

Fair Cooperative Multithreading

or

Typing Termination in a Higher-Order Concurrent Imperative Language

Gérard Boudol

INRIA Sophia Antipolis

CONCUR'07, Lisboa

COOPERATIVE THREADS

1
(1/2)

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** ∇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).

A LANGUAGE

(1/2)

An **imperative** and **functional** language: Core ML (*cf.* JAVA: mutable fields and methods), plus **threads**.

Syntax:

$M, N \dots$	$::=$	V	<i>value</i>
		(MN)	<i>application of the function M to the argument N</i>
		$(\mathbf{ref} M)$	<i>creation of a new memory location</i>
		$(! M)$	<i>contents of a memory location</i>
		$(M := N)$	<i>assignment</i>
		$(\mathbf{thread} M)$	<i>creation of a new thread</i>

A LANGUAGE

(2/2)

Values:

V	$::=$	x	<i>variable</i>
		$\lambda x M$	<i>anonymous function, of x, returning M</i>
		$\nabla y M$	<i>“yield-and-loop”</i>
		$()$	<i>termination</i>

Examples:

$$\begin{aligned} \text{yield} &= (\nabla y () ()) \\ (\text{repeat } M) &= \mu y. (\text{thread } y ()) ; M \quad \text{where} \\ \mu y M &= \{y \mapsto \nabla y M\} M \end{aligned}$$

SEMANTICS (HIGHLIGHTS)

Transitions between configurations (μ, M, T, S) .

Configuration = shared memory μ ,
 active thread M ,
 multiset T of threads in the **current turn**,
 multiset S of threads in the **next turn** of execution.

$$(\mu, \mathbf{E}[(\mathbf{thread} M)], T, S) \rightarrow (\mu, \mathbf{E}[\emptyset], T + M, S)$$

$$(\mu, \mathbf{E}[(\nabla y M \hat{\ })], T, S) \rightarrow (\mu, \hat{\ }, T, S + \mathbf{E}[\{y \mapsto \nabla y M\} M])$$

$$(\mu, V, N + T, S) \rightarrow (\mu, N, T, S)$$

$$(\mu, V, \emptyset, N + S) \rightarrow (\mu, N, S, \emptyset)$$

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 λ -calculus **fixpoint combinators**.

↳ **type system**.

▶ recursion by means of **circular references** [Landin 64]:

$$\begin{aligned} \text{rec } f(x)M &\simeq \text{let } y = (\mathbf{ref } \lambda x M) \\ &\quad \text{in } y := \lambda x (\lambda f M (!y)) ; !y \end{aligned}$$

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

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

$$(\mu, M) \text{ typable} \Rightarrow \exists V \dots (\mu, M, \emptyset, \emptyset) \xrightarrow{*} (\mu', V, T, S)$$

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 $\llbracket \tau \rrbracket$ of a type contains only expressions enjoying the intended computational property (e.g. to be “fair”);
- ▶ an expression typed τ belongs to $\llbracket \tau \rrbracket$, or **realizes** τ ($\models M : \tau$),

by **induction on types**. Main ingredient:

$$\models M : \tau \rightarrow \sigma \Leftrightarrow \forall N. \models N : \tau \Rightarrow \models (MN) : \sigma$$

A very general technique for typed λ -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 θ , **not smaller** than τ or σ (*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

(1/3)

[Lucassen & Gifford 88]:

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

$$\tau, \theta, \sigma \dots ::= \text{unit} \mid \theta \text{ref}_{\rho} \mid (\tau \xrightarrow{e} \sigma)$$

The TYPE and EFFECT SYSTEM

(2/3)

Main idea here: **stratification** of the memory by means of regions:

a function of type $(\tau \xrightarrow{e} \sigma)$ stored in region ρ does not have a latent effect in region ρ , i.e. $\rho \notin e$.

↳ “Landin’s trick” is **not typable**.

- ▶ **New:** enriched judgements $\Delta; \Gamma \vdash M : e, \tau$ with a **region typing** context $\Delta = \rho_1 : \theta_1, \dots, \rho_n : \theta_n$ associating types to regions,
- ▶ with predicates $\Delta \vdash$ and $\Delta \vdash \tau$ of “**well-formedness**” of region contexts and types resp.

The TYPE and EFFECT SYSTEM

(3/3)

Well-formedness:

$$\frac{}{\emptyset \vdash} \quad \frac{\Delta \vdash \theta}{\Delta, \rho : \theta \vdash} \quad \rho \notin \text{dom}(\Delta) \quad \frac{\Delta \vdash \quad \Delta(\rho) = \theta}{\Delta \vdash \theta \text{ref}_\rho}$$

$$\frac{\Delta \vdash}{\Delta \vdash \text{unit}} \quad \frac{\Delta \vdash \tau \quad \Delta \vdash \sigma \quad e \subseteq \text{dom}(\Delta)}{\Delta \vdash (\tau \xrightarrow{e} \sigma)}$$

➔ applying a function of type $(\tau \xrightarrow{e} \sigma)$ 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 μ satisfies

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

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 τ : $\Delta \models V : \tau$ (*),

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

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

IMPERATIVE REALIZABILITY

(2/2)

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

- ▶ $\rho \in e \ \& \ \Delta(\rho) = \theta \Rightarrow (\emptyset, \theta) \prec_{\Delta} (e, \tau)$
- ▶ $(\emptyset, \tau) \prec_{\Delta} (e, (\tau \xrightarrow{e'} \sigma))$ 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):

- ▶ any **typable** expression **cooperates**, i.e. yields a value;
- ▶ the “current turn” always terminates: any **typable thread system** (μ, M, T, S) reduces to $(\mu', V, \emptyset, S + S')$ for some value V .

CONCLUSION

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...