A Deadlock-Free Semantics
for
Shared Memory Concurrency

Motivation

Shared memory concurrency:

- traditional algorithmic programming style
- not far from the machine architecture

➥ needed at some implementation stage.

Problem: not modular (data-races, deadlocks...)

➥ keep the model, restrict the semantics, avoiding deadlocks.

An Example (1/3)

Bank accounts: a module where an account is a reference (pointer) to an integer, with some functions:

- to deposit some amount x on account y: $\lambda x \lambda y (y := \,!\,y + x)$.

Incorrect:

$$(\{a \mapsto 0\}, (\textsf{deposit}\,100\,a) \parallel (\textsf{deposit}\,100\,a)) \xrightarrow{*} (\{a \mapsto 100\}, ())$$

➥ needs an exclusive access to y:

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

where $(\textsf{lock}\ y\ \textsf{in}\ e)$ takes the [lock associated with] reference y, and release it upon termination of e.

An Example (2/3)

- to withdraw an amount x from account y:

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

- to transfer x from y to z:

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

Notice: reentrant locks.

An Example (3/3)

One may transfer money in any direction:

$$(\mathsf{transfer}\,100\,a\,b)\ \parallel\ (\mathsf{transfer}\,10\,b\,a)$$

where $!\,a \geq 100$ and $!\,b \geq 10$.

An Example (3/3)

One may transfer money in any direction:

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

where $!\,a \geq 100$ and $!\,b \geq 10$.

➥ potential deadlock:

$$\xrightarrow{*} \underbrace{(\mathsf{lock}\ b\ \mathsf{in}\ b := !\,b + 100)}_{holding\ \ a} \;\|\; \underbrace{(\mathsf{lock}\ a\ \mathsf{in}\ a := !\,a + 10)}_{holding\ \ b}$$

Deadlocks: Solutions

- deadlock prevention: only run code that is guaranteed to be free of deadlocks;
- deadlock avoidance: monitoring the execution so as to avoid dangerous states;
- deadlock detection and recovery: supervise execution and rollback (undoing operations) in case of deadlock.

Prevention: by means of static analysis, checking that locks are taken in some order.

➥ a unique lock for all the bank accounts!

Our Solution

Deadlock avoidance:

- static analysis by means of a type and effect system, anticipating the pointers to lock as the effect,
- translation of source programs into annotated programs

  $$(\text{lock } e_0 \text{ in } e_1) \;\Rightarrow\; (\text{lock}_\psi\ \overline{e_0} \text{ in } \overline{e_1})$$

  where ψ is the effect of e_1 = set of pointers to be locked by e_1,
- prudent semantics: to execute $(\text{lock}_\psi\ p \text{ in } e)$ one does not lock p if some pointer in ψ is already held by another thread.

➥ **type safety:** the annotated programs obtained by translation from typable source programs, executed in the prudent semantics, are free of deadlocks.

Technically

In analysing (**lock** e_0 **in** e_1) one has to have some information about the pointer to be locked, i.e. the value of e_0, to be recorded in the effect, and then used in the types.

- replace (**ref** e) with (**cref** e), a function f to create a pointer with initial value the value of e,
- restricted, by typing, to be used in a particular context, namely

$$(\textbf{let } x = (f()) \textbf{ in } e)$$

↪ singleton reference types $\theta\,\mathbf{ref}_x$, i.e. locks univocally associated with pointers.

Source Language

functional + imperative + concurrent:

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

Notation: $(\mathsf{ref}\ e)$ for $((\mathsf{cref}\ e)())$.

Target Language

$$
\begin{array}{rcll}
p,\ q \ldots & & & \textit{pointers} \\
v,\ w \ldots & ::= & x \mid \lambda x e \mid () \mid p & \textit{values} \\
e & ::= & v \mid (e_1 e_0) & \textit{expressions} \\
& \mid & (!\, e) \mid (e_0 := e_1) & \\
& \mid & (\mathsf{thread}\, e) \mid (\mathsf{lock}_\varphi\ e_0\ \mathsf{in}\ e_1) & \\
& \mid & (e \backslash p) \mid (\mathsf{new}\ x\ \mathsf{in}\ e) &
\end{array}
$$

where φ is an effect, that is a finite set of pointer names (either constant or variable)

TYPING and TRANSLATION (1/3)

Types:

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

In $(\theta\,\textsf{ref}_x \xrightarrow{\varphi} \sigma)$ the variable x is universally quantified, with scope φ and σ.

Judgements:

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

TYPING and TRANSLATION (2/3)

Main (unusual) rules:

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

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

i.e.

$$\begin{array}{lcl} (\mathsf{cref}\ e) & \Rightarrow & (\mathsf{let}\ x = \bar{e}\ \mathsf{in}\ \lambda y((y := x)\ ;\ y)) \\ \theta\ \mathsf{cref} & \Rightarrow & (\theta\ \mathsf{ref}_y \xrightarrow{\emptyset} \theta\ \mathsf{ref}_y) \end{array}$$

Typing and Translation (3/3)

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

where y is fresh and $x \notin \Gamma, \varphi_0, \tau$. Notice: the only way to create a reference is $(\mathsf{let}\ x = (e_0())\ \mathsf{in}\ e_1)$ where e_0 has type $\theta\,\mathsf{cref}$.

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

Example

$$\Gamma \vdash \mathsf{deposit} : \emptyset, \mathsf{int} \xrightarrow{\emptyset} (\mathsf{int\,ref}_y \xrightarrow{\{y\}} \mathsf{unit})$$

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

(polymorphic types) with translations

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

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

and one can type

$$\begin{aligned} &\mathsf{let\ create\_account}\ = \lambda x(\mathsf{cref}\, x)\ \mathsf{in} \\ &\quad \mathsf{let}\ a = (\mathsf{create\_account}\ 100)()\ \mathsf{in} \\ &\quad \mathsf{let}\ b = (\mathsf{create\_account}\ 10)()\ \mathsf{in}\ \cdots \end{aligned}$$

where a and b have distinct types, $\mathsf{int\,ref}_a$ and $\mathsf{int\,ref}_b$.

Prudent Semantics

Main (unusual) rules:

$$
\begin{array}{rcll}
(S, L, \mathbf{E}[(\mathsf{lock}_\psi\ p\ \mathsf{in}\ e)] \parallel T) & \rightarrow & (S, L, \mathbf{E}[e] \parallel T) & p \in \lceil \mathbf{E} \rceil \\
(S, L, \mathbf{E}[(\mathsf{lock}_\psi\ p\ \mathsf{in}\ e)] \parallel T) & \rightarrow & (S, L', \mathbf{E}[(e \backslash p)] \parallel T) & p \notin \lceil \mathbf{E} \rceil \\
 & & & \&\ (\spadesuit) \\
(S, L, \mathbf{E}[(v \backslash p)] \parallel T) & \rightarrow & (S, L - \{p\}, \mathbf{E}[v] \parallel T) &
\end{array}
$$

where $p \in \lceil \mathbf{E} \rceil$ means that p is currently locked by the thread, and

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

vs standard condition: $L \cap \{p\} = \emptyset$.

Results

- **Type Safety:** if $\Gamma \vdash e : \varphi, \tau \Rightarrow \bar{e}$ then evaluating $\bar{e}$ in the prudent semantics is free of deadlocks.
- modularity: composing systems of (typable) threads is safe – no deadlock.
- fine grained locking policy: each pointer has its own lock (the programmer does not have to think about locks at run time – only pointers).
- simple “pessimistic” semantics: only local (i.e. per thread) conditions, no global analysis of the current state, no rollback.