An introduction to the π-calculus

Michele Finelli
m@biodec.com
BioDec

Index

Index

Objectives

- Give an informal idea of what is a **calculus for concurrency**, what it looks like, and why it is useful.
- Support the idea that in twenty years (perhaps less) we will have a PiCon conference dedicated to concurrency languages, as we will have tomorrow (28 March 2015) a LambdaCon dedicated to functional languages.

Objectives

- Give an informal idea of what is a **calculus for concurrency**, what it looks like, and why it is useful.
- Support the idea that in twenty years (perhaps less) we will have a PiCon conference dedicated to concurrency languages, as we will have tomorrow (28 March 2015) a LambdaCon dedicated to functional languages.

Objectives

- Give an informal idea of what is a **calculus for concurrency**, what it looks like, and why it is useful.
- Support the idea that in twenty years (perhaps less) we will have a PiCon conference dedicated to concurrency languages, as we will have tomorrow (28 March 2015) a LambdaCon dedicated to functional languages.

Outline

- The example calculus will be the π-calculus, designed by Robin Milner in the nineties.
- To talk about concurrency we will first need to talk a little about *sequentiality*.
- So we will also have a little of λ-calculus too.

Outline

- The example calculus will be the π-calculus, designed by Robin Milner in the nineties.
- To talk about concurrency we will first need to talk a little about *sequentiality*.
- So we will also have a little of λ-calculus too.

Outline

- The example calculus will be the π-calculus, designed by Robin Milner in the nineties.
- To talk about concurrency we will first need to talk a little about *sequentiality*.
- So we will also have a little of λ-calculus too.

Outline

- The example calculus will be the π-calculus, designed by Robin Milner in the nineties.
- To talk about concurrency we will first need to talk a little about *sequentiality*.
- So we will also have a little of λ-calculus too.

Index

What is the λ-calculus ?

It is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution. (Wikipedia)

What is it used for ?

- It was born (Alonzo Church) to study the definability of functions.
- It was proved that it is Turing complete, so it became a model of computation (like *Turing Machines* etcetera).
- It entered computer science in the sixties and it is now fruitfully used alongside type theory to provide ideas and test beds for the theory of programming languages.

What is it used for ?

- It was born (Alonzo Church) to study the definability of functions.
- It was proved that it is Turing complete, so it became a model of computation (like *Turing Machines* etcetera).
- It entered computer science in the sixties and it is now fruitfully used alongside type theory to provide ideas and test beds for the theory of programming languages.

What is it used for ?

- It was born (Alonzo Church) to study the definability of functions.
- It was proved that it is Turing complete, so it became a model of computation (like *Turing Machines* etcetera).
- It entered computer science in the sixties and it is now fruitfully used alongside type theory to provide ideas and test beds for the theory of programming languages.

What is it used for ?

- It was born (Alonzo Church) to study the definability of functions.
- It was proved that it is Turing complete, so it became a model of computation (like *Turing Machines* etcetera).
- It entered computer science in the sixties and it is now fruitfully used alongside type theory to provide ideas and test beds for the theory of programming languages.

Formal definition

Remember:

> *It is a formal system in mathematical logic for expressing computation based on function* ***abstraction*** *and* ***application*** *using variable* ***binding*** *and* ***substitution****. (Wikipedia)*

Formal definition

Variables	A variable x is a valid λ-term.
Abstraction	If t is a λ-term, and x is a variable, then $(\lambda x.t)$ is a λ-term. Here x is the *bound* variable.
Application	If t and s are λ-terms then (ts) is a λ-term.

If all the variables are bound, the term is *closed*.
Usually λ-theories refer only to closed terms.

$\Lambda ::= x | \lambda x.\Lambda | \Lambda\Lambda$ for x in some set of variable names.

Formal definition

Variables A variable x is a valid λ-term.

Abstraction If t is a λ-term, and x is a variable, then $(\lambda x.t)$ is a λ-term. Here x is the *bound* variable.

Application If t and s are λ-terms then (ts) is a λ-term.
If all the variables are bound, the term is *closed*.
Usually λ-theories refer only to closed terms.

$\Lambda ::= x|\lambda x.\Lambda|\Lambda\Lambda$ for x in some set of variable names.

Formal definition

Variables	A variable x is a valid λ-term.
Abstraction	If t is a λ-term, and x is a variable, then $(\lambda x.t)$ is a λ-term. Here x is the *bound* variable.
Application	If t and s are λ-terms then (ts) is a λ-term.
	If all the variables are bound, the term is *closed*.
	Usually λ-theories refer only to closed terms.

$\Lambda ::= x|\lambda x.\Lambda|\Lambda\Lambda$ for x in some set of variable names.

Formal definition

Variables A variable x is a valid λ-term.

Abstraction If t is a λ-term, and x is a variable, then $(\lambda x.t)$ is a λ-term. Here x is the *bound* variable.

Application If t and s are λ-terms then (ts) is a λ-term.

If all the variables are bound, the term is *closed*. Usually λ-theories refer only to closed terms.

$\Lambda ::= x|\lambda x.\Lambda|\Lambda\Lambda$ for x in some set of variable names.

Formal definition

Variables A variable x is a valid λ-term.

Abstraction If t is a λ-term, and x is a variable, then $(\lambda x.t)$ is a λ-term. Here x is the *bound* variable.

Application If t and s are λ-terms then (ts) is a λ-term.
If all the variables are bound, the term is *closed*.
Usually λ-theories refer only to closed terms.

$\Lambda ::= x|\lambda x.\Lambda|\Lambda\Lambda$ for x in some set of variable names.

Formal definition

Variables	A variable x is a valid λ-term.
Abstraction	If t is a λ-term, and x is a variable, then $(\lambda x.t)$ is a λ-term. Here x is the *bound* variable.
Application	If t and s are λ-terms then (ts) is a λ-term. If all the variables are bound, the term is *closed*. Usually λ-theories refer only to closed terms.

$\Lambda ::= x|\lambda x.\Lambda|\Lambda\Lambda$ for x in some set of variable names.

Reductions

The *dynamics* of the calculus is captured by three reductions (*i.e. relations*) among terms.

Alpha two terms are α convertible if their differ in the names of their bound variables: $\lambda x \lambda y.x \xrightarrow{\alpha} \lambda t \lambda x.t$

Beta $(\lambda x.M)N \xrightarrow{\beta} M[N/x]$ Informally: $M[N/x]$ means *substituting N for x, in all of its occurrences in M*. The definition can be made rigorous.

Eta If x is not free in f then $\lambda x.(fx) \xrightarrow{\eta} f$

Reductions

The *dynamics* of the calculus is captured by three reductions (*i.e. relations*) among terms.

Alpha two terms are α convertible if their differ in the names of their bound variables: $\lambda x \lambda y.x \xrightarrow{\alpha} \lambda t \lambda x.t$

Beta $(\lambda x.M)N \xrightarrow{\beta} M[N/x]$ Informally: $M[N/x]$ means *substituting N for x, in all of its occurrences in M*. The definition can be made rigorous.

Eta If x is not free in f then $\lambda x.(fx) \xrightarrow{\eta} f$

Reductions

The *dynamics* of the calculus is captured by three reductions (*i.e. relations*) among terms.

Alpha two terms are α convertible if their differ in the names of their bound variables: $\lambda x \lambda y.x \xrightarrow{\alpha} \lambda t \lambda x.t$

Beta $(\lambda x.M)N \xrightarrow{\beta} M[N/x]$ Informally: $M[N/x]$ means *substituting N for x, in all of its occurrences in M*. The definition can be made rigorous.

Eta If x is not free in f then $\lambda x.(fx) \xrightarrow{\eta} f$

Reductions

The *dynamics* of the calculus is captured by three reductions (*i.e. relations*) among terms.

Alpha two terms are α convertible if their differ in the names of their bound variables: $\lambda x \lambda y.x \xrightarrow{\alpha} \lambda t \lambda x.t$

Beta $(\lambda x.M)N \xrightarrow{\beta} M[N/x]$ Informally: $M[N/x]$ means *substituting N for x, in all of its occurrences in M*. The definition can be made rigorous.

Eta If x is not free in f then $\lambda x.(fx) \xrightarrow{\eta} f$

Normal forms

- If a term M has a chain of β reductions to a term N than cannot be reduced anymore, N is said to be the *normal form* of M.
- Theorem: the normal form of a term, if it exists, is unique.
- Fact: there are terms which do not have a normal form.

Normal forms

- If a term M has a chain of β reductions to a term N than cannot be reduced anymore, N is said to be the *normal form* of M.
- Theorem: the normal form of a term, if it exists, is unique.
- Fact: there are terms which do not have a normal form.

Normal forms

- If a term M has a chain of β reductions to a term N than cannot be reduced anymore, N is said to be the *normal form* of M.
- Theorem: the normal form of a term, if it exists, is unique.
- Fact: there are terms which do not have a normal form.

Normal forms

- If a term M has a chain of β reductions to a term N than cannot be reduced anymore, N is said to be the *normal form* of M.
- Theorem: the normal form of a term, if it exists, is unique.
- Fact: there are terms which do not have a normal form.

Examples

- The identity $I = \lambda x.x$ is a term, which is already normal.
- Let T be the term $(\lambda x.\lambda y.xy)I$. Which is its normal form (if any ?) $(\lambda x.\lambda y.xy)I \xrightarrow{\beta} \lambda y.Iy \xrightarrow{\beta} \lambda y.y = I$
- What about the term $\Omega = (\lambda x.xx)\lambda x.xx$? It reduces to itself ... but Ω is not in normal form, since there is a redex that can be fired ... and on and on and on ...

$$\Omega \xrightarrow{\beta} \Omega \xrightarrow{\beta} \Omega \ldots$$

Examples

- The identity $I = \lambda x.x$ is a term, which is already normal.
- Let T be the term $(\lambda x.\lambda y.xy)I$. Which is its normal form (if any ?) $(\lambda x.\lambda y.xy)I \xrightarrow{\beta} \lambda y.Iy \xrightarrow{\beta} \lambda y.y = I$
- What about the term $\Omega = (\lambda x.xx)\lambda x.xx$? It reduces to itself ... but Ω is not in normal form, since there is a redex that can be fired ... and on and on and on ...

$$\Omega \xrightarrow{\beta} \Omega \xrightarrow{\beta} \Omega \ldots$$

Examples

- The identity $I = \lambda x.x$ is a term, which is already normal.
- Let T be the term $(\lambda x.\lambda y.xy)I$. Which is its normal form (if any ?) $(\lambda x.\lambda y.xy)I \xrightarrow{\beta} \lambda y.Iy \xrightarrow{\beta} \lambda y.y = I$
- What about the term $\Omega = (\lambda x.xx)\lambda x.xx$? It reduces to itself ... but Ω is not in normal form, since there is a redex that can be fired ... and on and on and on ...

$$\Omega \xrightarrow{\beta} \Omega \xrightarrow{\beta} \Omega \ldots$$

Examples

- The identity $I = \lambda x.x$ is a term, which is already normal.
- Let T be the term $(\lambda x.\lambda y.xy)I$. Which is its normal form (if any ?) $(\lambda x.\lambda y.xy)I \xrightarrow{\beta} \lambda y.Iy \xrightarrow{\beta} \lambda y.y = I$
- What about the term $\Omega = (\lambda x.xx)\lambda x.xx$? It reduces to itself ... but Ω is not in normal form, since there is a redex that can be fired ... and on and on and on ...

$$\Omega \xrightarrow{\beta} \Omega \xrightarrow{\beta} \Omega \ldots$$

Examples

- The identity $I = \lambda x.x$ is a term, which is already normal.
- Let T be the term $(\lambda x.\lambda y.xy)I$. Which is its normal form (if any ?) $(\lambda x.\lambda y.xy)I \xrightarrow{\beta} \lambda y.Iy \xrightarrow{\beta} \lambda y.y = I$
- What about the term $\Omega = (\lambda x.xx)\lambda x.xx$? It reduces to itself ... but Ω is not in normal form, since there is a redex that can be fired ... and on and on and on ...

$$\Omega \xrightarrow{\beta} \Omega \xrightarrow{\beta} \Omega \dots$$

Examples

- The identity $I = \lambda x.x$ is a term, which is already normal.
- Let T be the term $(\lambda x.\lambda y.xy)I$. Which is its normal form (if any ?) $(\lambda x.\lambda y.xy)I \xrightarrow{\beta} \lambda y.Iy \xrightarrow{\beta} \lambda y.y = I$
- What about the term $\Omega = (\lambda x.xx)\lambda x.xx$? It reduces to itself ... but Ω is not in normal form, since there is a redex that can be fired ... and on and on and on ...

$$\Omega \xrightarrow{\beta} \Omega \xrightarrow{\beta} \Omega \ldots$$

The λ-calculus captures sequentiality

- There are functions that cannot be expressed in the λ-calculus, the typical example is the *parallel-or* (*POR*):
  1. POR(x,1) = 1,
  2. POR(1,x) = 1,
  3. POR(0,0) = 0.
- If we give to POR two functions f and g that may either return 0 or 1, or that *may diverge* ...
- ... what should we do ? Should we compute first f or g ?
- Notice that if both f and g diverge, POR too **must** diverge.

The λ-calculus captures sequentiality

- There are functions that cannot be expressed in the λ-calculus, the typical example is the *parallel-or* (*POR*):
  1. POR(x,1) = 1,
  2. POR(1,x) = 1,
  3. POR(0,0) = 0.
- If we give to POR two functions f and g that may either return 0 or 1, or that *may diverge* . . .
- . . . what should we do ? Should we compute first f or g ?
- Notice that if both f and g diverge, POR too **must** diverge.

The λ-calculus captures sequentiality

- There are functions that cannot be expressed in the λ-calculus, the typical example is the *parallel-or* (*POR*):
  1. POR(x,1) = 1,
  2. POR(1,x) = 1,
  3. POR(0,0) = 0.
- If we give to POR two functions f and g that may either return 0 or 1, or that *may diverge* . . .
- . . . what should we do ? Should we compute first f or g ?
- Notice that if both f and g diverge, POR too **must** diverge.

The λ-calculus captures sequentiality

- There are functions that cannot be expressed in the λ-calculus, the typical example is the *parallel-or* (*POR*):
  1. POR(x,1) = 1,
  2. POR(1,x) = 1,
  3. POR(0,0) = 0.
- If we give to POR two functions f and g that may either return 0 or 1, or that *may diverge* ...
- ...what should we do ? Should we compute first f or g ?
- Notice that if both f and g diverge, POR too **must** diverge.

The λ-calculus captures sequentiality

- There are functions that cannot be expressed in the λ-calculus, the typical example is the *parallel-or* (*POR*):
  1. POR(x,1) = 1,
  2. POR(1,x) = 1,
  3. POR(0,0) = 0.
- If we give to POR two functions f and g that may either return 0 or 1, or that *may diverge* ...
- ...what should we do ? Should we compute first f or g ?
- Notice that if both f and g diverge, POR too **must** diverge.

Proof (informal)

- If POR could be expressed in the λ-calculus there should be a context $C[\cdot, \cdot]$ such that that $C[\Omega, \Omega]$ has no normal form, but $C[\Omega, I]$ and $C[I, \Omega]$ have normal forms, where $\Delta = \lambda x.xx$, $\Omega = \Delta\Delta$ and $I = \lambda x.x$.
- A theorem by Curry says that a normal form is always reached by the leftmost-outermost (normal) reduction.

Proof (informal)

- If POR could be expressed in the λ-calculus there should be a context $C[\cdot, \cdot]$ such that that $C[\Omega, \Omega]$ has no normal form, but $C[\Omega, I]$ and $C[I, \Omega]$ have normal forms, where $\Delta = \lambda x.xx$, $\Omega = \Delta\Delta$ and $I = \lambda x.x$.
- A theorem by Curry says that a normal form is always reached by the leftmost-outermost (normal) reduction.

Proof (informal)

- If POR could be expressed in the λ-calculus there should be a context $C[\cdot,\cdot]$ such that that $C[\Omega,\Omega]$ has no normal form, but $C[\Omega, I]$ and $C[I,\Omega]$ have normal forms, where $\Delta = \lambda x.xx$, $\Omega = \Delta\Delta$ and $I = \lambda x.x$.
- A theorem by Curry says that a normal form is always reached by the leftmost-outermost (normal) reduction.

Proof (continued)

So, we have three alternatives:

1. Either the normal reduction of $C[M, N]$ ignores M and N then $C[\Omega, \Omega]$ has a normal form — we do not want that.
2. Or the normal reduction starts with M, but this means that $C[\Omega, I]$ cannot have a normal form.
3. Or the normal reduction starts with N, but in this case it is $C[I, \Omega]$ that cannot have a normal form.

Conclusion: a context $C[\cdot, \cdot]$ that encodes POR in the λ-calculus does not exists.

Proof (continued)

So, we have three alternatives:

1. Either the normal reduction of $C[M, N]$ ignores M and N then $C[\Omega, \Omega]$ has a normal form — we do not want that.
2. Or the normal reduction starts with M, but this means that $C[\Omega, I]$ cannot have a normal form.
3. Or the normal reduction starts with N, but in this case it is $C[I, \Omega]$ that cannot have a normal form.

Conclusion: a context $C[\cdot, \cdot]$ that encodes POR in the λ-calculus does not exists.

Proof (continued)

So, we have three alternatives:

1. Either the normal reduction of $C[M, N]$ ignores M and N then $C[\Omega, \Omega]$ has a normal form — we do not want that.
2. Or the normal reduction starts with M, but this means that $C[\Omega, I]$ cannot have a normal form.
3. Or the normal reduction starts with N, but in this case it is $C[I, \Omega]$ that cannot have a normal form.

Conclusion: a context $C[\cdot, \cdot]$ that encodes POR in the λ-calculus does not exists.

Proof (continued)

So, we have three alternatives:

1. Either the normal reduction of $C[M, N]$ ignores M and N then $C[\Omega, \Omega]$ has a normal form — we do not want that.
2. Or the normal reduction starts with M, but this means that $C[\Omega, I]$ cannot have a normal form.
3. Or the normal reduction starts with N, but in this case it is $C[I, \Omega]$ that cannot have a normal form.

Conclusion: a context $C[\cdot, \cdot]$ that encodes POR in the λ-calculus does not exists.

Proof (continued)

So, we have three alternatives:

1. Either the normal reduction of $C[M, N]$ ignores M and N then $C[\Omega, \Omega]$ has a normal form — we do not want that.
2. Or the normal reduction starts with M, but this means that $C[\Omega, I]$ cannot have a normal form.
3. Or the normal reduction starts with N, but in this case it is $C[I, \Omega]$ that cannot have a normal form.

Conclusion: a context $C[\cdot, \cdot]$ that encodes POR in the λ-calculus does not exists.

Index

Concurrency

- To express parallelism we need something different from the λ-calculus. We need *concurrency*.
- Concurrency $\neq$ Parallelism.
- In an informal sense a system that manages concurrency also manages parallelism, the converse is not true.

Concurrency

- To express parallelism we need something different from the λ-calculus. We need *concurrency*.
- Concurrency $\neq$ Parallelism.
- In an informal sense a system that manages concurrency also manages parallelism, the converse is not true.

Concurrency

- To express parallelism we need something different from the λ-calculus. We need *concurrency*.
- Concurrency $\neq$ Parallelism.
- In an informal sense a system that manages concurrency also manages parallelism, the converse is not true.

Concurrency

- To express parallelism we need something different from the λ-calculus. We need *concurrency*.
- Concurrency $\neq$ Parallelism.
- In an informal sense a system that manages concurrency also manages parallelism, the converse is not true.

Names and actions

Names There exists an infinite set $\mathcal{N}$ of *names*. Lower letters x, y …range over $\mathcal{N}$.

Actions an *action* π is one of the following:

1. $x(y)$ receive y along x,
2. $\bar{x}\langle y\rangle$ send y along x,
3. τ the unobservable action.

Names and actions

Names There exists an infinite set $\mathcal{N}$ of *names*. Lower letters x, y ... range over $\mathcal{N}$.

Actions an *action* π is one of the following:

1. $x(y)$ receive y along x,
2. $\bar{x}\langle y\rangle$ send y along x,
3. τ the unobservable action.

Names and actions

Names There exists an infinite set $\mathcal{N}$ of *names*. Lower letters x, y … range over $\mathcal{N}$.

Actions an *action* π is one of the following:

1. $x(y)$ receive y along x,
2. $\bar{x}\langle y\rangle$ send y along x,
3. τ the unobservable action.

Processes

The set P^{π} of π-calculus process expressions is defined by the following syntax:

$$P ::= \sum_{i \in I} \pi_i . P_i \quad | \quad P_1 \mid P_2 \quad | \quad (\nu a)P \quad | \quad !P$$

where I is any *finite* indexing set.

Processes

Sums	The processes $\sum_{i \in I} \pi_i.P_i$ are called *summations* or *sums*.
Parallel	The operation $\cdot \mid \cdot$ is the *parallel composition*.
Restriction	The process $(\nu a)P$ has restricted the usage of a inside P.
Replication	The ! (*bang*) operator is the *replicator* operator.

Processes

Sums The processes $\sum_{i \in I} \pi_i.P_i$ are called *summations* or *sums*.

Parallel The operation $\cdot \mid \cdot$ is the *parallel composition*.

Restriction The process $(\nu a)P$ has restricted the usage of a inside P.

Replication The ! (*bang*) operator is the *replicator* operator.

Processes

Sums — The processes $\sum_{i \in I} \pi_i.P_i$ are called *summations* or *sums*.

Parallel — The operation $\cdot \mid \cdot$ is the *parallel composition*.

Restriction — The process $(\nu a)P$ has restricted the usage of a inside P.

Replication — The ! (*bang*) operator is the *replicator* operator.

Processes

Sums: The processes $\sum_{i \in I} \pi_i.P_i$ are called *summations* or *sums*.

Parallel: The operation $\cdot \mid \cdot$ is the *parallel composition*.

Restriction: The process $(\nu a)P$ has restricted the usage of a inside P.

Replication: The ! (*bang*) operator is the *replicator* operator.

Processes

Sums	The processes $\sum_{i \in I} \pi_i.P_i$ are called *summations* or *sums*.
Parallel	The operation $\cdot \mid \cdot$ is the *parallel composition*.
Restriction	The process $(\nu a)P$ has restricted the usage of a inside P.
Replication	The ! (*bang*) operator is the *replicator* operator.

Structural rules (informally)

We need some rules to equate different representations of the same term:

- $P \mid 0 \equiv P, P \mid Q \equiv Q \mid P, (P \mid Q) \mid R \equiv P \mid (Q \mid R)$
- $!P \equiv P \mid !P$
- And rules for α-conversion, rearrangement of terms, etcetera.

Structural rules (informally)

We need some rules to equate different representations of the same term:

- $P \mid 0 \equiv P, P \mid Q \equiv Q \mid P, (P \mid Q) \mid R \equiv P \mid (Q \mid R)$
- $!P \equiv P \mid !P$
- And rules for α-conversion, rearrangement of terms, etcetera.

Structural rules (informally)

We need some rules to equate different representations of the same term:

- $P \mid 0 \equiv P, P \mid Q \equiv Q \mid P, (P \mid Q) \mid R \equiv P \mid (Q \mid R)$
- $!P \equiv P \mid !P$
- And rules for α-conversion, rearrangement of terms, etcetera.

Structural rules (informally)

We need some rules to equate different representations of the same term:

- $P \mid 0 \equiv P, P \mid Q \equiv Q \mid P, (P \mid Q) \mid R \equiv P \mid (Q \mid R)$
- $!P \equiv P \mid !P$
- And rules for α-conversion, rearrangement of terms, etcetera.

Reactions (informally)

- The basic idea is that $x(y).P$ and $\bar{x}\langle z\rangle$ react in a way similar to β-reduction:

$$x(y).P \mid \bar{x}\langle z\rangle.Q \longrightarrow \{z/y\}P \mid Q$$

  $\{z/y\}P$ is basically what in the λ-calculus would have been written as $P[z/y]$.
- There is a precise notion of substitution, to avoid name capture.
- There are also *structural* rules, rules for the *parallel* operator, etcetera.

Reactions (informally)

- The basic idea is that $x(y).P$ and $\bar{x}\langle z\rangle$ react in a way similar to β-reduction:

  $$x(y).P \mid \bar{x}\langle z\rangle.Q \longrightarrow \{z/y\}P \mid Q$$

  $\{z/y\}P$ is basically what in the λ-calculus would have been written as $P[z/y]$.
- There is a precise notion of substitution, to avoid name capture.
- There are also *structural* rules, rules for the *parallel* operator, etcetera.

Reactions (informally)

- The basic idea is that $x(y).P$ and $\bar{x}\langle z\rangle$ react in a way similar to β-reduction:

  $$x(y).P \mid \bar{x}\langle z\rangle.Q \longrightarrow \{z/y\}P \mid Q$$

  $\{z/y\}P$ is basically what in the λ-calculus would have been written as $P[z/y]$.
- There is a precise notion of substitution, to avoid name capture.
- There are also *structural* rules, rules for the *parallel* operator, etcetera.

Reactions (informally)

- The basic idea is that $x(y).P$ and $\bar{x}\langle z\rangle$ react in a way similar to β-reduction:

  $$x(y).P \mid \bar{x}\langle z\rangle.Q \longrightarrow \{z/y\}P \mid Q$$

  $\{z/y\}P$ is basically what in the λ-calculus would have been written as $P[z/y]$.
- There is a precise notion of substitution, to avoid name capture.
- There are also *structural* rules, rules for the *parallel* operator, etcetera.

Index

Calculi for concurrency

- There are *many* calculi for concurrency, and concurrency is *way* less understood that sequentiality.
- Research started in the sixties (CSS, a precursor of the π-calculus (Milner), CSP (Hoare), the actor model (Hewitt), etcetera).
- The links with logical theories are less understood (*i.e.* there is no Curry-Howard Isomorphism, nor a good type system).

Calculi for concurrency

- There are *many* calculi for concurrency, and concurrency is *way* less understood that sequentiality.
- Research started in the sixties (CSS, a precursor of the π-calculus (Milner), CSP (Hoare), the actor model (Hewitt), etcetera).
- The links with logical theories are less understood (*i.e.* there is no Curry-Howard Isomorphism, nor a good type system).

Calculi for concurrency

- There are *many* calculi for concurrency, and concurrency is *way* less understood that sequentiality.
- Research started in the sixties (CSS, a precursor of the π-calculus (Milner), CSP (Hoare), the actor model (Hewitt), etcetera).
- The links with logical theories are less understood (*i.e.* there is no Curry-Howard Isomorphism, nor a good type system).

Calculi for concurrency

- There are *many* calculi for concurrency, and concurrency is *way* less understood that sequentiality.
- Research started in the sixties (CSS, a precursor of the π-calculus (Milner), CSP (Hoare), the actor model (Hewitt), etcetera).
- The links with logical theories are less understood (*i.e.* there is no Curry-Howard Isomorphism, nor a good type system).

A real necessity

- Nevertheless real system will be ever larger, ever more distributed and complex.
- A theory to reason about distributed concurrent systems will be necessary, sooner or later.
- And then new languages will follow.

A real necessity

- Nevertheless real system will be ever larger, ever more distributed and complex.
- A theory to reason about distributed concurrent systems will be necessary, sooner or later.
- And then new languages will follow.

A real necessity

- Nevertheless real system will be ever larger, ever more distributed and complex.
- A theory to reason about distributed concurrent systems will be necessary, sooner or later.
- And then new languages will follow.

A real necessity

- Nevertheless real system will be ever larger, ever more distributed and complex.
- A theory to reason about distributed concurrent systems will be necessary, sooner or later.
- And then new languages will follow.

Robin Milner, 13 January 1934 – 20 March 2010

Picture taken at CONCUR 09, the 20th International Conference on Concurrency Theory, in Bologna, 4 September 2009.

Questions

?

Thanks & see you soon ...

Thanks	for paying attention.
See you	tomorrow at ΛCON or ...
The biggest Italian DevOps meeting	is in Bologna, April 10th, 2015.
IDI2015	Incontro DevOps Italia 2015: `http://incontrodevops.it/idi2015/`

BIODEC
Evolving ICT Infrastructures

Thanks & see you soon . . .

Thanks	for paying attention.
See you	tomorrow at ΛCON or . . .
The biggest Italian DevOps meeting	is in Bologna, April 10^{th}, 2015.
IDI2015	Incontro DevOps Italia 2015: `http://incontrodevops.it/idi2015/`

Thanks & see you soon . . .

Thanks for paying attention.

See you tomorrow at ΛCON or . . .

The biggest Italian DevOps meeting is in Bologna, April 10th, 2015.

IDI2015 Incontro DevOps Italia 2015: http://incontrodevops.it/idi2015/

Thanks & see you soon . . .

Thanks for paying attention.

See you tomorrow at ΛCON or . . .

The biggest Italian DevOps meeting is in Bologna, April 10th, 2015.

IDI2015 Incontro DevOps Italia 2015: `http://incontrodevops.it/idi2015/`

BIODEC Evolving ICT Infrastructures

Thanks & see you soon . . .

Thanks	for paying attention.
See you	tomorrow at ΛCON or . . .
The biggest Italian DevOps meeting	is in Bologna, April 10$^{\text{th}}$, 2015.
IDI2015	Incontro DevOps Italia 2015: http://incontrodevops.it/idi2015/