Functions As Processes

Overview
- An asynchronous \( \pi \)-calculus sublanguage
- The lambda-calculus
- \( \pi \)-calculus characterizations of \( \lambda \)-calculus reduction strategies

References
- Robin Milner, “Functions as Processes”
- Gerald K. Ostheimer and Antony J. T. Davie, “\( \pi \)-Calculus Characterizations of some Practical \( \lambda \)-Calculus Reduction Strategies”
- Davide Sangiorgi, “Lazy functions and mobile processes”

The Mini \( \pi \)-Calculus

The mini \( \pi \)-calculus is built from the operators of inaction, input prefix, output, parallel composition, restriction, and replication:

\[
P ::= 0 \mid a.b.P \mid \exists b.P \mid P \cdot P \mid \nu a.P \mid P \\
\]

\begin{align*}
\text{OUT:} & \quad \exists b.\ldots \Rightarrow 0 \\
\text{INP:} & \quad a \cdot \exists b.\ldots \Rightarrow P \\
\text{REPL:} & \quad P \Rightarrow \nu a.P \\
\text{COMP:} & \quad P \cdot Q \Rightarrow (P(a)Q) \\
\text{OPER:} & \quad \nu a.b.P \Rightarrow \nu a.b.Q \\
\text{PAR*:} & \quad P \cdot Q \Rightarrow P(Q) \\
\text{OPEN:} & \quad \nu a.b.P \Rightarrow P(b(x)) \\
\text{REPL:} & \quad \nu a.b.P \Rightarrow \nu a.b.Q \\
\text{INP:} & \quad a \cdot \nu a.b.P \Rightarrow \nu a.b.Q \\
\text{OUT:} & \quad \exists b.\ldots \Rightarrow P(b(x)) \\
\text{INP:} & \quad a \cdot \exists b.\ldots \Rightarrow P(Q) \\
\text{REPL:} & \quad P \Rightarrow \nu a.P \\
\text{COMP:} & \quad P \cdot Q \Rightarrow (P(a)Q) \\
\text{OPER:} & \quad \nu a.b.P \Rightarrow \nu a.b.Q \\
\text{PAR*:} & \quad P \cdot Q \Rightarrow P(Q) \\
\end{align*}

The Lambda Calculus

The Lambda Calculus was invented by Alonzo Church [1932] as a mathematical formalism for expressing computation by functions.

Syntax:
\[ M, N ::= x \mid \lambda x.M \mid M N \]

(Operational) Semantics:
- \( \alpha \)-conversion (renaming): \( \lambda x.M \leftrightarrow \lambda y.(y/x)M \) if \( y \notin \text{fn}(M) \)
- \( \beta \)-reduction (application): \( (\lambda x.M)N \rightarrow (N(x))M \)
- \( \eta \)-reduction: \( \lambda x. (M x) \rightarrow M \) if \( x \notin \text{fn}(M) \)

The lambda calculus can be viewed as the simplest possible pure functional programming language.
Standard Terms

\[ I = \lambda x.x \]
\[ K = \lambda x.\lambda y.x \]
\[ S = \lambda x.\lambda y.\lambda z.x \, z \, (y \, z) \]
\[ Y = \lambda f.(\lambda x.f \, (x \, x)) \, (\lambda x.f \, (x \, x)) \]
\[ \Omega = (\lambda x.x \, x) \, (\lambda x.x \, x) \]

Lambda-Calculus Reduction

- The call-by-value reduction relation \( \rightarrow_V \) over lambda terms is the smallest relation, which satisfies the rules:
  \[ \beta_V: (\lambda x.M) \, V \rightarrow_V \{V/x\}M, \text{ where } V \text{ does not contain a } \beta \text{-redex} \]

- The lazy reduction relation \( \rightarrow \) over lambda terms is the smallest relation, which satisfies \( \beta \), together with the rule:
  \[ \text{APPL: } M \, N \rightarrow M' \, N' \]
  \[ \text{APPR: } N \, M' \rightarrow N \, M' \]

Normal Form

- A lambda expression is in normal form if it can no longer be reduced by the \( \beta \)- or \( \eta \)-reduction rules.

- But not all lambda expressions have normal forms!

\[ \Omega = (\lambda x.x \, x) \, (\lambda x.x \, x) \rightarrow \{\lambda x.\, x \, x\}/(x \, x) \]
\[ = (\lambda x.x \, x) \, (\lambda x.x \, x) \rightarrow (\lambda x.x \, x) \, (\lambda x.x \, x) \rightarrow (\lambda x.x \, x) \, (\lambda x.x \, x) \rightarrow \ldots \]

- Reduction of a lambda expression to a normal form is analogous to the fact that a Turing machine halts or a program terminates.
The Church-Rosser Property

"If an expression can be evaluated at all, it can be evaluated by consistently using normal-order evaluation. If an expression can be evaluated in several different orders (mixing normal-order and applicative-order reduction), then all of these evaluation orders yield the same result."

So, evaluation order "does not matter" in the lambda calculus. However, applicative order reduction may not terminate, even if a normal form exists!

\[ (\lambda x . y) ((\lambda x . x x) (\lambda x . x x)) \]
\[ \Rightarrow (\lambda x . y) ((\lambda x . x x) (\lambda x . x x)) \]
\[ \Rightarrow (\lambda x . y) ((\lambda x . x x) (\lambda x . x x)) \]
\[ \Rightarrow y \ldots \]

Convergence

The reflexive and transitive closure of \( \rightarrow \) is \( \Rightarrow \).

A term \( M \) converges to \( M' \), written \( M \Downarrow M' \), if \( M \Rightarrow M' \).

If \( M \Downarrow M' \), then \( M' \) is a term that does not contain a \( \beta \)-redex anymore.

In the case of normal-order reduction (lazy reduction) this means that \( M' \) can only be an abstraction \( \lambda x . N \) or of the form \( x \ N_1 \ldots N_n \) (\( n \geq 0 \)).

In the case of applicative-order reduction (call-by-value reduction) this means that \( M' \) can only be an abstraction \( \lambda x . N \) or of the form \( M_1 \ldots M_n x \) (\( n \geq 0 \)).

Lambda-Calculus Encoding

The core of any encoding of the \( \lambda \)-calculus into a process calculus is the translation of function application, which becomes a particular form of parallel composition of two processes – the function (or abstraction) and its argument.

Beta-reduction is modeled as process interaction.

The syntax of the \( \lambda \)-calculus allows only for the communication of names along channels (or other names). Therefore, the communication of a lambda-calculus term is simulated by the communication of a trigger for it (see Executor).
Locations

- In the λ-calculus, λ is the only port. That is, a λ-term receives its argument at λ.
- In the π-calculus, there are infinitely many ports, so the encoding of a λ-term M is parametric over a port p, which can be thought of as the location of M.
  - The name p represents the unique port along which M interacts with its environment.
  - M receives two names along p:
    - The first is a trigger for its argument and
    - The second is the location to be used for the next interaction.

Milner’s Encoding of Λ_{lazy}

\{x\}(p) = \pi(p)

\{λx. M\}(p) = p(x, q).\{M\}(q)

\{M N\}(p) = \nu qr. x\{M\}(r) | r(x, p) | x(q), \{N\}(q), x \text{ fresh}

Intuition

M = N \Rightarrow \{M\}(p) \approx \{N\}(p)
Example

\[
\{(\lambda x.x) y(p) \\
= \{(\lambda z.z) y(p) \\
= (v r. x)\{(\lambda z.z)(r)\} | r(x, p) | l(x(q), y(q)) \\
= (v r. x)\{(\lambda z.q')(z)(q')\} | f(x, p) | l(x(q), y(q)) \\
= (v r. x)\{(\lambda z.q')(z)(q')\} | f(x, p) | l(x(q), y(q)) \\
= (v r. x)\{(\lambda z.q')(z)(q')\} | f(x, p) | l(x(q), y(q)) \\
\rightarrow (v r. x)\{(\lambda z.q')(z)(q')\} | f(x, p) | l(x(q), y(q)) \\
\rightarrow (v r. x)\{(\lambda z.q')(z)(q')\} | f(x, p) | l(x(q), y(q)) \\
= \{y\}(p)
\]

(\{\lambda x.x\} y(p) is weakly bisimilar to \{y\}(p)\)

Alternative Encoding of \(\Lambda_{\text{lazy}}\)

\[
\{x\}(p) = \chi(p) \\
(\lambda x. M)(p) = (v f)(f(f) | ff(x, q). (M) (q) \\
(M N)(p) = (v x, m)\{(M)(m) | m f, f(x, p) | l(x(q), (N)(q)) \}
\]

Example

\[
\{(\lambda x.x) y(p) \\
= \{(\lambda z.z) y(p) \\
= (v x, m)\{(\lambda z.z)(m)\} | m f, f(x, p) | l(x(q), (y)(q)) \\
= (v x, m)\{(v f)(f(f) | ff(z, q). ((z)(q)) \} | m f, f(x, p) | l(x(q), (y)(q)) \\
= (v x, m)\{(v f)(f(f) | ff(z, q). ((z)(q)) \} | m f, f(x, p) | l(x(q), (y)(q)) \\
\rightarrow (v x, m)\{(v f)(f(f) | ff(z, q). ((z)(q)) \} | m f, f(x, p) | l(x(q), (y)(q)) \\
\rightarrow (v x, m)\{(v f)(f(f) | ff(z, q). ((z)(q)) \} | m f, f(x, p) | l(x(q), (y)(q)) \\
\rightarrow (v x, m)\{(v f)(f(f) | ff(z, q). ((z)(q)) \} | m f, f(x, p) | l(x(q), (y)(q)) \\
= \{y\}(p)
\]
Encoding of Call-By-Value

Sequential call-by-value

Send value 'x' along channel 'p'.

\((x)(p) = \overrightarrow{f(x)}\)

\(\langle x . M \rangle(p) = (\nu f)(\nu f') \mid \{(x, q), \{M\}; q\})\)

\(\langle M N \rangle(p) = (\nu n)(\nu n') \mid \{(x, m), \{M\}; m\} \mid m(f, f(x, p)))\)

Example

Parallel Encoding of Call-By-Value

Parallel call-by-value

Send value 'x' along channel 'p'.

\((x)(p) = \overrightarrow{f(x)}\)

\(\langle x . M \rangle(p) = (\nu f)(\nu f') \mid \{(x, q), \{M\}; q\})\)

\(\langle M N \rangle(p) = (\nu m, n)(\nu n') \mid \{(M) ; n\} \mid n(x, m) \mid m(f, f(x, p)))\)
The Full Abstraction Problem

- An interpretation of the $\lambda$-calculus into the $\pi$-calculus, as a translation of one language into another, can be considered a form of denotational semantics.
- The denotation of a $\lambda$-term is an equivalence class of processes. These equivalence classes are the quotient of the $\pi$-calculus processes with respect to the behavioral equivalence of the $\pi$-calculus.
- An interpretation of a calculus is said to be sound if it equates only operationally equivalent terms, complete if it equates all operationally equivalent terms, and fully abstract if it is sound and complete.

Achieve Full Abstraction

- Not all interpretations are fully abstract.
- When an interpretation of a calculus is not fully abstract, one may hope to achieve full abstraction by
  - Enriching the calculus,
  - Choosing a finer notion of operational equivalence for the calculus, or
  - Cutting down the codomain of the interpretation.

Observation Equivalence

Let $M, N \in \Lambda^0$, the set of closed $\lambda$-terms. We say $M$ and $N$ are observationally equivalent if for all closed contexts $C$ it holds that $C[M] \Downarrow \Leftrightarrow C[N] \Downarrow$.
Applicative Bisimulation

A symmetric relation $R \subseteq \Lambda \times \Lambda$ is an applicative bisimulation if $M R N$ and $M \Rightarrow \lambda x. M'$ imply that there exists an $N'$ such that $N \Rightarrow \lambda x. N'$ and $\{L/x\}M' R \{L/x\}N'$ for all $L \in \Lambda$.

Two terms $M, N$ are applicative bisimilar, written $M \approx_N N$, if there exist some applicative bisimulation $R$ with $(M, N) \in R$.

Soundness

Our goal is to compare applicative bisimilarity with the equivalence on $\lambda$-terms induced by the encoding into the $\pi$-calculus.

Let $M, N \in \Lambda$. We write $M =_{\pi} N$ if $\{M\} = \{N\}$. We call $=_{\pi}$ the local structure of the $\pi$-interpretation.

Operational soundness:
Let $M, N \in \Lambda$. If $M =_{\pi} N$ then $M \approx_N N$.

Soundness is a necessary requirement for any interpretation.

Non-Completeness

Non-completeness:
Suppose $M, N \in \Lambda$. Then $M \approx_N N$ does not imply $M =_{\pi} N$.

Take $M = \lambda x. x x$ and $N = \lambda x. (x \lambda y. (x y))$, then it holds that $M \approx_N N$, but not $M =_{\pi} N$. 
$(\lambda x. x \ x) \ vs. \ (\lambda x. x \ (\lambda y. x \ y))$

\[
\{x, x \ x\}(p) \\
= p(x, q) \cdot \{x \ x\}(q) \\
= p(x, q) \cdot (\forall r, y)(\forall x, s)(y, q) | t(y, q) | y(s)(x)(s) \\
= p(x, q) \cdot (\forall r, y)(\forall x, s)(y, q) | t(y, q) | y(s)(x)(s) \quad \nu \text{(s)}
\]

\[
\{x, x \ y, x \ y\}(p) \\
= p(x, q) \cdot (\forall (x, y)) \cdot (q) \\
= p(x, q) \cdot (\forall r, y)(\forall x, s)(y, q) | t(y, q) | y(s)(x)(y)(s) \\
= p(x, q) \cdot (\forall r, y)(\forall x, s)(y, q) | t(y, q) | y(s)(x)(y)(s) \quad \nu \text{(q)}
\]

\[
(\forall x, z)(x, x \ x) \ | t(z, s) | t(z)(q') \ | q(q') \quad \nu \text{(q')}
\]