Concurrency theory

functions as processes, simple types

Francesco Zappa Nardelli

INRIA Rocquencourt, MOSCOVA research team

francesco.zappa_nardelli@inria.fr

together with

Frank Valencia (INRIA Futurs) Catuscia Palamidessi (INRIA Futurs) Roberto Amadio (PPS)

Function as processes: call-by-value lambda calculus

The call-by-value lambda calculus:

$$
\begin{array}{llclclcl}
\text{terms} & M, N & ::= & x & \mid & \lambda x.M & \mid & M\,N \\
\text{values} & V & ::= & x & \mid & \lambda x.M & &
\end{array}
$$

Reduction rules:

$$
(\lambda x.M)\,V \;\rightharpoonup\; M\{^{V}\!/_{x}\}
\qquad\qquad
\frac{M \rightharpoonup M'}{M\,N \rightharpoonup M'\,N}
\qquad\qquad
\frac{N \rightharpoonup N'}{V\,N \rightharpoonup V\,N'}
$$

Lambda vs. pi

- Lambda-calculus substitues *terms for variables*; pi-calculus relies only on name substitutions;

- In $(\lambda x.M)\,N$ the terms $\lambda x.M$ and N are committed to interacting with each other, even if they are part of a larger term; interference for the environment can prevent pi-calculus term to interact.

- Lambda-calculus is *sequential*: *parallel or* cannot be defined in PCF

$$\texttt{Por}\ \ M\ \ N = \begin{cases} \text{true if } M \text{ or } N \text{ reduces to true} \\ \text{diverges otherwise} \end{cases}$$

Call-by-value CPS transform (Plotkin)

Idea: function calls terminate by passing their result to a continuation.

Example:

$$\lambda y.\,\texttt{let}\ \ g = \lambda n.\, n + 2\ \ \texttt{in}\ \ (g\ \ y) + (g\ \ y)$$

can be rewritten as

$$\begin{aligned}\lambda k\ \ y.\,\texttt{let}\ \ g = \lambda k\ \ n.\, k\ \ (n + 2)\ \ \texttt{in}\\ g\ \ (\lambda v.\, g(\lambda w.\, k\,(v + w))\ \ y)\ \ y\end{aligned}$$

Transformation:

$$[[V]] = \lambda k.\, k[V] \qquad\qquad [x] = x$$

$$[[M\ N]] = \lambda k.\,[[M]](\lambda v.\,[[N]](\lambda w.\, v\, w\, k)) \qquad\qquad [\lambda x.M] = \lambda x.[[M]]$$

Call-by-value CPS transform, ctd.

Exercise: simulate the β-reduction

$$(\lambda x.M)V \rightarrow M\{^V/_x\}$$

after the CPS transform.

Further readings: if you want to know more about continuations, then some readings:

- Reynolds, *The discoveries of continuations*;
- Appel, *Compiling with continuations*;
- the works of Plotkin, Danvy, Gordon, Felleisen, etc...

(Remark: this list is wildly uncomplete).

Encoding of call-by-value lambda into pi

Notations: Suppose $[[M]]$ is $(q).Q$. Then $y(x)[[M]]$ is $y(x,q).Q$, and $[[M]]r$ is $Q\{^r/_q\}$.

Encoding:

$$[[\lambda x.M]] \;=\; (p).\,\overline{p}\langle y\rangle.\,!y(x)[[M]]$$

$$[[x]] \;=\; (p).\overline{p}\langle x\rangle$$

$$[[M\;N]] \;=\; (p).\,(\boldsymbol{\nu} q)([[M]]q \;\|\; q(v).(\boldsymbol{\nu} r)([[N]]r \;\|\; r(w).\overline{v}\langle w,p\rangle))$$

If $M =_\beta V$, then $[[M]]p$ reduces to a process that, very roughly, returns the value V at p. When V is a function, it cannot be passed directly at p: instead we pass a pointer to the function.

And call-by-name?

The call-by-name lambda calculus:

$$\text{terms} \quad M, N \ ::= \ x \ \mid \ \lambda x.\, M \ \mid \ M\, N$$

Reduction rules:

$$(\lambda x.M)\, N \ \rightharpoondown \ M\{^N/_x\} \qquad\qquad \frac{M \rightharpoondown M'}{M\, N \rightharpoondown M'\, N}$$

Call-by-name CPS transform (Plotkin)

Transform:

$$
\begin{aligned}
[[x]] &= \lambda k.\, x\, k \\
[[\lambda x.M]] &= \lambda k.\, k\, (\lambda x.[[M]]) \\
[[M\ N]] &= \lambda k.\, [[M]](\lambda v.\, v[[N]]k)
\end{aligned}
$$

Remark: the encoding of a variable x is used as a trigger for activating a term and providing it with a location.

Exercise: simulate β-reduction in the transform.

Encoding of call-by-name lambda into pi

Notations: Suppose $[[M]]$ is $(q).Q$. Then $y(x)[[M]]$ is $x(y,q).Q$, and $[[M]]r$ is $Q\{^r/_q\}$.

Encoding:

$$[[\lambda x.M]] \;=\; (p).\,\overline{p}\langle v\rangle.\,v(x)[[M]]$$

$$[[x]] \;=\; (p).\overline{x}\langle p\rangle$$

$$[[M\;N]] \;=\; (p).\,(\boldsymbol{\nu} q)\,([[M]]q \;||\; q(v).(\boldsymbol{\nu} x)(\overline{v}\langle x,p\rangle.!x[[N]]))$$

A term that is evaluated may only be the operand – not the argument – of an application: as such it cannot be copied. Also, the argument is passed without being evaluated, and can be evaluated every time its value is needed.

Other reduction strategies

It is possible to encode other reduction strategies. For instance, parallel call-by-value.

$$(\lambda x.M)\,V \;\rightharpoonup\; M\{{}^{V}\!/_{x}\} \qquad \frac{M \rightharpoonup M'}{M\,N \rightharpoonup M'\,N} \qquad \frac{N \rightharpoonup N'}{M\,N \rightharpoonup M\,N'}$$

We rely on parallel composition to perform parallel reduction of M and N:

$$[[M\,N]] \;=\; (p).\,(\boldsymbol{\nu} q,r)([[M]]q \;\|\; [[N]]r \;\|\; q(v).r(w).\overline{v}\langle w,p\rangle)$$

Observational congruence for call-by-name lambda

Since call-by-name is deterministic, the definition of observation equivalence (barbed congruence) can be simplified, by removing the bisimulation clause on interactions.

Definition: We say that M and N are *observationally equivalent*, if, in all closed contexts $C[-]$, it holds that $C[M]\Downarrow$ iff $C[N]\Downarrow$.

A tractable characterisation of observation equivalence:

Definition: *Applicative bisimilarity* is the largest symmetric relation $\approx_\lambda$ on lambda terms such that whenever $M \approx_\lambda N$, $M \rightarrowtriangle^* \lambda x.M'$ implies $N \rightarrowtriangle^* \lambda x.N'$ with $M'\{^L/_x\} \approx_\lambda N'\{^L/_x\}$ for all L.

Observation equivalence and applicative bisimilarity *coincide* (Stoughton).

Soundness and non-completeness

Let M and N be two lambda terms. We write $M =_\pi N$ if $[[M]] \cong [[N]]$.

Theorem (soundness): If $M =_\pi N$ then $M \approx_\lambda N$.

We would not expect completeness to hold: pi-calculus contexts are potentially more discriminating than lambda-calculus contexts.

More in detail:

- the CPS transform itself is not complete: there are terms that are applicative bisimilar, but whose CPS images are distinguishable (and thus separated by appropriate pi-calculus contexts);
- the canonical model (Abramsky) is sound but not complete, because the model contains denotations of terms not definable (eg. the *parallel convergence test*), and whose addition increases the discriminating power of the contexts.

See Sangiorgi and Walker book, part 4, for more details.

Types and sequential languages

In sequential languages, types are *"widely"* used:

- to detect simple programming errors at compilation time;
- to perform optimisations in compilers;
- to aid the structure and design of systems;
- to compile modules separately;
- to reason about programs;
- ahem, etc...

Data types and pi-calculus

In pi-calculus, the only values are *names*. We now extend pi-calculus with *base values* of type `int` and `bool`, and with *tuples*.

Unfortunately (?!) this allows writing terms which make no sense, as

$$\overline{x}\langle \mathtt{true} \rangle.P \;\Big|\Big|\; x(y).\overline{z}\langle y+1 \rangle$$

or (even worse)

$$\overline{x}\langle \mathtt{true} \rangle.P \;\Big|\Big|\; x(y).\overline{y}\langle 4 \rangle \ .$$

These terms raise *runtime errors*, a concept you should be familiar with.

Preventing runtime errors

We know that $3 : \texttt{int}$ and $\texttt{true} : \texttt{bool}$.

Names are values (they denote channels). Question: in the term

$$P \equiv \overline{x}\langle 3\rangle.P'$$

which type can we assign to x?

Idea: state that x is a channel that can transport values of type `int`. Formally

$$x : \mathsf{ch}(\texttt{int}) \ .$$

A complete type system can be developed along these lines...

Simply-typed pi-calculus: syntax and reduction semantics

Types:

$$T \quad ::= \quad \mathsf{ch}(T) \quad | \quad T \times T \quad | \quad \mathtt{unit} \quad | \quad \mathtt{int} \quad | \quad \mathtt{bool}$$

Terms (messages and processes):

$$M \quad ::= \quad x \quad | \quad (M, M) \quad | \quad () \quad | \quad 1, 2, \ldots \quad | \quad \mathtt{true} \quad | \quad \mathtt{false}$$

$$\begin{array}{rcl} P & ::= & \mathbf{0} \quad | \quad x(y : T).P \quad | \quad \overline{x}\langle M \rangle.P \quad | \quad P \parallel P \quad | \quad (\boldsymbol{\nu} x : T)P \\ & & | \quad \mathsf{match}\ z\ \mathsf{with}\ (x : T_1, y : T_2)\ \mathsf{in}\ P \quad | \quad !P \end{array}$$

Notation: we write $w(x, y).P$ for $w(z : T_1 \times T_2).\mathsf{match}\ z\ \mathsf{with}\ (x : T_1, y : T_2)\ \mathsf{in}\ P$.

Simply-typed pi-calculus: the type system

Type environment: $\Gamma \;::=\; \emptyset \;\;\Big|\;\; \Gamma, x{:}T.$

Type judgements:

- $\Gamma \vdash M : T$ value M has type T under the type assignement for names Γ;
- $\Gamma \vdash P$ process P respects the type assignement for names Γ.

Simply-typed pi-calculus: the type rules (excerpt)

Messages:

$$3 : \texttt{int} \qquad \frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \frac{\Gamma \vdash M_1 : T_1 \quad \Gamma \vdash M_2 : T_2}{\Gamma \vdash (M_1, M_2) : T_1 \times T_2}$$

Processes:

$$\Gamma \vdash \mathbf{0} \qquad \frac{\Gamma \vdash P_1 \quad \Gamma \vdash P_2}{\Gamma \vdash P_1 \mid\mid P_2} \qquad \frac{\Gamma, x{:}T \vdash P}{\Gamma \vdash (\boldsymbol{\nu} x : T)P}$$

$$\frac{\Gamma \vdash x : \mathsf{ch}(T) \quad \Gamma, y{:}T \vdash P}{\Gamma \vdash x(y : T).P} \qquad \frac{\Gamma \vdash x : \mathsf{ch}(T) \quad \Gamma \vdash M : T \quad \Gamma \vdash P}{\Gamma \vdash \overline{x}\langle M \rangle.P}$$

Soundness

The soundness of the type system can be proved along the lines of Wright and Felleisen's *syntactic approach to type soundness*.

- extend the syntax with the `wrong` process, and add reduction rules to capture runtime errors:

$$\frac{\text{where } x \text{ is not a name}}{\overline{x}\langle M\rangle.P \xrightarrow{\tau} \texttt{wrong}} \qquad \frac{\text{where } x \text{ is not a name}}{x(y{:}T).P \xrightarrow{\tau} \texttt{wrong}}$$

- prove that if $\Gamma \vdash P$, with Γ closed, and $P \rightarrow\!\!\triangleright^* P'$, then P' does not have `wrong` as a subterm.

Soundness, ctd.

Lemma Suppose that $\Gamma \vdash P$, $\Gamma(x) = T$, $\Gamma \vdash v : T$. Then $\Gamma \vdash P\{^{v}/_{x}\}$.

Proof. Induction on the derivation of $\Gamma \vdash P$.

Theorem Suppose $\Gamma \vdash P$, and $P \xrightarrow{\alpha} P'$.

1. If $\alpha = \tau$ then $\Gamma \vdash P'$.
2. If $\alpha = a(v)$ then there is T such that $\Gamma \vdash a : \mathsf{ch}(T)$ and if $\Gamma \vdash v : T$ then $\Gamma \vdash P'$.
3. If $\alpha = (\boldsymbol{\nu}\tilde{x} : \tilde{S})\overline{a}\langle v\rangle$ then there is T such that $\Gamma \vdash a : \mathsf{ch}(T)$, $\Gamma, \tilde{x} : \tilde{S} \vdash v : T$, $\Gamma, \tilde{x} : \tilde{S} \vdash P'$, and each component of $\tilde{S}$ is a link type.

Proof. At the blackboard.

pi-calculus is hardly ever used untyped

Types can be enriched in many ways.

- *Familiar type constructs:* union, record, variants, basic values, functions, linearity, polymorphsim,. . .
- *Type constructs specific to processes:* process-passing, receptiveness, deadlock-freedom, sessions, . . .

In all cases, types are assigned to names.