}\ e_0\ \mathsf{in}\ e_1) : \{\rho\} \cup \varphi_0 \cup \varphi_1, \tau}$$

$$\frac{\Gamma \vdash_t e : \varphi, \tau}{\Gamma \vdash_t (e \backslash p)_{\psi,P} : \varphi, \tau}\ \varphi \cap P \subseteq \psi,\ P \subseteq \mathsf{dom}(\Gamma) \qquad \frac{\Gamma, x : \theta\,\mathsf{ref}_x \vdash_t e : \varphi, \tau}{\Gamma \vdash_t \nu x\, e : \varphi - \{x\}, \tau}\ x \notin \Gamma, \tau$$

Figure 3: Type and Effect System (Target Language)

(where x is not in θ). The judgements of the type system for the target language are $\Gamma \vdash_t e : \varphi, \tau$ where Γ, the typing context, is a mapping from a finite set $\mathsf{dom}(\Gamma)$ of variables and pointers to types.

As in the case of the source language, we only consider *well-formed* judgements, meaning that if a type $\theta\,\mathsf{ref}_\rho$ occurs in the judgement then ρ does not occur in θ, and if $\Gamma(\rho') = \theta\,\mathsf{ref}_\rho$ then $\rho' = \rho$. The typing rules are given in Figure 3. These are essentially the same as for the source language, with some new rules. One should in particular notice the constraints on the typing of $(e\backslash p)_{\psi,P}$: the anticipated effect of e must be recorded, as far as the known pointers are concerned, in the ψ component. This is the condition that will ensure the safety of typable expressions. First, we wish to show that the translation from the source to the target language preserves typability. To this end, we need a standard weakening property:

LEMMA (WEAKENING) 5.1. *If $\Gamma \vdash_t e : \varphi, \tau$ and x and p do not occur in this judgement then $\Gamma, x : \sigma \vdash_t e : \varphi, \tau$ and $\Gamma, p : \theta \vdash_t e : \varphi, \tau$.*

Then we have, denoting by $\overline{\Gamma}$ the typing context obtained from Γ by translating the types assigned to the variables:

LEMMA 5.2. *If $\Gamma \vdash_s e : \varphi, \tau \ \Rightarrow\ \bar{e}$ then $\overline{\Gamma} \vdash_t \bar{e} : \varphi, \overline{\tau}$.*

PROOF SKETCH: by induction on the definition of $\Gamma \vdash_s e : \varphi, \tau \ \Rightarrow\ \bar{e}$. The only cases to consider are the rule for $(\lambda x e_1(e_0()))$ with e_0 of type $\theta\,\mathsf{cref}$, using the Lemma 5.1, and the rules for $(\mathsf{cref}\, e)$ and $e = (\mathsf{lock}\ e_0\ \mathsf{in}\ e_1)$. ❏

Some obvious properties are:

REMARK 5.3.

(i) *If* $\Gamma \vdash_t v : \varphi, \tau$ *then* $\varphi = \emptyset$, *and if* $\tau = \theta\,\mathsf{ref}_\rho$ *then* $v = \rho$.

(ii) *If* $\Gamma \vdash_t e : \varphi, \tau$ *and* x *is free in* e *then* $x \in \mathsf{dom}(\Gamma)$.

(iii) *If* $\Gamma \vdash_t e : \varphi, \tau$ *and* p *occurs in* e *then* $p \in \mathsf{dom}(\Gamma)$.

LEMMA (STRENGTHENING) 5.4.

(i) *If* $\Gamma, x : \tau \vdash_t e : \varphi, \tau$ *and* x *is not free in* e *then* $\Gamma \vdash_t e : \varphi, \sigma$.

(ii) *If* $\Gamma, p : \sigma \vdash_t e : \varphi, \tau$ *and* p *does not occur in* e *then* $\Gamma \vdash_t e : \varphi, \tau$.

Our type safety result is established following the standard steps (see [13]), that is, the main property to show is that typability is preserved by reduction (the so-called "Subject Reduction" property). To this end, we need a lemma regarding typing and substitution, and another one regarding the typing of expressions of the form $\mathbf{E}[e]$ (the "Replacement Lemma"). We denote by $\{x \mapsto \rho\}(\Gamma \vdash_t e : \varphi, \tau)$ the substitution of x by ρ in all its free occurrences in this judgement. This is only defined if $x \in \mathsf{dom}(\Gamma)$ & $\rho \in \mathsf{dom}(\Gamma) \;\Rightarrow\; \Gamma(x) = \Gamma(\rho)$.

LEMMA (SUBSTITUTION) 5.5.

(i) *If* $\Gamma \vdash_t e : \varphi, \tau$ *and* $\rho' \in \mathsf{dom}(\Gamma) \;\Rightarrow\; \Gamma(\rho') = \theta\,\mathsf{ref}_{\rho'}$ *for* $\rho' \in \{x, \rho\}$ *then* $\{x \mapsto \rho\}(\Gamma \vdash_t e : \varphi, \tau)$.

(ii) *If* $x \notin \Gamma$ *then* $\Gamma, x : \sigma \vdash_t e : \varphi, \tau$ & $\Gamma \vdash_t v : \emptyset, \sigma \;\Rightarrow\; \Gamma \vdash_t \{x \mapsto v\}(e : \varphi, \tau)$.

PROOF SKETCH:

(i) The proof, by induction on the inference of $\Gamma \vdash_t e : \varphi, \tau$, is straightforward.

(ii) This is a standard property, established by induction on the inference of $\Gamma, x : \sigma \vdash_t e : \varphi, \tau$ (using the Weakening Lemma 5.1, and the previous point). In the case where $e = (\mathsf{lock}_{\varphi_1}\ e_0\ \mathsf{in}\ e_1)$ with $\Gamma, x : \sigma \vdash_t e_0 : \varphi_0, \theta\,\mathsf{ref}_x$ and $\Gamma, x : \sigma \vdash_t e_1 : \varphi_1, \tau$, we have $\sigma = \theta\,\mathsf{ref}_x$ by the well-formedness assumption, and we use Remark 5.3(i), that is $v = x$. ❑

LEMMA (REPLACEMENT) 5.6. *If* $\Gamma \vdash_t \mathbf{E}[e] : \varphi, \tau$ *then there exist* ψ *and* σ *such that* $\Gamma \vdash_t e : \psi, \sigma$ *and if* $\Gamma' \vdash_t e' : \psi', \sigma$ *with* $\Gamma \subseteq \Gamma'$ *and* $\psi' \cap \mathsf{dom}(\Gamma) \subseteq \psi$ *then there exists* φ' *such that* $\Gamma' \vdash_t \mathbf{E}[e'] : \varphi', \tau$ *with* $\varphi' \cap \mathsf{dom}(\Gamma) \subseteq \varphi$.

PROOF SKETCH: by induction on the evaluation context, and then by case on the frame $\mathbf{F}$ such that $\mathbf{E} = \mathbf{E}'[\mathbf{F}]$. We only examine some cases.

• $\mathbf{F} = (\mathsf{lock}_{\varphi''}\ []\ \mathsf{in}\ e'')$. We have $\varphi = \{\rho\} \cup \psi \cup \varphi''$ with $\Gamma \vdash_t e : \psi, \theta\,\mathsf{ref}_\rho$ for some θ and $\Gamma \vdash_t e'' : \varphi'', \tau$. If $\Gamma' \vdash_t e' : \psi', \theta\,\mathsf{ref}_\rho$ with $\Gamma \subseteq \Gamma'$ and $\psi' \cap \mathsf{dom}(\Gamma) \subseteq \psi$ then $\Gamma' \vdash_t (\mathsf{lock}_{\varphi''}\ e'\ \mathsf{in}\ e'') : \{\rho\} \cup \psi' \cup \varphi'', \tau$, and we conclude using the induction hypothesis on $\mathbf{E}'$.

• $\mathbf{F} = ([]\backslash p)_{\varphi'', P}$. We have $\Gamma \vdash_t e : \varphi, \tau$ with $\varphi \cap P \subseteq \varphi''$ and $P \subseteq \mathsf{dom}(\Gamma)$. If $\Gamma' \vdash_t e' : \varphi', \tau$ with $\Gamma \subseteq \Gamma'$ and $\varphi' \cap \mathsf{dom}(\Gamma) \subseteq \varphi$ then $\varphi' \cap P \subseteq \varphi''$, and therefore $\Gamma' \vdash_t (e'\backslash p)_{\varphi'', P} : \varphi', \tau$, and we conclude using the induction hypothesis on $\mathbf{E}'$. ❑

In order to show the type safety result, we have to extend the typing to configurations. The extension of typing to multisets of threads, that is $\Gamma \vdash T$, is given by

$$\frac{\Gamma \vdash_t e : \varphi, \tau}{\Gamma \vdash e} \qquad \frac{\Gamma \vdash T \qquad \Gamma \vdash T'}{\Gamma \vdash T \parallel T'}$$

Typing the store is defined as follows:

$$\Gamma \vdash S \quad \Leftrightarrow_{\text{def}} \quad \begin{cases} \mathsf{dom}(S) \subseteq \mathsf{dom}(\Gamma) \ \& \\ \forall p.\ \Gamma(p) = \theta\,\mathsf{ref}_p \ \&\ S(p) = v \ \Rightarrow\ \Gamma \vdash_t v : \emptyset, \theta \end{cases}$$

Finally one defines

$$\Gamma \vdash (S, L, T) \quad \Leftrightarrow_{\text{def}} \quad \Gamma \vdash S \ \&\ \Gamma \vdash T$$

Proposition (Subject Reduction) 5.7. *If $\Gamma \vdash (S, L, T)$ and $(S, L, T) \to (S', L', T')$ then $\Gamma' \vdash (S', L', T')$ for some Γ' such that $\Gamma \subseteq \Gamma'$.*

Proof Sketch: by case on the transition $(S, L, T) \to (S', L', T')$, where $T = \mathbf{E}[r] \parallel T''$ and r is the redex that is reduced. We only examine some cases.

- $r = (\lambda x e v)$. We have $S' = S$, $L' = L$ and $T' = \mathbf{E}[\{x \mapsto v\}e] \parallel T''$. There are two cases.
(i) $\Gamma, x : \zeta \vdash_t e : \varphi, \sigma$ and $\Gamma \vdash_t v : \emptyset, \zeta$ with $\zeta \neq \theta\,\mathsf{ref}_y$ and $\Gamma \vdash_t r : \varphi, \sigma$. We use the Substitution Lemma 5.5(ii) and the Replacement Lemma 5.6.
(ii) $\Gamma, x : \theta\,\mathsf{ref}_y \vdash_t e : \varphi, \sigma$ and $\Gamma \vdash_t v : \emptyset, \theta\,\mathsf{ref}_\rho$ with $\Gamma \vdash_t r : \{y \mapsto \rho\}(\varphi, \sigma)$ then by the well-formedness assumption we have $y = x$, and $v = \rho$ by Remark 5.3(i), and therefore by the Substitution Lemma 5.5(i) we have $\Gamma \vdash_t \{x \mapsto v\}e : \{y \mapsto \rho\}(\varphi, \sigma)$, and we conclude using the Replacement Lemma 5.6.
- $r = (\mathsf{lock}_\psi\ p\ \mathsf{in}\ e)$. We have $S = S'$ and $\Gamma \vdash_t r : \{p\} \cup \psi, \tau$ with $\Gamma = \Gamma', p : \theta\,\mathsf{ref}_p$ and $\Gamma \vdash_t e : \psi, \tau$. There two cases.
(i) $p \in \lceil \mathbf{E} \rceil$ and $L' = L$ and $T' = \mathbf{E}[e]$. We use the Replacement Lemma 5.6 to conclude.
(ii) $p \notin \lceil \mathbf{E} \rceil$, $L' = L \cup \{p\}$ and $T' = \mathbf{E}[(e \backslash p)_{\psi, P}]$ where $P = \mathsf{dom}(S)$. Then $P \subseteq \mathsf{dom}(\Gamma)$, and therefore $\Gamma \vdash_t (e \backslash p)_{\psi, P} : \psi, \tau$, and we conclude using the Replacement Lemma 5.6.
- $r = \nu x e$. We have $S' = S + p$ where $p \notin \mathsf{dom}(S)$ and $L' = L$ and $T = \mathbf{E}[\{x \mapsto p\}e] \parallel T''$. Then $\Gamma \vdash_t \nu x e : \varphi - \{x\}, \tau$ with $x \notin \Gamma, \tau$ and $\Gamma, x : \theta\,\mathsf{ref}_x \vdash_t e : \varphi, \tau$. By the Strengthening Lemma 5.4 (and well-formedness of configurations) we may assume that $p \notin \mathsf{dom}(\Gamma)$, and therefore $\Gamma, p : \theta\,\mathsf{ref}_p \vdash_t \{x \mapsto p\}e : \{x \mapsto p\}\varphi, \tau$ by the Substitution Lemma 5.5(i), and we use the Replacement Lemma 5.6 to conclude. ❏

Theorem (Type Safety) 5.8. *For any closed expression e of the source language, if $\Gamma \vdash_s e : \varphi, \tau \ \Rightarrow\ \bar{e}$ then $\bar{e}$ is safe.*

Proof: this is a consequence of Lemma 5.2 and the Subject Reduction property, since if

$$\mathbf{E}[(\mathbf{E}'[(\mathsf{lock}_{\psi_1}\ p_1\ \mathsf{in}\ e)] \backslash p_0)_{\psi_0, P_0}]$$

is typable, we have $p_1 \in P_0 \ \Rightarrow\ p_1 \in \psi_0$, for

$$\Gamma \vdash_t \mathbf{E}'[(\mathsf{lock}_{\psi_1}\ p_1\ \mathsf{in}\ e)] : \varphi, \tau \ \Rightarrow\ p_1 \in \varphi \cap \mathsf{dom}(\Gamma)$$

❏

An obvious consequence of this result and Proposition 4.4 is that, if the closed expression e of the source language is typable, and translates into $\bar{e}$, then executing the latter (in the initial configuration where $S = \emptyset = L$) is free from deadlocks.

6 Conclusion

Designing a semantics for shared variable concurrency that is provably free of deadlocks is a step towards a *modular* concurrent programming style, where one can compose a system from several (typable) threads and modules without running the risk of entering into a deadlock. We have proposed such a deadlock-free semantics, that relies on a static analysis of programs which is not much more constraining than usual typing. Moreover, thanks to the use of singleton reference types, we obtain a fine grained locking policy, where each pointer has its own lock. That is, the programmer does not have to think about locks, but only about pointers.

References

1. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: preventing data-races and deadlocks. In: OOPSLA 2002, pp. 211–230 (2002)
2. Coffman Jr, E.G., Elphick, M.J., Shoshani, A.: System Deadlocks. ACM Comput. Surveys 3(2), 67–78 (1971)
3. Flanagan, C., Abadi, M.: Types for safe locking. In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576, pp. 91–108. Springer, Heidelberg (1999)
4. Hayashi, S.: Singleton, union and intersection types for program extraction. In: Ito, T., Meyer, A.R. (eds.) TACS 1991. LNCS, vol. 526, pp. 701–730. Springer, Heidelberg (1991)
5. Kobayashi, N.: A new type system for deadlock-free processes. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 233–247. Springer, Heidelberg (2006)
6. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: POPL 1988, pp. 47–57 (1988)
7. McCloskey, B., Zhou, F., Gray, D., Brewer, E.: Autolocker: synchronization inference for atomic sections. In: POPL 2006, pp. 346–358 (2006)
8. Nanevski, A., Morrisett, G., Shinnar, A., Birkedal, L.: Ynot: dependent types for imperative programs. In: ICFP 2008, pp. 229–240 (2008)
9. Peyton Jones, S.L.: Beautiful concurrency. In: Oram, A., Wilson, G. (eds.) Beautiful Code. O'Reilly, Sebastopol (2007)
10. Suenaga, K.: Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 155–170. Springer, Heidelberg (2008)
11. Vasconcelos, V., Martins, F., Cogumbreiro, T.: Type inference for deadlock detection in a multithreaded polymorphic typed assembly language. In: Proceedings of PLACES 2009 (2009)
12. Wang, Y., Lafortune, S., Kelly, T., Kudlur, M., Mahlke, S.: The theory of deadlock avoidance via discrete control. In: POPL 2009, pp. 252–263 (2009)
13. Wright, A., Felleisen, M.: A syntactic approach to type soundness. Information and Computation 115(1), 38–94 (1994)