# Formalising the $\pi$-calculus in Isabelle

Jesper Bengtson

Department of Computer Systems
University of Uppsala, Sweden

30th May 2006

## Overview

This talk will cover the following

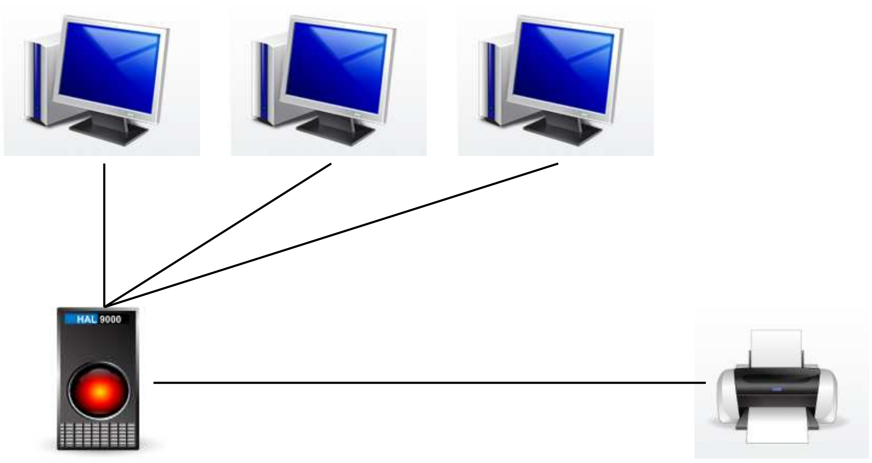
- Motivation – Why are we doing this?
- The $\pi$-calculus – A rundown on what the $\pi$-calculus actually is with a focus on the operational semantics.
- Nominal logic – A description of the nominal package we use to deal with binders.
- Formalisation in Isabelle – How do we create induction rules for nominal types?
- Formalisation in Isabelle – The proof strategies used to do proofs in Isabelle regarding simulation and bisimulation.
- Conclusions and future work – what have we learnt and what remains to be done.

## Motivation

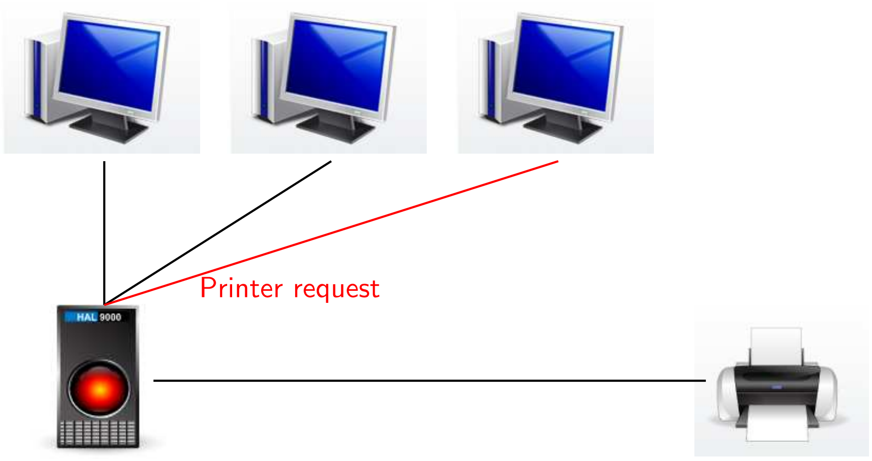
Why formalise the $\pi$-calculus? Has this not been done before?

- The $\pi$-calculus has been formalised in Isabelle and Coq.
- Binders have been dealt with either by using HOAS or de-Bruijn indices.
- The formalisations to date have flaws which make them cumbersome to work with.
- There are automatic tools available to reason about actual processes, like the Mobility Workbench - it can only reason about agents with finite state space, however. It is not verified.
- We want to create a library of lemmas and tactics to reason about process calculi in Isabelle. The amount of work for the end user is, to date, far too much.

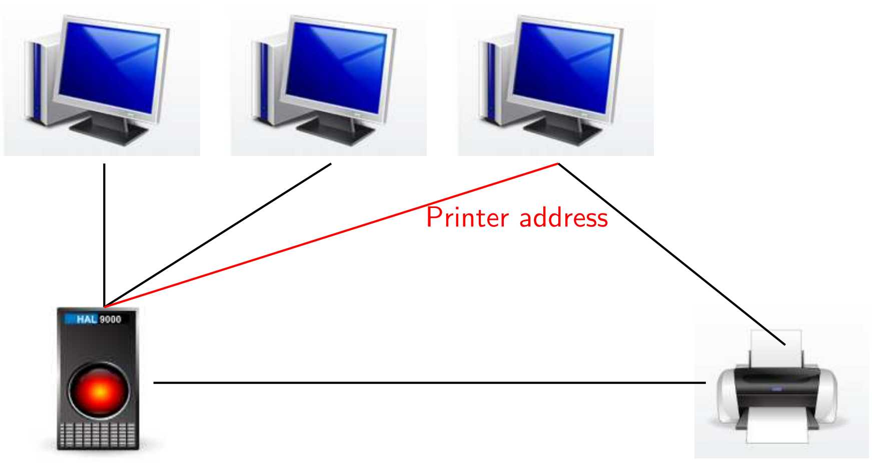
## A simple distributed system



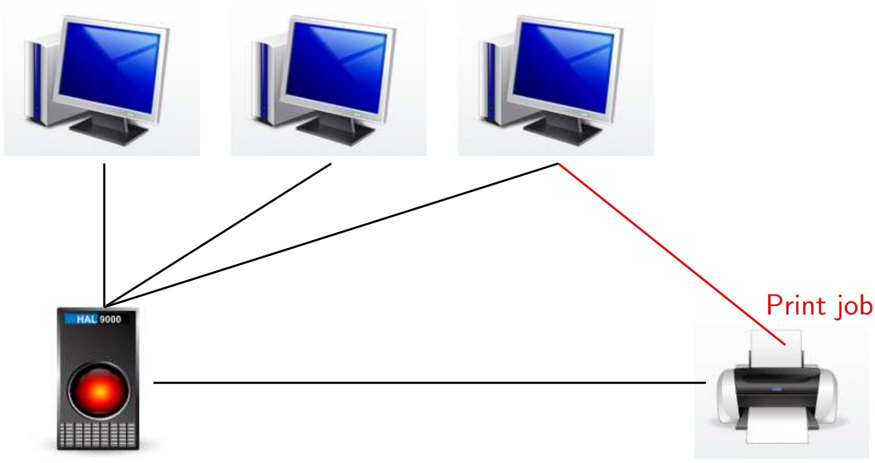
## A simple distributed system



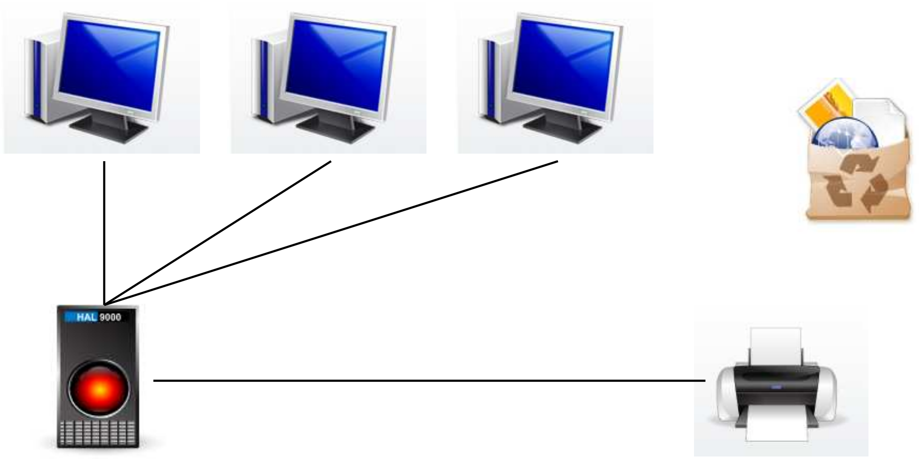
## A simple distributed system



## A simple distributed system



## A simple distributed system



## The $\pi$-calculus - a calculus of mobile processes

- The $\pi$-calculus was created by Robin Milner, Joachim Parrow and David Walker.
- It is used to model communication between processes (agents).
- Since its conception, the *pi*-calculus has had many spin-offs – The Spi-calculus, the Fusion calculus, typed and higher order $\pi$-calculus just to name a few.

## Notation

The following notation will be used throughout this talk.

- Processes, often called *agents* will be denoted $P$, $Q$, $R$.
- Actions that the agents perform are denoted $\alpha$, $\beta$, $\gamma$.
- free names will be denoted $a$, $b$, $c$
- bound names will be denoted $x$, $y$, $z$.
- The state of a process after an action is called a *derivative* and is denoted $P'$, $Q'$, $R'$.
- A process $P$ doing an action $\alpha$ to the derivative $P'$ will be denoted $P \xrightarrow{\alpha} P'$.$P'$ is also said to be an $\alpha$-derivative of $P$.
- Substitution is denoted $P\{a/b\}$ which can be read *"b for a"* – all free occurrances of $b$ in $P$ are replaced by $a$.
- The free names of a process can be obtained through the function fn – fn($P$) represents the free names of the process $P$.

## Syntax

The following is the syntax of the *pi*-calculus.

**Prefixes**

$$\begin{aligned} \pi_p &= a(x) \\ &\mid \bar{a}b \\ &\mid \tau \end{aligned}$$

**Processes**

$$\begin{aligned} \pi &= 0 \\ &\mid \pi_p.\pi \\ &\mid [a = b]\pi \\ &\mid \pi + \pi \\ &\mid \pi \mid \pi \\ &\mid (\nu x)\pi \\ &\mid !\pi \end{aligned}$$

## Prefixes

There are three types of prefixes in the $\pi$-calculus.

### Input

$a(x).P \xrightarrow{a(x)} P$

The process $a(x).P$ can receive a name $x$ over the channel $a$.

### Output

$\bar{a}b.P \xrightarrow{\bar{a}b} P$

The process $\bar{a}b.P$ can output the name $b$ over the channel $a$.

### Silent action

$\tau.P \xrightarrow{\tau} P$

The process $\tau.P$ can perform a $\tau$-action, often called a silent action.

## Actions

The actions of a process are closely linked to the prefixes. They appear over the transition arrows. The actions available are:

**Actions**

- Input: $a(x)$
- Output: $\bar{a}b$
- Tau: $\tau$

Input and output have some similarities to the actions and co-actions of CCS.

## Actions cont. - Free and bound names

The names in an action can be grouped into free and bound names according to the following scheme:

| | fn | bn | n |
|---|---|---|---|
| $a(x)$ | $\{a\}$ | $\{x\}$ | $\{a, x\}$ |
| $\bar{a}b$ | $\{a, b\}$ | $\{\}$ | $\{a, b\}$ |
| $\tau$ | $\{\}$ | $\{\}$ | $\{\}$ |

There is one more type of action which will be covered later.

## Match

An equality test can be added as a guard for a transition.

### Match

$$\frac{P \xrightarrow{\alpha} P'}{[a = a]P \xrightarrow{\alpha} P'}$$

Intuitively, $[a = b]P$ can only do $\alpha$ and end up in $P'$ only if $a = b$.

## Nondeterministic choice

The $+$-operator can be used to encode nondeterministic choice.

**Sum**

$$\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}$$

## Parallel composition

Processes running in parallel can be encoded with the $|$-operator.

**Par**

$$\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \quad \forall x \in \mathrm{bn}(\alpha).\ x \notin \mathrm{fn}(Q)$$

$$\frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'} \quad \forall x \in \mathrm{bn}(\alpha).\ x \notin \mathrm{fn}(P)$$

**Comm**

$$\frac{P \xrightarrow{a(x)} P' \quad Q \xrightarrow{\bar{a}b} Q'}{P \mid Q \xrightarrow{\tau} P'\{b/x\} \mid Q'} \qquad \frac{P \xrightarrow{\bar{a}b} P' \quad Q \xrightarrow{a(x)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'\{b/x\}}$$

## Restriction

The $\nu$-operator can be used to bind a name to a specific process. This name will be unique for this process and may not occur free in any other.

**Res**

$$\frac{P \xrightarrow{\alpha} P'}{(\nu x)P \xrightarrow{\alpha} (\nu x)P'} \quad x \notin \mathrm{n}(\alpha)$$

## The Bang-operator

The !-operator is used to model a process running in parallel with itself infinitely many times.

**Bang**

$$\frac{P \mid\, !P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}$$

## Scope extrusion

So far we have omitted one type of action – the bound output. When restricting a name, that name is considered local to that process only. What happens if we output this name?

**Open**

$$\frac{P \xrightarrow{\bar{a}x} P'}{(\nu x)P \xrightarrow{\bar{a}\nu x} P'} \quad a \neq x$$

**Close**

$$\frac{P \xrightarrow{a(x)} P' \quad Q \xrightarrow{\bar{a}\nu y} Q'}{P \mid Q \xrightarrow{\tau} (\nu y)(P'\{y/x\} \mid Q')} \quad y \notin \mathrm{fn}(P)$$

$$\frac{P \xrightarrow{\bar{a}\nu y} P' \quad Q \xrightarrow{a(x)} Q'}{P \mid Q \xrightarrow{\tau} (\nu y)(P' \mid Q'\{y/x\})} \quad y \notin \mathrm{fn}(Q)$$

## Actions again - Free and bound names

We can now complete the table we had earlier.

|  | fn | bn | n |
|---|---|---|---|
| $a(x)$ | $\{a\}$ | $\{x\}$ | $\{a, x\}$ |
| $\bar{a}b$ | $\{a, b\}$ | $\{\}$ | $\{a, b\}$ |
| $\bar{a}\nu x$ | $\{a\}$ | $\{x\}$ | $\{a, x\}$ |
| $\tau$ | $\{\}$ | $\{\}$ | $\{\}$ |

## Structural congruence

There are structural congruence rules for the $\pi$-calculus. These are equivalences which we intuitively know to be true.

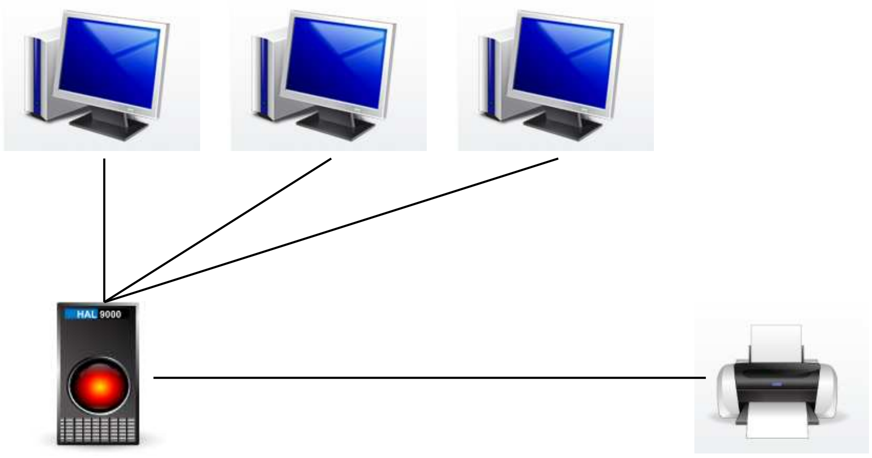
- Rules for the $+$-operator
  - $P + 0 \equiv P$
  - $P + Q \equiv Q + P$
  - $P + (Q + R) \equiv (P + Q) + R$
- Corresponding rules for the $|$-operator
  - $P \mid 0 \equiv P$
  - $P \mid Q \equiv Q \mid P$
  - $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$

## Structural congruence cont. – scope extrusion

The more complex structural congruence rules revolve around restriction.

- $(\nu x)P \equiv P$ $\quad$ *if* $x \notin \mathrm{fn} P$
- $(\nu x)(P + Q) \equiv P + (\nu x)Q$ $\quad$ *if* $x \notin \mathrm{fn} P$
- $(\nu x)(P \mid Q) \equiv P \mid (\nu x)Q$ $\quad$ *if* $x \notin \mathrm{fn} P$

## Recall our system



## Modeling our system

We need to model the the printer, the printer server and the clients in order to model the complete system.

### The printer

$P \equiv a(x).\overline{print}\ x.P$

## Modeling our system

We need to model the the printer, the printer server and the clients in order to model the complete system.

**The printer**

$P \equiv a(x).\overline{print}\ x.P$

**The printer server**

$S \equiv b(y).\bar{y}a.S$

## Modeling our system

We need to model the the printer, the printer server and the clients in order to model the complete system.

**The printer**

$P \equiv a(x).\overline{print}\ x.P$

**The printer server**

$S \equiv b(y).\bar{y}a.S$

**The client**

$C \equiv (\nu z)\bar{b}z.z(q).\bar{q}\ job.C$

## Modeling our system

We need to model the the printer, the printer server and the clients in order to model the complete system.

**The printer**

$P \equiv a(x).\overline{print}\ x.P$

**The printer server**

$S \equiv b(y).\bar{y}a.S$

**The client**

$C \equiv (\nu z)\bar{b}z.z(q).\bar{q}\ job.C$

**The system**

$Sys \equiv C \mid C \mid C \mid \qquad S \mid P$

## Modeling our system

We need to model the the printer, the printer server and the clients in order to model the complete system.

**The printer**

$P \equiv a(x).\overline{print}\ x.P$

**The printer server**

$S \equiv b(y).\bar{y}a.S$

**The client**

$C \equiv (\nu z)\bar{b}z.z(q).\bar{q}\ job.C$

**The system**

$Sys \equiv C \mid C \mid C \mid (\nu a)(S \mid P)$

## A sample run

$$P \equiv a(x).\overline{print}\ x.P \qquad S \equiv b(y).\bar{y}a.S \qquad C \equiv (\nu z)\bar{b}z.z(q).q(job).C$$

$$Sys \equiv \quad C \mid C \mid C \mid (\nu a)(S \mid P)$$

## A sample run

$$P \equiv a(x).\overline{print}\ x.P \qquad S \equiv b(y).\bar{y}a.S \qquad C \equiv (\nu z)\bar{b}z.z(q).q(job).C$$

$$Sys \equiv \quad C \mid C \mid C \mid (\nu a)(S \mid P) \xrightarrow{\tau}$$

$$C \mid C \mid (\nu z)(z(q).\bar{q}\ job.C \mid (\nu a)(\bar{z}a.S \mid P))$$

## A sample run

$$P \equiv a(x).\overline{print}\ x.P \qquad S \equiv b(y).\bar{y}a.S \qquad C \equiv (\nu z)\bar{b}z.z(q).q(job).C$$

$$\begin{aligned} Sys \equiv\quad & C \mid C \mid C \mid (\nu a)(S \mid P) \xrightarrow{\tau} \\ & C \mid C \mid (\nu z)(z(q).\bar{q}\ job.C \mid (\nu a)(\bar{z}a.S \mid P)) \xrightarrow{\tau} \\ & C \mid C \mid (\nu z)(\nu a)(\bar{a}\ job.C \mid S \mid P) \end{aligned}$$

## A sample run

$$P \equiv a(x).\overline{print}\ x.P \qquad S \equiv b(y).\bar{y}a.S \qquad C \equiv (\nu z)\bar{b}z.z(q).q(job).C$$

$$\begin{aligned}
Sys \equiv\ & C \mid C \mid C \mid (\nu a)(S \mid P) \xrightarrow{\tau} \\
& C \mid C \mid (\nu z)(z(q).\bar{q}\ job.C \mid (\nu a)(\bar{z}a.S \mid P)) \xrightarrow{\tau} \\
& C \mid C \mid (\nu z)(\nu a)(\bar{a}\ job.C \mid S \mid P) \xrightarrow{\tau} \\
& C \mid C \mid (\nu z)(\nu a)(C \mid S \mid \overline{print}\ job.P)
\end{aligned}$$

## A sample run

$$P \equiv a(x).\overline{print}\ x.P \qquad S \equiv b(y).\bar{y}a.S \qquad C \equiv (\nu z)\bar{b}z.z(q).q(job).C$$

$$\begin{aligned}
Sys \equiv\quad & C \mid C \mid C \mid (\nu a)(S \mid P) \xrightarrow{\tau} \\
& C \mid C \mid (\nu z)(z(q).\bar{q}\ job.C \mid (\nu a)(\bar{z}a.S \mid P)) \xrightarrow{\tau} \\
& C \mid C \mid (\nu z)(\nu a)(\bar{a}\ job.C \mid S \mid P) \xrightarrow{\tau} \\
& C \mid C \mid (\nu z)(\nu a)(C \mid S \mid \color{red}{\overline{print}\ job}\color{black}{.P}) \xrightarrow{\overline{print}\ job} \\
& C \mid C \mid (\nu z)(\nu a)(C \mid S \mid P)
\end{aligned}$$

## A sample run

$$P \equiv a(x).\overline{print}\ x.P \qquad S \equiv b(y).\bar{y}a.S \qquad C \equiv (\nu z)\bar{b}z.z(q).q(job).C$$

$$\begin{aligned}
Sys \equiv\ & C \mid C \mid C \mid (\nu a)(S \mid P) \xrightarrow{\tau} \\
& C \mid C \mid (\nu z)(z(q).\bar{q}\ job.C \mid (\nu a)(\bar{z}a.S \mid P)) \xrightarrow{\tau} \\
& C \mid C \mid (\nu z)(\nu a)(\bar{a}\ job.C \mid S \mid P) \xrightarrow{\tau} \\
& C \mid C \mid (\nu z)(\nu a)(C \mid S \mid \overline{print}\ job.P) \xrightarrow{\overline{print}\ job} \\
& C \mid C \mid (\nu z)(\nu a)(C \mid S \mid P) \equiv \\
& C \mid C \mid (\nu a)(C \mid S \mid P)
\end{aligned}$$

## A sample run

$$P \equiv a(x).\overline{print}\ x.P \qquad S \equiv b(y).\bar{y}a.S \qquad C \equiv (\nu z)\bar{b}z.z(q).q(job).C$$

$$\begin{aligned}
Sys \equiv\quad & C \mid C \mid C \mid (\nu a)(S \mid P) \xrightarrow{\tau} \\
& C \mid C \mid (\nu z)(z(q).\bar{q}\ job.C \mid (\nu a)(\bar{z}a.S \mid P)) \xrightarrow{\tau} \\
& C \mid C \mid (\nu z)(\nu a)(\bar{a}\ job.C \mid S \mid P) \xrightarrow{\tau} \\
& C \mid C \mid (\nu z)(\nu a)(C \mid S \mid \overline{print}\ job.P) \xrightarrow{\overline{print}\ job} \\
& C \mid C \mid (\nu z)(\nu a)(C \mid S \mid P) \equiv \\
& C \mid C \mid (\nu a)(C \mid S \mid P) \equiv C \mid C \mid C \mid (\nu a)(S \mid P)
\end{aligned}$$

## A sample run

$$P \equiv a(x).\overline{print}\ x.P \qquad S \equiv b(y).\bar{y}a.S \qquad C \equiv (\nu z)\bar{b}z.z(q).q(job).C$$

$$\begin{aligned}
Sys \equiv \quad & C \mid C \mid C \mid (\nu a)(S \mid P) \xrightarrow{\tau} \\
& C \mid C \mid (\nu z)(z(q).\bar{q}\ job.C \mid (\nu a)(\bar{z}a.S \mid P)) \xrightarrow{\tau} \\
& C \mid C \mid (\nu z)(\nu a)(\bar{a}\ job.C \mid S \mid P) \xrightarrow{\tau} \\
& C \mid C \mid (\nu z)(\nu a)(C \mid S \mid \overline{print}\ job.P) \xrightarrow{\overline{print}\ job} \\
& C \mid C \mid (\nu z)(\nu a)(C \mid S \mid P) \equiv \\
& C \mid C \mid (\nu a)(C \mid S \mid P) \equiv C \mid C \mid C \mid (\nu a)(S \mid P) \equiv Sys
\end{aligned}$$

## A recapitulation

So far we have covered:

- What is the $\pi$-calculus?
- Operational semantics
- A *relatively* simple example

The next part of this talk will cover:

- How do we incorporate this into a theorem prover?
- What technical difficulties are there?
- Why choose Isabelle out of all the available theorem provers?

## Isabelle

Isabelle is an interactive theorem prover.

- It has support for first order logic (FOL), higher order logic (HOL) and Zermaelo Fraenkel (ZF) set theory.
- It uses classical logic.
- It is extensively used in the formalisation of large *real life* problems.
- It has support for writing proofs which are readable by humans!

## $\alpha$-equivalence

$\alpha$-equivalence is a key concept in process calculi. Intuitively, the names of the bound variables are unimportant.

**Two $\alpha$-equivalent processes**

$$a(x).\bar{x}b.0 \quad =_\alpha \quad a(y).\bar{y}b.0$$

Since $y$ does not occur free in $a(x).\bar{x}a.0$, all occurrances of $x$ can be substituted for $y$ and the resulting processes are $\alpha$-equivalent.

## $\alpha$-equivalence

$\alpha$-equivalence is a key concept in process calculi. Intuitively, the names of the bound variables are unimportant.

**Two $\alpha$-equivalent processes**

$$a(x).\bar{x}b.0 \quad =_\alpha \quad a(y).\bar{y}b.0$$

Since $y$ does not occur free in $a(x).\bar{x}a.0$, all occurrances of $x$ can be substituted for $y$ and the resulting processes are $\alpha$-equivalent.

The following two processes, however, are not $\alpha$-equivalent.

**Two non $\alpha$-equivalent processes**

$$a(x).\bar{x}b.0 \quad \neq_\alpha \quad a(b).\bar{b}b.0$$

## $\alpha$-equivalence

$\alpha$-equivalence is a key concept in process calculi. Intuitively, the names of the bound variables are unimportant.

**Two $\alpha$-equivalent processes**

$$a(x).\bar{x}b.0 \quad =_\alpha \quad a(y).\bar{y}b.0$$

Since $y$ does not occur free in $a(x).\bar{x}a.0$, all occurrances of $x$ can be substituted for $y$ and the resulting processes are $\alpha$-equivalent.

The following two processes, however, are not $\alpha$-equivalent.

**Two non $\alpha$-equivalent processes**

$$a(x).\bar{x}b.0 \quad \neq_\alpha \quad a(b).\bar{b}b.0$$

Any process has infinitely many $\alpha$-equivalent processes!

## $\alpha$-equivalence in literature

When doing non-formal proofs there is usually a lot of hand-waving regarding $\alpha$-equivalence.

**The $\pi$-calculus, *Sangiorgi* and *Walker***

In any discussion, we assume tha the bound names of any processes or actions under consideration are chosen to be different from the names free in any other entities under consideration, such as processes, actions, substitutions and sets of names.

**An introduction to the $\pi$-calculus, *Parrow***

... we will use the phrase “$bn(\alpha)$ is fresh” in a definition to mean that the name in $bn(\alpha)$, if any, is different from any free name occurring in any of the agents in the definition.

## $\alpha$-equivalence in theorem provers

This type of hand waving does not work well with theorem provers and methods for dealing with binders in formalisations have been proposed – most notably de-Bruijn indices and HOAS.

We have adopted a new approach using nominal logic and the new package which has been incorporated into Isabelle.

## Nominal logic

Nominal logic was designed in Cambridge by Andy Pitts, Jamie Gabbay and Christian Urban.

- Alpha equivalence is dealt with using permutations of names.
- Nominal logic has some very nice mathematical properties, which will be covered later.
- A nominal package is being implemented in Isabelle which provides extensive support for working with nominal logic.
- So far, the nominal package has been used to formalise the $\lambda$-calculus (Urban), SystemF (Urban) and the *pi*-calculus (Bengtson).

## Permutations

Permutation is denoted $p \bullet P$ where $p$ is a permutation and $P$ is an agent.

A single permutation is written $(a\ b) \bullet P$ where $a$ and $b$ are the names that are to be permuted in $P$.

### Equivariance

A relation $\mathcal{R}$ is said to be equivariant if it is closed under permutations. More formally:

$$\forall a\ b\ p.\ (a, b) \in \mathcal{R} \longrightarrow (p \bullet a, p \bullet b) \in \mathcal{R}$$

We will find that all of our relations are equivariant, which is crucial.

## Permutations on $\pi$-calculus agents

### Permutations on names

$(a\ b) \bullet a = b$

$(a\ b) \bullet b = a$

$(a\ b) \bullet c = c \quad \textit{iff } c \sharp (a, b)$

### Permutations on $\pi$-calculus agents

$$
\begin{aligned}
p \bullet \tau.P &= \tau.(p \bullet P) \\
p \bullet a(x).P &= p \bullet a(p \bullet x).(p \bullet P) \\
p \bullet \bar{a}b.P &= \overline{p \bullet a}(p \bullet b).(p \bullet P) \\
p \bullet [a = b]P &= [(p \bullet a) = (p \bullet b)](p \bullet P) \\
p \bullet P + Q &= (p \bullet P) + (p \bullet Q) \\
p \bullet P \mid Q &= (p \bullet P) \mid (p \bullet Q) \\
p \bullet (\nu x)P &= (\nu(p \bullet x))(p \bullet P) \\
p \bullet !P &= !(p \bullet P)
\end{aligned}
$$

## Support and freshness

The support of an agent is the free names of the agent. More formally:

**Support**

$supp\ P = \{a \mid inf\{b \mid (a\ b) \bullet P \neq P\}\}$

The definition for freshness then becomes trivial.

**Freshness**

$a \sharp P = a \notin supp\ P$

## Permutations and $\alpha$-equivalence

$\alpha$-equivalence can be defined using permutations. In the pi-calculus, we have two cases where binders occur in the syntax – Input and Restriction.

**$\alpha$-equivalence on restriction**

$$(\nu x)P =_\alpha (\nu y)Q \textit{ iff either } \begin{array}{l} x = y \wedge P = Q \textit{ or} \\ x \neq y \wedge P = (x\ y) \bullet Q \wedge x \sharp Q \end{array}$$

**$\alpha$-equivalence on input-actions**

$$a(x)P =_\alpha b(y)Q \textit{ iff } a = b \textit{ and either } \begin{array}{l} x = y \wedge P = Q \textit{ or} \\ x \neq y \wedge P = (x\ y) \bullet Q \wedge \\ x \sharp Q \end{array}$$

From now on $=$ on agents will denote $\alpha$-equivalence.

## Defining the $\pi$-calculus using the nominal package

The datatype for the $\pi$-calculus is defined in Isabelle as follows.

**The $\pi$-datatype**

```
atom_decl name

nominal_datatype pi = PiNil
                    | Tau pi
                    | Input name ‘‘<<name>> pi’’
                    | Output name name pi
                    | Match name name pi
                    | Sum pi pi
                    | Par pi pi
                    | Res ‘‘<<name>> pi’’
                    | Bang pi
```

## Creating the operational semantics

Creating operational semantics in Isabelle is best done using inductively defined sets. Intuitively, if the tuple $(P, \alpha, P')$ is in the set, then $P \xrightarrow{\alpha} P'$ is a valid transition.

This causes problems, however as the action is split from the derivatives along with any bound names they share.

Consider the transition $a(x).P \xrightarrow{a(x)} P$. The x needs to be bound in P until a communication is done. If we split them up, as in the example above, we lose the ability to $\alpha$-convert.

## The residual datatype

We create the notion of a *residual*. A residual is the result of the transition, but with the action still bound to the derivative. The action $P \xrightarrow{\alpha} P'$, is written $P \longmapsto \alpha \prec P'$.

When looking at rules for the operational semantics, the old notation will be used. However, the residual notation will appear in Isabelle code.

The operational semantics is thus a set of type $(\mathit{pi} \times \mathit{residual})\ \mathit{set}$.

## Defining the residual datatype

### The subject datatype

```
datatype subject = InputS name
                 | BoundOutputS name
```

### The freeRes datatype

```
datatype freeRes = OutputR name name
                 | TauR
```

### The residual datatype

```
nominal_datatype residual = BoundR subject ‘‘<<name>>
pi’’
                          | FreeR freeRes pi
```

## Structural induction

When defining the operational semantics of a calculus, Isabelle automatically creates the rules for induction and structural induction.

These automated rules, however, assume equivalence to be syntactic equivalence and not $\alpha$-equivalence, and this makes them tedious to work with.

## Structural induction – a simple example

Recall the rules for the $+$-operator:

**Nondeterministic choice**

$$\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}$$

Isabelle will from these create the following rule:

**Isabelle's rule for structural induction**

$$\frac{\begin{array}{c} P + Q \longmapsto Rs \\ \forall P' \; Q' \; . \; P' \longmapsto Rs \wedge P + Q = P' + Q' \longrightarrow Pred \\ \forall P' \; Q' \; . \; Q' \longmapsto Rs \wedge Q + Q = P' + Q' \longrightarrow Pred \end{array}}{Pred}$$

Improving structural induction on the $+$-operator.

**Isabelle's rule for structural induction**

$$\frac{\begin{array}{c} P + Q \longmapsto Rs \\ \forall P'\ Q'\ .\ P' \longmapsto Rs \wedge P + Q = P' + Q' \longrightarrow Pred \\ \forall P'\ Q'\ .\ Q' \longmapsto Rs \wedge Q + Q = P' + Q' \longrightarrow Pred \end{array}}{Pred}$$

This rule is about as simple as it can get. We can, however, polish it a bit.

Improving structural induction on the $+$-operator.

**Isabelle's rule for structural induction**

$$\frac{\begin{array}{c} P + Q \longmapsto Rs \\ \forall P'\ Q'\ .\ P' \longmapsto Rs \wedge P + Q = P' + Q' \longrightarrow Pred \\ \forall P'\ Q'\ .\ Q' \longmapsto Rs \wedge Q + Q = P' + Q' \longrightarrow Pred \end{array}}{Pred}$$

This rule is about as simple as it can get. We can, however, polish it a bit.

**Derived structural induction rule**

$$\frac{\begin{array}{c} P + Q \longmapsto Rs \wedge \\ P \longmapsto Rs \longrightarrow Pred \\ Q \longmapsto Rs \longrightarrow Pred \end{array}}{Pred}$$

## Structural induction – Parallel composition

We concentrate on communication.

**Comm**

$$\frac{P \xrightarrow{a(x)} P' \qquad Q \xrightarrow{\bar{a}b} Q'}{P \mid Q \xrightarrow{\tau} P'\{b/x\} \mid Q'}$$

This rule is mapped to the following induction rule:

**Isabelle's rule for structural induction**

$$\frac{\begin{array}{ll} & P \mid Q \longmapsto \alpha \prec PQ' \\ \forall P'\ Q'\ P''\ Q''\ a\ b\ x. & P' \longmapsto a(x) \prec P'' \wedge Q' \longmapsto \bar{a}b \prec Q'' \wedge \\ & P \mid Q = P' \mid Q' \wedge \\ & \alpha \prec PQ' = \tau \prec P''\{b/x\} \mid Q'' \\ & \longrightarrow Pred \end{array}}{Pred}$$

## Improving structural induction on the |-operator.

### Isabelle's rule for structural induction

$$\frac{\begin{array}{l} \qquad\qquad\qquad P \mid Q \longmapsto \alpha \prec PQ' \\ \forall P'\ Q'\ P''\ Q''\ a\ b\ \color{red}{x}. \quad P' \longmapsto a(\color{red}{x}) \prec P'' \wedge Q' \longmapsto \bar{a}b \prec Q'' \wedge \\ \qquad\qquad\qquad P \mid Q = P' \mid Q' \wedge \\ \qquad\qquad\qquad \alpha \prec PQ' = \tau \prec P''\{b/x\} \mid Q'' \\ \qquad\qquad\qquad \longrightarrow \mathit{Pred} \end{array}}{\mathit{Pred}}$$

We know nothing of the bound name generated by this rule. It could appear free anywhere and force us to do manual $\alpha$-conversions.

## Improving structural induction on the |-operator.

### Isabelle's rule for structural induction

$$\frac{\begin{array}{ll} & P \mid Q \longmapsto \alpha \prec PQ' \\ \forall P'\ Q'\ P''\ Q''\ a\ b\ x. & P' \longmapsto a(x) \prec P'' \wedge Q' \longmapsto \bar{a}b \prec Q'' \wedge \\ & P \mid Q = P' \mid Q' \wedge \\ & \alpha \prec PQ' = \tau \prec P''\{b/x\} \mid Q'' \\ & \longrightarrow Pred \end{array}}{Pred}$$

### Derived rule for structural induction

$$\frac{\begin{array}{ll} & P \mid Q \longmapsto \alpha \prec PQ' \\ \forall P'\ Q'\ a\ b\ x. & P \longmapsto a(x) \prec P' \wedge Q \longmapsto \bar{a}b \prec Q' \wedge \\ & \alpha = \tau \wedge PQ' = P'\{b/x\} \mid Q' \wedge x \sharp \mathcal{C} \\ & \longrightarrow Pred \end{array}}{Pred}$$

## Structural induction – Restriction

We will concentrate on the transitions with no bound actions.

**Res**

$$\frac{P \xrightarrow{\alpha} P'}{(\nu x)P \xrightarrow{\alpha} (\nu x)P'} \quad x \notin n(\alpha)$$

From this rule Isabelle derives the following induction rule:

**Isabelle's rule for structural induction**

$$\frac{\begin{array}{c} (\nu x)P \longmapsto \alpha \prec xP' \\ \forall P'\ P''\ \beta\ y. \quad P' \longmapsto \beta \prec P'' \wedge y \sharp \beta \wedge (\nu x)P = (\nu y)P' \wedge \\ \alpha \prec xP' = \beta \prec (\nu y)P'' \longrightarrow Pred \end{array}}{Pred}$$

## Improving structural induction on restriction

**Isabelle's rule for structural induction**

$$\frac{\begin{array}{c}(\nu x)P \longmapsto \alpha \prec xP' \\ \forall P'\ P''\ \beta\ y.\quad P' \longmapsto \beta \prec P'' \wedge y \sharp \beta \wedge (\nu x)P = (\nu y)P' \wedge \\ \alpha \prec xP' = \beta \prec (\nu y)P'' \longrightarrow Pred\end{array}}{Pred}$$

Now things start to get hairy. The assumption $(\nu x)P = (\nu y)P'$ forces us to show the same goal twice for different permutations of names. If the action contains bound names, the same goal would have to be proven $2 \times 2 = 4$ times!!!

## Improving structural induction on restriction

**Isabelle's rule for structural induction**

$$\frac{\begin{array}{c}(\nu x)P \longmapsto \alpha \prec xP' \\ \forall P'\ P''\ \beta\ y.\quad P' \longmapsto \beta \prec P'' \wedge y \sharp \beta \wedge (\nu x)P = (\nu y)P' \wedge \\ \alpha \prec xP' = \beta \prec (\nu y)P'' \longrightarrow Pred\end{array}}{Pred}$$

Fortunately, we can derive the following rule:

**Derived rule for structural induction**

$$\frac{\begin{array}{c}(\nu x)P \longmapsto \alpha \prec xP' \\ \forall P'.\ P \longmapsto \alpha \prec P' \wedge x \sharp \alpha \wedge xP' = (\nu x)P' \longrightarrow Pred\end{array}}{Pred}$$

## The !-operator

Recall the operational semantics for the !-operator.

**Bang**

$$\frac{P \mid !P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}$$

This rule causes a few problems:

- The operator appears in the premise hence structural induction cannot be used – we must do induction on the depth of inference.
- Doing inductive proofs of this kind in theorem provers is tricky and care has to be taken when constructing the inductive rules.

## Simulation

A process $P$ is said to simulate a process $Q$ if for every action $Q$ can do, $P$ can mimic and their derivative are simulations. More formally, a relation $\mathcal{R}$ is said to be a simulation iff:

**Simulation**

$$\begin{aligned} P \rightsquigarrow_{\mathcal{R}} Q \equiv \; & \forall Q' \; \alpha. \; (\forall x \in \mathrm{bn}(\alpha). \; x \sharp P) \wedge \\ & Q \xrightarrow{\alpha} Q' \longrightarrow \exists P'. P \xrightarrow{\alpha} P' \wedge D(P', \; Q', \; \alpha, \; \mathcal{R}) \end{aligned}$$

The requirements of the derivatives depend on the action used.

**D**

$$\begin{aligned} D(P', Q', \alpha, \mathcal{R}) \equiv \; & \text{case } \alpha \text{ of } a(x) && \rightarrow && \forall u. P'\{u/x\} \; \mathcal{R} \; Q'\{u/x\} \\ & \qquad\qquad\quad \mid \_ && \rightarrow && P' \; \mathcal{R} \; Q' \end{aligned}$$

$P \rightsquigarrow_{\mathcal{R}} Q$ can be read "$P$ simulates $Q$ preserving $\mathcal{R}$"

## Restricting the bound names

In the definition for simulation we had the following restriction of bound names: $\forall x \in bn(\alpha).\ x \sharp P$. Consider the following two processes.

$P = a(u).0$
$Q = a(x).(\nu v)\bar{v}u.0$

These two processes simulate each other as they both can do an Input-action and then nothing more. However, $P$ can do the action $a(u)$ whereas Q cannot since $u$ is not free in $Q$. We don't want to distinguish between these two processes, however.

## Proofs on simulations

Typically, proofs made on simulations have the following form:

**Proving simulations**

$$\frac{P \leadsto_{\mathcal{R}} Q \quad \mathcal{R} \subseteq \mathcal{R}' \quad \Phi(\mathcal{R}')}{P' \leadsto_{\mathcal{R}'} Q'}$$ where $\Phi$ is a condition which must be met for the proof to hold.

The proof that simulation is preserved by the $+$-operator has the following form.

**Simulation is preserved by the $+$-operator**

$$\frac{P \leadsto_{\mathcal{R}} Q \quad \mathcal{R} \subseteq \mathcal{R}' \quad Id \subseteq \mathcal{R}'}{P + R \leadsto_{\mathcal{R}'} Q + R}$$

The reason for not having the same relation in the assumption and the conclusion will become apparent when we discuss bisimulation.

## Problems with this approach

Recall the definition for simulation

**Simulation**

$$P \rightsquigarrow_{\mathcal{R}} Q \equiv \forall Q'\ \alpha.\ (\forall x \in \mathrm{bn}(\alpha).\ x \sharp P) \wedge Q \xrightarrow{\alpha} Q' \longrightarrow \exists P'.P \xrightarrow{\alpha} P' \wedge D(P',\ Q',\ \alpha,\ \mathcal{R})$$

Again we face the problem that we know nothing of any bound names that might emerge in a transition.

## Problems with this approach

Recall the definition for simulation

**Simulation**

$$P \rightsquigarrow_{\mathcal{R}} Q \equiv \quad \forall Q'\ \alpha.\ (\forall x \in \mathrm{bn}(\alpha).\ x \sharp P)\ \wedge Q \xrightarrow{\alpha} Q' \longrightarrow \exists P'.P \xrightarrow{\alpha} P' \wedge D(P',\ Q',\ \alpha,\ \mathcal{R})$$

Fortunately, the following introduction rule can be derived from the definition.

**Simulation introduction**

$$\frac{\begin{array}{c}\mathrm{eqvt}\ \mathcal{R} \\ \forall Q'\ \alpha.\quad (\forall x \in bn(\alpha).\ x \sharp \mathcal{C}) \wedge Q \xrightarrow{\alpha} Q' \longrightarrow \\ \exists P'.\ P \xrightarrow{\alpha} P' \wedge D(P',\ Q',\ \alpha,\ \mathcal{R})\end{array}}{P \rightsquigarrow_{\mathcal{R}} Q}$$

## Bisimulation

A relation $\mathcal{R}$ is said to be a bisimulation if both $\mathcal{R}$ and $\mathcal{R}^-$ are simulations. In Isabelle, bisimulation is formalised using coinduction.

**Bisimulation**

$P \sim Q \equiv P \leadsto_\sim Q \land Q \leadsto_\sim P$

## Proofs on bisimulation

When doing coinductive proofs you pick a subset $X$ of your definition which captures what you want to prove. To prove that bisimulation is preserved by the $+$-operator, the following technique can be used:

**Bisimulation is preserved by the $+$-operator.**

$$\frac{P \sim Q}{P + R \sim Q + R} \qquad X = \{x.\exists P\ Q\ R.\ P \sim Q \wedge x = (P + R, Q + R)\}$$

When applying the *coinduct*-tactic in Isabelle, you will need to prove the following:

**Symmetric step (we have to prove both directions)**

$$\frac{P \leadsto_{\sim} Q}{P + R \leadsto_{(\{x.\exists P\ Q\ R.\ P\sim Q\wedge x=(P+R,Q+R)\}\cup\sim)} Q + R}$$

We can then use the congruence rule for simulation where $\mathcal{R} = \sim$

## Advantages with this approach

There are many advantages with this approach.

- The core of all proofs is done on simulation- and not bisimulation level.
- Most proofs need only be proven one way – Isabelle will automatically infer the symmetric versions.
- We can extend the coinductive package of Isabelle so that our work is greatly simplified.

## Strong equivalence

Strong bisimulation is not a congruence. It is not preserved by the Input-prefix. The largest bisimulation relation which also is a congruence needs to be closed under substitutions.

**Closure under substiutions**

substClosed $\mathcal{R} \equiv \{(P, Q).\ \forall \sigma.\ P\sigma\ \mathcal{R}\ Q\sigma\}$

## Strong equivalence

Strong bisimulation is not a congruence. It is not preserved by the Input-prefix. The largest bisimulation relation which also is a congruence needs to be closed under substitutions.

**Closure under substiutions**

substClosed $\mathcal{R} \equiv \{(P, Q).\ \forall\sigma.\ P\sigma\ \mathcal{R}\ Q\sigma\}$

We can now define strong equivalence:

**Strong equivalence**

$\simeq\ \equiv$ substClosed $\sim$

## Proofs with strong equivalence

Proving properties for strongly equivalent processes is usually trivial once you have the corresponding proof for strong bisimulation. The tricky cases are the processes with restrictions as manual $\alpha$-conversions have to be done when the substitution clashes with the bound name.

This is unavoidable even when doing paper proofs.

## What have we accomplished

So far our results are very encouraging:

- We have successfully incorporated the *hand waving* regarding $\alpha$-equivalence into a theorem prover.
- We have proven that strong bisimulation is preserved by all operators except the Input-prefix.
- We have proven that strong equivalence is a congruence.
- We have proven all of the structural congruence law – i.e. if two processes are structurally congruent, then they are strongly equivalent (and strongly bisimilar).

## What have we accomplished

So far our results are very encouraging:

- We have successfully incorporated the *hand waving* regarding $\alpha$-equivalence into a theorem prover.
- We have proven that strong bisimulation is preserved by all operators except the Input-prefix.
- We have proven that strong equivalence is a congruence.
- We have proven all of the structural congruence law – i.e. if two processes are structurally congruent, then they are strongly equivalent (and strongly bisimilar).

Some manual $\alpha$-conversions still have to be done, but only where they would have to be done in a solid paper proof as well.

## Conclusions

The road has been long. What have we learnt?

- The hand waving which is done in literature regarding $\alpha$-equivalence is not easily formalised in a theorem prover (old news).
- it is essential to create rules for structural induction which work up to $\alpha$-equivalence and not only syntactic equivalence.
- The nominal package has very nice logical properties on the other hand and its further incorporation into Isabelle, especially with tactics regarding substitutions and permutations, is eagerly anticipated.

## Future work

There is plenty of work left to do. There is the imminent.

- Finish the proofs for weak bisimulation.
- Polish all the theories and publish a paper.

... and the more distant

- Create support for reasoning about recursively defined processes in Isabelle.
- Create a tactic to do automatic bisimulation checking of finite processes.
- Increase the library of lemmas to include structural congruence.

... and maybe... possibly if we feel so inclined

- create an external user interface to allow for the framework to be used by other tools.

## Thank you

Thank you for your attention.