Fair Cooperative Multithreading

or

Typing Termination in a Higher-Order Concurrent Imperative Language

Gérard Boudol
INRIA Sophia Antipolis

CONCUR’07, Lisboa
Concurrent sequential programs that:

- share a memory,
- may spawn new threads,
- run until completion or cooperation.

≠ interleaving, where threads (or rather their executable code) are preempted by the scheduler.

A thread leaves its turn of execution for another thread by performing specific cooperation instructions like yield (or synchronization operations).
Cooperative Threads (2/2)

Pros – as opposed to preemptive scheduling:

- no data race, no need for mutual exclusion,
- modularity: no need to rewrite libraries,
- scheduling controlled at the application level (no ill-timed context switching), with a deterministic implementation,
- easier to program with, better performance.

Cons:

- not directly suited for “true concurrency” (exploiting multicore architectures),
- threads must be fair, or cooperative.
A Problem/A Solution?

Purposely non-terminating programs: any server for instance should not be programmed to terminate.

How can we guarantee that such a program is fair?

- distinguish a specific recursion construct $\nabla yP$ for “purposely non-terminating programs”, $\neq$ ordinary recursive functions,
- yield the scheduler on every recursive call $\nabla yP \rightarrow \{y \mapsto \nabla yP\}P$.

Is this fair? Should be... (provided ordinary recursive programs terminate).
An imperative and functional language: Core ML (cf. JAVA: mutable fields and methods), plus threads.

Syntax:

\[
M, N \ldots ::= V \quad \text{value} \\
\quad | (MN) \quad \text{application of the function } M \text{ to the argument } N \\
\quad | (\text{ref} M) \quad \text{creation of a new memory location} \\
\quad | (!M) \quad \text{contents of a memory location} \\
\quad | (M := N) \quad \text{assignment} \\
\quad | (\text{thread} M) \quad \text{creation of a new thread}
\]
Values:

\[ V ::= x \quad \text{variable} \]
\[ \quad | \quad \lambda x M \quad \text{anonymous function, of } x, \text{ returning } M \]
\[ \quad | \quad \nabla y M \quad \text{“yield-and-loop”} \]
\[ \quad | \quad () \quad \text{termination} \]

Examples:

\[
\text{yield} = (\nabla y ()() )
\]
\[
\text{(repeat } M) = \mu y. (\text{thread } y()); M \quad \text{where}
\]
\[
\mu y M = \{ y \mapsto \nabla y M \} M
\]
Transitions between configurations \((\mu, M, T, S)\).

Configuration = shared memory \(\mu\),
active thread \(M\),
multiset \(T\) of threads in the current turn,
multiset \(S\) of threads in the next turn of execution.

\[
(\mu, E[(\text{thread } M)], T, S) \rightarrow (\mu, E[\emptyset], T + M, S)
\]
\[
(\mu, E[(\nabla y M(\cdot))], T, S) \rightarrow (\mu, \emptyset, T, S + E[\{y \mapsto \nabla y M\}M])
\]
\[
(\mu, V, N + T, S) \rightarrow (\mu, N, T, S)
\]
\[
(\mu, V, \emptyset, N + S) \rightarrow (\mu, N, S, \emptyset)
\]

Sequential constructs: as usual.
**PROBLEM: Recursion without Recursion**

Two ways of diverging in an imperative and functional language, without explicit recursive call:

- recursion by means of \(\lambda\)-calculus fixpoint combinators.
- type system.

- recursion by means of circular references [Landin 64]:

\[
\text{rec } f(x) M \simeq \text{let } y = (\text{ref } \lambda x M) \\
\text{in } y := \lambda x (\lambda f M(!y)) ; !y
\]

- type and effect system, to eliminate circularities in the memory.

**Expected result:** typed threads are fair, i.e.

\[
(\mu, M) \text{ typable } \Rightarrow \exists V... (\mu, M, \emptyset, \emptyset) \rightarrow^* (\mu', V, T, S')
\]
The **Realizability Technique** (1/2)

To prove properties akin to termination (strong normalization, evaluation to a head-normal form...) for typed expressions: define an interpretation of types as sets of expressions, s.t.

- the interpretation \([\lbrack \tau \rbrack]\) of a type contains only expressions enjoying the intended computational property (e.g. to be “fair”);
- an expression typed \(\tau\) belongs to \([\lbrack \tau \rbrack]\), or realizes \(\tau\) (\(\models M : \tau\)),

by induction on types. Main ingredient:

\[
\models M : \tau \rightarrow \sigma \iff \forall N. \models N : \tau \Rightarrow \models (MN) : \sigma
\]

A very general technique for typed \(\lambda\)-calculi.
The Realizability Technique (2/2)

not available for higher-order imperative (and concurrent) languages.

- A difficulty: applying a function of type $\tau \rightarrow \sigma$ may read/update memory locations of type $\theta$, not smaller than $\tau$ or $\sigma$ (cf. “Landin’s trick”).

- cannot define a realizability interpretation by induction on types.

(Pitts & Stark 98: memory restricted to contain only values of basic types – boolean, integer... no function in the memory.)
The Type and Effect System

[Lucassen & Gifford 88]:

- The memory is partitioned into regions $\rho$.
- **Judgements:** $\Gamma \vdash M : e, \tau$, meaning "$M$ has effect $e$ and type $\tau$ in the typing context $\Gamma$.”
- **Effect:** set of regions $e$ where $M$ may create, read or update a reference.
- **Types:**

$$\tau, \theta, \sigma \ldots ::= \text{unit} \mid \theta \text{ref}_\rho \mid (\tau \overset{e}{\rightarrow} \sigma)$$
Main idea here: **stratification** of the memory by means of regions:

*a function of type* \( (\tau \rightarrow \sigma) \) *stored in region* \( \rho \) *does not have a latent effect in region* \( \rho \), *i.e.* \( \rho \not\in e \).

“Landin’s trick” is **not typable**.

- **New**: enriched judgements \( \Delta; \Gamma \vdash M : e, \tau \) with a **region typing** context \( \Delta = \rho_1 : \theta_1, \ldots, \rho_n : \theta_n \) associating types to regions,

- with predicates \( \Delta \vdash \) and \( \Delta \vdash \tau \) of “well-formedness” of region contexts and types resp.
Well-formedness:

\[
\begin{align*}
\emptyset & \vdash \\
\Delta & \vdash \theta \\
\Delta, \rho : \theta & \vdash \rho \notin \text{dom}(\Delta) \quad \Delta & \vdash \Delta(\rho) = \theta \\
\Delta & \vdash \theta \text{ref}_\rho \\
\Delta & \vdash \text{unit} \\
\Delta & \vdash \tau \\
\Delta & \vdash \sigma \\
e & \subseteq \text{dom}(\Delta) \\
\Delta & \vdash (\tau^e \rightarrow \sigma)
\end{align*}
\]

\[
\Rightarrow \text{ applying a function of type } (\tau^e \rightarrow \sigma) \text{ only has effects at strictly “smaller” types.}
\]

The typing rules are standard, except that the types are checked for well-formedness against the region context.
**Imperative Realizability (1/2)**

For $M$ closed: $\Delta \models M : e, \tau \iff_{\text{def}}$ if the memory $\mu$ satisfies

$$\rho \in e \& \Delta(\rho) = \theta \Rightarrow \Delta \models \mu(u_\rho) : \theta \quad (\ast)$$

then computing $(\mu, M, \emptyset, \emptyset)$

- has only effects in $e$,
- cooperates, i.e. converges to a value $V$ (while possibly spawning new threads),
- which realizes $\tau$: $\Delta \models V : \tau \quad (\ast)$,

$(\ast)$ where $\Delta \models V : \tau$ is defined by induction on $\tau$:

- $\Delta \models V : \text{unit} \iff_{\text{def}} V = ()$
- $\Delta \models V : \theta \text{ref}_\rho \iff_{\text{def}} V$ is a reference in region $\rho$
- $\Delta \models V : (\theta \xrightarrow{e} \sigma) \iff_{\text{def}} \forall W.\Delta \models W : \theta \Rightarrow \Delta \models (V W) : e, \sigma$
IMPERATIVE REALIZABILITY

For $\Delta \vdash$, the definition of $\Delta \models M : e, \tau$ is well-founded w.r.t. an ordering $\prec_\Delta$ on pairs $(e, \tau)$ s.t.

1. $\rho \in e \& \Delta(\rho) = \theta \Rightarrow (\emptyset, \theta) \prec_\Delta (e, \tau)$
2. $(\emptyset, \tau) \prec_\Delta (e, (\tau \xrightarrow{e'} \sigma)) \text{ and } (e', \sigma) \prec_\Delta (e, (\tau \xrightarrow{e'} \sigma))$

**Main result:** The type and effect system is sound w.r.t. the realizability interpretation.

**Corollary (Fairness/Type Safety):**

1. any typable expression cooperates, i.e. yields a value;
2. the “current turn” always terminates: any typable thread system $(\mu, M, T, S)$ reduces to $(\mu', V, \emptyset, S + S')$ for some value $V$. 
The “yield-and-loop” construct for programming non-terminating processes is indeed a solution to our fairness problem (together with a stratification of types) – but the proof needs some machinery...