Semantics for a Pure, Functional Programming Language with Names

Steffen Lösch and Andrew M. Pitts

UNIVERSITY OF CAMBRIDGE

Computer Laboratory
Semantics

Semantics is the field concerned with the rigorous mathematical study of the meaning of programming languages. - Wikipedia

Why are we interested?

Steffen Lösch and Andrew M. Pitts
Semantics for a Pure, Functional Prog. Lang. with Names
Semantics is the field concerned with the rigorous mathematical study of the meaning of programming languages. - Wikipedia
Semantics

Semantics is the field concerned with the rigorous mathematical study of the meaning of programming languages. - Wikipedia

Why are we interested?
Semantics is the field concerned with the rigorous mathematical study of the meaning of programming languages. - Wikipedia

Why are we interested?
Pure, Functional Programming

- Functions are first class citizens
- Pure
- No state or effects

Important examples:
- $\lambda$-calculus
- PCF, System F
- (LISP/Scheme, Haskell)
- Coq, Agda

(Results presented can be generalised to other languages)
Pure, Functional Programming

Functional

- functions are first class citizens
Pure, Functional Programming

*Functional*

- functions are first class citizens

*Pure*

- no state or effects
Pure, Functional Programming

**Functional**
- functions are first class citizens

**Pure**
- no state or effects

Important examples:
- $\lambda$-calculus
- **PCF**, System F
- (LISP/Scheme, Haskell)
- Coq, Agda
Pure, Functional Programming

Functional

▶ functions are first class citizens

Pure

▶ no state or effects

Important examples:

- $\lambda$-calculus
- **PCF**, System F
- (LISP/Scheme, Haskell)
- Coq, Agda

(results presented can be generalised to other languages)
Overview

1. Introduction

2. PCF
   - language and operational semantics
   - denotational semantics
   - parallel-or and full abstraction

3. PCF with names = PNA
   - motivation and language
   - operational semantics

4. Denotational semantics and full abstraction for PNA

Steffen Lösch and Andrew M. Pitts

Semantics for a Pure, Functional Prog. Lang. with Names
Overview

1. Introduction
1. Introduction
2. PCF
   - language and operational semantics
   - denotational semantics
   - parallel-or and full abstraction
1. Introduction

2. PCF
   - language and operational semantics
   - denotational semantics
   - parallel-or and full abstraction

3. PCF with names = PNA
   - motivation and language
   - operational semantics
Overview

1. Introduction
2. PCF
   ▶ language and operational semantics
   ▶ denotational semantics
   ▶ parallel-or and full abstraction
3. PCF with names = PNA
   ▶ motivation and language
   ▶ operational semantics
4. Denotational semantics and full abstraction for PNA
PCF – Syntax

\[ \tau \in \text{Typ} ::= \text{bool} \mid \text{nat} \mid \tau \times \tau \mid \tau \times \tau \]

\[ e \in \text{Exp} ::= T \mid F \mid \text{if } e \text{ then } e \text{ else } e \mid \text{booleans and if } O \mid S \mid e \mid \text{pred } e \mid \text{zero } e \mid \text{numbers and zero test } (e, e) \mid \text{fst } e \mid \text{snd } e \mid \text{pairs and projections } x \mid \lambda x : \tau \mid e e \mid \text{variable } (x \in V) \mid \text{functions } \text{fix } e \text{ fixed-point recursion} \]

Steffen Lösch and Andrew M. Pitts
Semantics for a Pure, Functional Prog. Lang. with Names
PCF – Syntax

▶ Types

\[ \tau \in \text{Typ} ::= \text{bool} \mid \text{nat} \mid \tau \times \tau \mid \tau \rightarrow \tau \]
PCF – Syntax

- **Types**
  \[ \tau \in \text{Typ} ::= \text{bool} | \text{nat} | \tau \times \tau | \tau \to \tau \]

- **Expressions**
  \[ e \in \text{Exp} ::= T | F | \text{if } e \text{ then } e \text{ else } e | \text{booleans and if} \]
  \[ 0 | S e | \text{pred } e | \text{zero } e | \text{numbers and zero test} \]
  \[ (e, e) | \text{fst } e | \text{snd } e | \text{pairs and projections} \]
  \[ x | \lambda x : \tau \to e | e e | \text{variable } (x \in \mathbb{V}) \text{ and functions} \]
  \[ \text{fix } e | \text{fixed-point recursion} \]
PCF – Syntax

- **Types**

  \[ \tau \in \text{Typ} ::= \text{bool} \mid \text{nat} \mid \tau \times \tau \mid \tau \rightarrow \tau \]

- **Expressions**

  \[ e \in \text{Exp} ::= T \mid F \mid \text{if } e \text{ then } e \text{ else } e \mid \text{booleans and if}
  
  0 \mid S\ e \mid \text{pred } e \mid \text{zero } e \mid \text{numbers and zero test}
  
  (e, e) \mid \text{fst } e \mid \text{snd } e \mid \text{pairs and projections}
  
  x \mid \lambda x : \tau \rightarrow e \mid e\ e \mid \text{variable } (x \in V) \text{ and functions}
  
  \text{fix } e \mid \text{fixed-point recursion} \]

- **Binding forms**

  \[ \lambda x : \tau \rightarrow e \]

Steffen Lösch and Andrew M. Pitts  
Semantics for a Pure, Functional Prog. Lang. with Names
PCF – Typing

Typing judgement: $\Gamma \vdash e : \tau$

$c = T \mid F$

$\Gamma \vdash c : bool$

$\Gamma \vdash e_1 : bool$

$\Gamma \vdash e_2 : \tau$

$\Gamma \vdash e_3 : \tau$

$\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau$

$\Gamma \vdash e : \text{nat}$

$\Gamma \vdash \text{zero} : bool$

$\Gamma \vdash e_1 : \tau_1$

$\Gamma \vdash e_2 : \tau_2$

$\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2$

$\Gamma \vdash e : \tau_1 \times \tau_2$

$\Gamma \vdash \text{fst} e : \tau_1$

$\Gamma \vdash \text{snd} e : \tau_2$

$(x : \tau) \in \Gamma$

$\Gamma, x : \tau \vdash e : \tau'$

$\Gamma \vdash \lambda x : \tau . e : \tau \tau'$

$\Gamma \vdash e_1 : \tau \tau'$

$\Gamma \vdash e_2 : \tau$

$\Gamma \vdash e_1 e_2 : \tau'$

$\Gamma \vdash \text{fix } e : \tau$

$\Gamma \vdash O : \text{nat}$

$\Gamma \vdash e : \text{nat}$

$\Gamma \vdash S e : \text{nat}$

$\Gamma \vdash \text{pred } e : \text{nat}$

$\Gamma = \{ x_1 : \tau_1, \ldots, x_n : \tau_n \}$

Steffen Lösch and Andrew M. Pitts

Semantics for a Pure, Functional Prog. Lang. with Names
Typing judgement: $\Gamma \vdash e : \tau$
Typing judgement: $\Gamma \vdash e : \tau$

$\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$
Typing judgement: $\Gamma \vdash e : \tau$

$\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$

$(x : \tau) \in \Gamma$

$$\frac{}{\Gamma \vdash x : \tau}$$
Typing judgement: $\Gamma \vdash e : \tau$

$c = T | F$

$\Gamma \vdash c : \text{bool}$

$(x : \tau) \in \Gamma$

$\Gamma \vdash x : \tau$
Typing judgement: $\Gamma \vdash e : \tau$

\[
\begin{align*}
  c & = \text{T} \mid \text{F} \\
  \Gamma & \vdash c : \text{bool} & \\
  \Gamma & \vdash e_1 : \text{bool} & \Gamma & \vdash e_2 : \tau & \Gamma & \vdash e_3 : \tau \\
  \Gamma & \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau
\end{align*}
\]

\[
\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}
\]
Typing judgement: $\Gamma \vdash e : \tau$

\[
\begin{array}{ll}
\frac{c = T \mid F}{\Gamma \vdash c : bool} & \frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \\
\frac{\Gamma \vdash e : \text{nat}}{\Gamma \vdash \text{zero } e : \text{bool}} & \\
\end{array}
\]

\[
(\chi : \tau) \in \Gamma \quad \frac{}{\Gamma \vdash \chi : \tau}
\]
Typing judgement: $\Gamma \vdash e : \tau$

- $c = T \mid F \quad \frac{\Gamma \vdash e_1 : \text{bool}}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau}$
- $\Gamma \vdash e : \text{nat} \quad \frac{\Gamma \vdash e_1 : \text{nat} \quad \Gamma \vdash e_2 : \text{nat}}{\Gamma \vdash \text{pred } e : \text{nat}}$
- $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$

$x : \tau \in \Gamma 
\frac{\Gamma \vdash x : \tau}{\Gamma, x : \tau \vdash e : \tau'} 
\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau \to e : \tau \to \tau'}$

$\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau 
\frac{\Gamma \vdash e_1 \ e_2 : \tau'}{\Gamma \vdash e_1 \ e_2 : \tau'}$
Typing judgement: $\Gamma \vdash e : \tau$

\[
\begin{align*}
  c &= T \mid F \\
  \Gamma &\vdash c : bool
\end{align*}
\]

\[
\begin{align*}
  \Gamma &\vdash e_1 : bool \\
  \Gamma &\vdash e_2 : \tau \\
  \Gamma &\vdash e_3 : \tau \\
  \Gamma &\vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau
\end{align*}
\]

\[
\begin{align*}
  \Gamma &\vdash e : \tau
\end{align*}
\]

\[
\begin{align*}
  (x : \tau) &\in \Gamma \\
  \Gamma &\vdash x : \tau
\end{align*}
\]

\[
\begin{align*}
  \Gamma, x : \tau &\vdash e : \tau' \quad \Gamma &\vdash \lambda x : \tau \to e : \tau \to \tau'
\end{align*}
\]

\[
\begin{align*}
  \Gamma &\vdash e_1 : \tau \to \tau' \\
  \Gamma &\vdash e_2 : \tau \\
  \Gamma &\vdash e_1\ e_2 : \tau'
\end{align*}
\]

\[
\begin{align*}
  \Gamma &\vdash e : \tau \to \tau
\end{align*}
\]

\[
\begin{align*}
  \Gamma &\vdash \text{fix } e : \tau
\end{align*}
\]
### PCF – Typing

#### Typing judgement: $\Gamma \vdash e : \tau$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c = T \mid F$</td>
<td>$\Gamma \vdash c : \text{bool}$</td>
</tr>
<tr>
<td>$\Gamma \vdash e_1 : \text{bool}$</td>
<td>$\Gamma \vdash e_2 : \tau$</td>
</tr>
<tr>
<td>$\Gamma \vdash e : \text{nat}$</td>
<td>$\Gamma \vdash e_1 : \tau_1$</td>
</tr>
<tr>
<td>$\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2$</td>
<td>$\Gamma \vdash \text{fst } e : \tau_1$</td>
</tr>
<tr>
<td>$\Gamma \vdash \text{snd } e : \tau_2$</td>
<td>$\Gamma, x : \tau \vdash e : \tau'$</td>
</tr>
<tr>
<td>$\Gamma \vdash x : \tau$</td>
<td>$\Gamma \vdash \lambda x : \tau \to e : \tau \to \tau'$</td>
</tr>
<tr>
<td>$\Gamma \vdash e_1 : \tau \to \tau'$</td>
<td>$\Gamma \vdash e_2 : \tau$</td>
</tr>
<tr>
<td>$\Gamma \vdash e_1 e_2 : \tau'$</td>
<td></td>
</tr>
<tr>
<td>$\Gamma \vdash 0 : \text{nat}$</td>
<td>$\Gamma \vdash S e : \text{nat}$</td>
</tr>
</tbody>
</table>
Currying

\[ \lambda f : (\tau_1 \times \tau_2 \times \tau_3) \lambda x_1 : \tau_1 \lambda x_2 : \tau_2 f(x_1, x_2) \]

Sum of numbers:

\[ x + y \]

Equality test for numbers:

\[ x = y ? \]

\[ \text{fix} \left( \lambda f : \text{nat} \times \text{nat} \times \text{nat} \lambda x : \text{nat} \lambda y : \text{nat} \text{if zero } y \text{ then } x \text{ else } S \left( f x \left( \text{pred } y \right) \right) \right) \]

\[ \text{fix} \left( \lambda f : \text{nat} \times \text{nat} \times \text{nat} \lambda x : \text{nat} \lambda y : \text{nat} \text{if zero } x \text{ then zero } y \text{ else } \text{if zero } y \text{ then } \text{F else } f \left( \text{pred } x \right) \left( \text{pred } y \right) \right) \]
Currying

\[ \lambda f : (\tau_1 \times \tau_2 \rightarrow \tau_3) \rightarrow \lambda x_1 : \tau_1 \rightarrow \lambda x_2 : \tau_2 \rightarrow f (x_1, x_2) \]
PCF – Example Programs

Currying

$$\lambda f : (\tau_1 \times \tau_2 \rightarrow \tau_3) \rightarrow \lambda x_1 : \tau_1 \rightarrow \lambda x_2 : \tau_2 \rightarrow f \ (x_1 \ , \ x_2)$$

Sum of numbers: $x + y$

$$\text{fix}(\lambda (f : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}) \rightarrow \lambda x : \text{nat} \rightarrow \lambda y : \text{nat} \rightarrow \text{if zero } y \ \text{then } x \ \text{else } S (f \ x \ (\text{pred} \ y)))$$
PCF – Example Programs

Currying

\[ \lambda f : (\tau_1 \times \tau_2 \rightarrow \tau_3) \rightarrow \lambda x_1 : \tau_1 \rightarrow \lambda x_2 : \tau_2 \rightarrow f(x_1, x_2) \]

Sum of numbers: \( x + y \)

\[ \text{fix}(\lambda(f : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}) \rightarrow \lambda x : \text{nat} \rightarrow \lambda y : \text{nat} \rightarrow \text{if zero } y \text{ then } x \text{ else } S(f(x(pred\,y))) \]  

Equality test for numbers: \( x = y ? \)

\[ \text{fix}(\lambda(f : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}) \rightarrow \lambda x : \text{nat} \rightarrow \lambda y : \text{nat} \rightarrow \text{if zero } x \text{ then zero } y \text{ else } (\text{if zero } y \text{ then } F \text{ else } f(\text{pred}\,x_1)(\text{pred}\,x_2))) \]
PCF – Operational Semantics

Evaluation judgement:
\[ e \downarrow c \]

\[ c = T \mid F \mid O \mid (e_1, e_2) \mid \lambda x: \tau \]

\[ e \downarrow c \]

\[ S e \downarrow c \]

\[ \text{pred} e \downarrow c \]

\[ e_1 \downarrow T \]

\[ e_2 \downarrow c \]

\[ \text{if} e_1 \text{then} e_2 \text{else} e_3 \downarrow c \]

\[ e_1 \downarrow F \]

\[ e_3 \downarrow c \]

\[ \text{if} e_1 \text{then} e_2 \text{else} e_3 \downarrow c \]

\[ e_1 \downarrow O \]

\[ \text{zero} e \downarrow T \]

\[ e \downarrow S \]

\[ \text{zero} e \downarrow F \]

\[ (e_1, e_2) \]

\[ e_1 \downarrow c \]

\[ \text{fst} e \downarrow c \]

\[ (e_1, e_2) \]

\[ e_2 \downarrow c \]

\[ \text{snd} e \downarrow c \]

\[ e_1 \downarrow \lambda x: \tau \]

\[ e \left[ e_2 / x \right] \downarrow c \]

\[ e \downarrow \text{fix} e \downarrow c \]

\[ \text{closed expression} \]

\[ c \in \text{Can} \]

\[ c ::= T \mid F \mid O \mid S c \mid (e_1, e_2) \mid \lambda x: \tau \]

Steffen Lösch and Andrew M. Pitts

Semantics for a Pure, Functional Prog. Lang. with Names
Evaluation judgement: \( e \downarrow c \)
PCF – Operational Semantics

Evaluation judgement: \( e \downarrow c \)

Closed expression

\[ c \in \text{Can} ::= T \mid F \mid 0 \mid S \mid (e,e) \mid \lambda x : \tau \rightarrow e \]
**Evaluation judgement**: \( e \downarrow c \)

Closed expression:

\[ c \in \text{Can} ::= T \mid F \mid 0 \mid S \, c \mid (e, e) \mid \lambda \, x : \tau \to e \]

\[ c = T \mid F \mid 0 \mid (e_1, e_2) \mid \lambda \, x : \tau \to e \]

\[ c \downarrow c \]
PCF – Operational Semantics

Evaluation judgement: \( e \downarrow c \)

\[
c \in \text{Can} ::= T \mid F \mid O \mid S \mid (e, e) \mid \lambda x : \tau \rightarrow e
\]

\[
c = T \mid F \mid O \mid (e_1, e_2) \mid \lambda x : \tau \rightarrow e
\]

\[
c \downarrow c
\]

\[
e_1 \downarrow F \quad e_3 \downarrow c
\]

if \( e_1 \) then \( e_2 \) else \( e_3 \) \( \downarrow c \)
PCF – Operational Semantics

Evaluation judgement: $e \Downarrow c$

$c \in \text{Can} ::= T | F | 0 | S \ c | (e, e) | \lambda x : \tau \rightarrow e$

$c = T | F | 0 | (e_1, e_2) | \lambda x : \tau \rightarrow e$

\[ \frac{}{c \Downarrow c} \]

\[ \frac{e_1 \Downarrow F \quad e_3 \Downarrow c}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow c} \]

\[ \frac{e \Downarrow 0}{\text{zero } e \Downarrow T} \]
PCF – Operational Semantics

Evaluation judgement: $e \Downarrow c$

$c \in \text{Can} ::= T | F | O | S \; c | (e_1, e_2) | \lambda x : \tau \to e$

$$c = T | F | O | (e_1, e_2) | \lambda x : \tau \to e$$

$$c \Downarrow c$$

$$\frac{e_1 \Downarrow F \quad e_3 \Downarrow c}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow c} \quad \frac{e \Downarrow 0}{\text{zero } e \Downarrow T}$$

$$\frac{e_1 \Downarrow \lambda x : \tau \to e \quad e[e_2/x] \Downarrow c}{e_1 \; e_2 \Downarrow c}$$
Evaluation judgement: $e \Downarrow c$

$c \in \text{Can} ::= T | F | 0 | S \ c | (e, e) | \lambda x: \tau \to e$

$c = T | F | 0 | (e_1, e_2) | \lambda x: \tau \to e$

$\frac{\quad c \Downarrow c \quad}{\quad e_1 \Downarrow F \quad e_3 \Downarrow c \quad e \Downarrow 0}
\frac{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow c \quad}{\text{zero } e \Downarrow T}

\frac{\quad e_1 \Downarrow \lambda x: \tau \to e \quad e[e_2/x] \Downarrow c \quad e(\text{fix } e) \Downarrow c \quad}{\quad e_1 e_2 \Downarrow c \quad \text{fix } e \Downarrow c \quad}$
PCF – Operational Semantics

Evaluation judgement: $e \Downarrow c$

$c \in \text{Can} ::= T \mid F \mid 0 \mid S \cdot c \mid (e, e) \mid \lambda x : \tau \rightarrow e$

$c = T \mid F \mid 0 \mid (e_1, e_2) \mid \lambda x : \tau \rightarrow e$

\[ \frac{c \Downarrow c}{S \cdot e \Downarrow S \cdot c} \]

\[ \frac{e \Downarrow c}{\text{pred} \cdot e \Downarrow c} \]

$e_1 \Downarrow T$

$e_2 \Downarrow c$

\[ \frac{\text{if} \cdot e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow c}{e_1 \Downarrow F} \]

$e_3 \Downarrow c$

$e \Downarrow 0$

\[ \frac{\text{zero} \cdot e \Downarrow T}{\text{zero} \cdot e \Downarrow F} \]

$e \Downarrow S \cdot c$

$e \Downarrow (e_1, e_2)$

$e_1 \Downarrow c$

\[ \frac{\text{fst} \cdot e \Downarrow c}{e \Downarrow (e_1, e_2)} \]

\[ \frac{e_2 \Downarrow c}{\text{snd} \cdot e \Downarrow c} \]

$e_1 \Downarrow \lambda x : \tau \rightarrow e$

\[ \frac{e_2 [x / x] \Downarrow c}{e_1 \cdot e_2 \Downarrow c} \]

$e \Downarrow (e_1, e_2)$

\[ \frac{e(\text{fix} \cdot e) \Downarrow c}{\text{fix} \cdot e \Downarrow c} \]

Steffen Lösch and Andrew M. Pitts
Semantics for a Pure, Functional Prog. Lang. with Names
Overview

1. Introduction
2. PCF
   ▶ language and operational semantics
   ▶ denotational semantics
   ▶ parallel-or and full abstraction
3. PCF with names = PNA
   ▶ motivation and language
   ▶ operational semantics
4. Denotational semantics and full abstraction for PNA
Domains

Domain

- directed-complete partial order (dcpo) with bottom
- partial order: set with reflexive, transitive and anti-symmetric relation
- directed: every pair of elements has an upper bound
- directed-complete: every directed subset has a join (= least upper bound, written as $\bigvee$)
- bottom: least element of the partial order, written as $\bot$
Domains

Domain = directed-complete partial order (dcpo) with bottom
Domains

\[ \text{Domain} = \text{directed-complete partial order (dcpo) with bottom} \]

- **partial order**: set with reflexive, transitive and anti-symmetric relation \( \sqsubseteq \)
Domains

\[ \text{Domain} = \text{directed-complete partial order (dcpo) with bottom} \]

- **partial order**: set with reflexive, transitive and anti-symmetric relation \( \sqsubseteq \)
- **directed**: every pair of elements has an upper bound
Domains

Domain = directed-complete partial order (dcpo) with bottom

- **partial order**: set with reflexive, transitive and anti-symmetric relation \( \sqsubseteq \)
- **directed**: every pair of elements has an upper bound
- **directed-complete**: every directed subset has a join (= least upper bound, written as \( \sqcup \))
Domains

Domain = directed-complete partial order (dcpo) with bottom

- **partial order**: set with reflexive, transitive and anti-symmetric relation \( \sqsubseteq \)
- **directed**: every pair of elements has an upper bound
- **directed-complete**: every directed subset has a join (= least upper bound, written as \( \sqcup \))
- **bottom**: least element of the partial order, written as \( \bot \)
Continuity

We use continuous functions to model function types.

Continuous function = directed-join-preserving function

▶ directed join = \( \sqcup \) S of a directed subset S

▶ directed-join-preserving:

\[
\forall f \in \text{dom}(\sqcup S) \quad f(\sqcup S) = \sqcup \{ f(d) \mid d \in S \}
\]

▶ point-wise order:

\[
f \sqsubseteq g \iff \forall d \in \text{dom}(f) \quad f(d) \sqsubseteq g(d)
\]

▶ fixed point:

\[
f(x) = x
\]

Crucial results:

1. The point-wise ordered set of continuous functions \( D \rightarrow C \) between two domains \( D, E \) is again a domain.
2. Every continuous function has a least fixed point given by

\[
\text{fix}(f) = \sqcup \{ f^n(\bot) \mid n \geq 0 \}
\]
We use continuous functions to model function types

Continuous function = directed-join-preserving function
Continuity

We use **continuous** functions to model function types

\[ \text{Continuous function} = \text{directed-join-preserving function} \]

- **directed join**: \( \text{join} \bigcup S \) of a directed subset \( S \)

Crucial results:

1. The point-wise ordered set of continuous functions \( D \mapsto E \) between two domains \( D, E \) is again a domain.
2. Every continuous function has a least fixed point given by \( \text{fix}(f) = \bigcup \{ f^n(\bot) \mid n \geq 0 \} \).
We use continuous functions to model function types.

Continuous function = directed-join-preserving function

- **directed join**: $\text{join } \bigsqcup S$ of a directed subset $S$
- **directed-join-preserving**: $f(\bigsqcup S) = \bigsqcup \{f(d) \mid d \in S\}$

Crucial results:
1. The point-wise ordered set of continuous functions $D^cE$ between two domains $D, E$ is again a domain.
2. Every continuous function has a least fixed point given by $\text{fix}(f) = \bigsqcup \{f^n(\bot) \mid n \geq 0\}$. 
We use **continuous** functions to model function types

*Continuous function* $= \text{directed-join-preserving function}

- **directed join**: $\text{join } \bigsqcup S$ of a directed subset $S$
- **directed-join-preserving**: $f(\bigsqcup S) = \bigsqcup \{f(d) \mid d \in S\}$
- **point-wise order**: $f \sqsubseteq g$ iff $(\forall d) \, f(d) \sqsubseteq g(d)$

Crucial results:
1. The point-wise ordered set of continuous functions $D^c E$ between two domains $D, E$ is again a domain.
2. Every continuous function has a least fixed point given by $\text{fix}(f) = \bigsqcup \{f^n(\perp) \mid n \geq 0\}$. 
We use **continuous** functions to model function types

**Continuous function** = directed-join-preserving function

- **directed join**: $\text{join} \bigcup S$ of a directed subset $S$
- **directed-join-preserving**: $f(\bigcup S) = \bigcup \{f(d) \mid d \in S\}$
- **point-wise order**: $f \sqsubseteq g$ iff $(\forall d) f(d) \sqsubseteq g(d)$
- **fixed point**: $f(x) = x$

Crucial results:
1. The point-wise ordered set of continuous functions $D \leq E$ between two domains $D$, $E$ is again a domain.
2. Every continuous function has a least fixed point given by $\text{fix}(f) = \bigcup \{f^n(\bot) \mid n \geq 0\}$.
We use **continuous** functions to model function types

**Continuous function** = directed-join-preserving function

- **directed join**: $\bigvee S$ of a directed subset $S$
- **directed-join-preserving**: $f(\bigvee S) = \bigvee\{f(d) \mid d \in S\}$
- **point-wise order**: $f \sqsubseteq g$ iff $(\forall d) f(d) \sqsubseteq g(d)$
- **fixed point**: $f(x) = x$

Crucial results:

1. The point-wise ordered set of continuous functions $D \to_c E$ between two domains $D, E$ is again a domain.
2. Every continuous function has a least fixed point given by $\text{fix}(f) = \bigvee\{f^n(\bot) \mid n \geq 0\}$. 
Domains and continuous functions give enough structure to define a denotational semantics $J$ for PCF.

$\Gamma \vdash e : \tau \Rightarrow J e^K \in J \Gamma K$

Types denote domains:

$\text{bool} = \{\text{true}, \text{false}\}$

$\text{nat} = \{0, 1, 2, ...\}$

$\tau_1 \times \tau_2 = \tau_1 \times \tau_2$

$\Gamma = \{x_1 : \tau_1, ..., x_n : \tau_n\}$
Domains and continuous functions give enough structure to define a denotational semantics $\llbracket - \rrbracket$ for PCF.

$$\Gamma \vdash e : \tau \Rightarrow \llbracket e \rrbracket \in \llbracket \Gamma \rrbracket \rightarrow_{c} \llbracket \tau \rrbracket$$
Domains and continuous functions give enough structure to define a denotational semantics $\llbracket - \rrbracket$ for PCF.

\[ \Gamma \vdash e : \tau \Rightarrow \llbracket e \rrbracket \in \llbracket \Gamma \rrbracket \rightarrow_c \llbracket \tau \rrbracket \]

Types denote domains:
Domains and continuous functions give enough structure to define a denotational semantics $\llbracket - \rrbracket$ for PCF.

$$\Gamma \vdash e : \tau \Rightarrow \llbracket e \rrbracket \in \llbracket \Gamma \rrbracket \rightarrow_c \llbracket \tau \rrbracket$$

Types denote domains:

- $\llbracket \text{bool} \rrbracket = \{\text{true}, \text{false}\}$
Domains and continuous functions give enough structure to define a denotational semantics $\llbracket \cdot \rrbracket$ for PCF.

$$\Gamma \vdash e : \tau \Rightarrow \llbracket e \rrbracket \in \llbracket \Gamma \rrbracket \rightarrow c \llbracket \tau \rrbracket$$

Types denote domains:

- $\llbracket \text{bool} \rrbracket = \{\text{true}, \text{false}\} \bot$
- $\llbracket \text{nat} \rrbracket = \{0, 1, 2, \ldots\} \bot$
Domains and continuous functions give enough structure to define a denotational semantics $⟦−⟧$ for PCF.

$$\Gamma \vdash e : \tau \Rightarrow [e] \in [\Gamma] \rightarrow c [\tau]$$

Types denote domains:

- $[\text{bool}] = \{\text{true}, \text{false}\}_{\bot}$
- $[\text{nat}] = \{0, 1, 2, \ldots\}_{\bot}$
- $[\tau_1 \times \tau_2] = [\tau_1] \times [\tau_2]$
Domains and continuous functions give enough structure to define a denotational semantics $\llbracket \cdot \rrbracket$ for PCF.

\[
\Gamma \vdash e : \tau \Rightarrow \llbracket e \rrbracket \in \llbracket \Gamma \rrbracket \to^c \llbracket \tau \rrbracket
\]

Types denote domains:

- $\llbracket \text{bool} \rrbracket = \{\text{true, false}\} \perp$
- $\llbracket \text{nat} \rrbracket = \{0, 1, 2, \ldots\} \perp$
- $\llbracket \tau_1 \times \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket$
- $\llbracket \tau_1 \to \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \to^c \llbracket \tau_2 \rrbracket$
Domains and continuous functions give enough structure to define a denotational semantics $\llbracket - \rrbracket$ for PCF.

\[
\Gamma \vdash e : \tau \Rightarrow \llbracket e \rrbracket \in \llbracket \Gamma \rrbracket \rightarrow_c \llbracket \tau \rrbracket
\]

Types denote domains:

- $\llbracket \text{bool} \rrbracket = \{\text{true}, \text{false}\} \bot$
- $\llbracket \text{nat} \rrbracket = \{0, 1, 2, \ldots\} \bot$
- $\llbracket \tau_1 \times \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket$
- $\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \rightarrow_c \llbracket \tau_2 \rrbracket$
- $\llbracket \Gamma \rrbracket = \llbracket \tau_1 \rrbracket \times \ldots \times \llbracket \tau_n \rrbracket$ if $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$
Expressions denote continuous functions:
Expressions denote continuous functions:
- $[x] \rho = \rho x$
Expressions denote continuous functions:

- \([x] \rho = \rho x\)
- \([T] \rho = \text{true}\)
Expressions denote continuous functions:

- $[x] \rho = \rho x$
- $[T] \rho = \text{true}$
- $[\text{if } e_1 \text{ then } e_2 \text{ else } e_3] \rho = \begin{cases} [e_2] \rho & \text{if } [e_1] \rho = \text{true} \\ [e_3] \rho & \text{if } [e_1] \rho = \text{false} \\ \bot & \text{otherwise} \end{cases}$
Expressions denote continuous functions:

- $[x] \rho = \rho x$
- $[T] \rho = \text{true}$
- $[\text{if } e_1 \text{ then } e_2 \text{ else } e_3] \rho = \begin{cases} [e_2] \rho & \text{if } [e_1] \rho = \text{true} \\ [e_3] \rho & \text{if } [e_1] \rho = \text{false} \\ \bot & \text{otherwise} \end{cases}$
- $[\lambda x : \tau \to e] \rho = \lambda d \in [\tau]. [e](\rho[x \mapsto d])$
Expressions denote continuous functions:

- $[x] \rho = \rho x$
- $[T] \rho = \text{true}$
- $[\text{if } e_1 \text{ then } e_2 \text{ else } e_3] \rho = \begin{cases} [e_2] \rho & \text{if } [e_1] \rho = \text{true} \\ [e_3] \rho & \text{if } [e_1] \rho = \text{false} \\ \bot & \text{otherwise} \end{cases}$
- $[\lambda x : \tau \to e] \rho = \lambda d \in [\tau]. [e]([\rho[x \mapsto d]])$
- $[\text{fix } e] \rho = \text{fix}([e] \rho)$
Expressions denote continuous functions:

- $\llbracket x \rrbracket \rho = \rho x$
- $\llbracket T \rrbracket \rho = \text{true}$
- $\llbracket \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rrbracket \rho = \begin{cases} \llbracket e_2 \rrbracket \rho & \text{if } \llbracket e_1 \rrbracket \rho = \text{true} \\ \llbracket e_3 \rrbracket \rho & \text{if } \llbracket e_1 \rrbracket \rho = \text{false} \\ \bot & \text{otherwise} \end{cases}$
- $\llbracket \lambda x : \tau \to e \rrbracket \rho = \lambda d \in [\tau]. [e](\rho[x \mapsto d])$
- $\llbracket \text{fix } e \rrbracket \rho = \text{fix}(\llbracket e \rrbracket \rho)$

$\llbracket F \rrbracket \rho = \text{false}$
$\llbracket 0 \rrbracket \rho = 0$
$\llbracket S e \rrbracket \rho = \begin{cases} n + 1 & \text{if } \llbracket e \rrbracket \rho = n \in \mathbb{N} \\ \bot & \text{otherwise} \end{cases}$

$\llbracket \text{pred } e \rrbracket \rho = \begin{cases} n & \text{if } \llbracket e \rrbracket \rho = n + 1 \in \mathbb{N} \\ \bot & \text{otherwise} \end{cases}$
$\llbracket \text{zero } e \rrbracket \rho = \begin{cases} \text{true} & \text{if } \llbracket e \rrbracket \rho = 0 \in \mathbb{N} \\ \text{false} & \text{if } \llbracket e \rrbracket \rho = n + 1 \in \mathbb{N} \\ \bot & \text{otherwise} \end{cases}$

$\llbracket (e_1, e_2) \rrbracket \rho = (\llbracket e_1 \rrbracket \rho, \llbracket e_2 \rrbracket \rho)$
$\llbracket \text{fst } e \rrbracket \rho = \pi_1(\llbracket e \rrbracket \rho)$
$\llbracket \text{snd } e \rrbracket \rho = \pi_2(\llbracket e \rrbracket \rho)$

$\llbracket e_1 \ e_2 \rrbracket \rho = \llbracket e_1 \rrbracket \rho (\llbracket e_2 \rrbracket \rho)$
PCF – Adequacy

Contextual equivalence $\equiv$ two expressions $e_1, e_2$ are contextual equivalent $e_1 \equiv_{PCF} e_2$ iff they have the same operational behaviour in every context.

▶ context: an expression with a hole; we write $C[\_\_]$ for contexts and $C[e]$ is the result of replacing the hole with $e$

▶ operational behaviour: evaluate by the operational semantics to the same canonical form

▶ same behaviour in every context: for every context $C[\_\_]$ it holds that $C[e_1] \Downarrow_c \iff C[e_2] \Downarrow_c$

We obtain a theorem and proof principle:

Adequacy: $Je_1K = Je_2K \Rightarrow e_1 \equiv_{PCF} e_2$
Contextual equivalence = two expressions $e_1, e_2$ are contextual equivalent $e_1 \simeq_{\text{PCF}} e_2$ iff they have the same operational behaviour in every context.
Contextual equivalence $=$ two expressions $e_1, e_2$ are contextual equivalent $e_1 \simeq_{\text{PCF}} e_2$ iff they have the same operational behaviour in every context.

- **context**: an expression with a hole; we write $C[\_]$ for contexts and $C[e]$ is the result of replacing the hole with $e$
Contextual equivalence = two expressions $e_1, e_2$ are contextual equivalent $e_1 \simeq_{PCF} e_2$ iff they have the same operational behaviour in every context.

- **context**: an expression with a hole; we write $C[\_]$ for contexts and $C[e]$ is the result of replacing the hole with $e$
- **operational behaviour**: evaluate by the operational semantics to the same canonical form
Contextual equivalence = two expressions $e_1, e_2$ are contextual equivalent $e_1 \simeq_{\text{PCF}} e_2$ iff they have the same operational behaviour in every context.

- context: an expression with a hole; we write $C[-]$ for contexts and $C[e]$ is the result of replacing the hole with $e$
- operational behaviour: evaluate by the operational semantics to the same canonical form
- same behaviour in every context: for every context $C[-]$ it holds that $C[e_1] \Downarrow c \iff C[e_2] \Downarrow c$
Contextual equivalence = two expressions $e_1, e_2$ are contextual equivalent $e_1 \cong_{PCF} e_2$ iff they have the same operational behaviour in every context.

- **context**: an expression with a hole; we write $C[-]$ for contexts and $C[e]$ is the result of replacing the hole with $e$
- **operational behaviour**: evaluate by the operational semantics to the same canonical form
- **same behaviour in every context**: for every context $C[-]$ it holds that $C[e_1] \downarrow c \iff C[e_2] \downarrow c$

We obtain a theorem and proof principle:

**Adequacy**: $\llbracket e_1 \rrbracket = \llbracket e_2 \rrbracket \Rightarrow e_1 \cong_{PCF} e_2$
Scott Domains

A Scott domain is a bounded-complete, $\omega$-algebraic domain. A domain is bounded if it has an upper bound. A domain is bounded-complete if bounded subsets have a join, which is known as a compact element. A domain is $\omega$-algebraic if it is compact: for each directed set $S$, if $c \sqsubseteq \bigsqcup S$, then there exists $s \in S$ such that $c \sqsubseteq s$. A domain is algebraic dcpo if every element is the join of the compact elements below it. A domain is $\omega$-algebraic if the compact elements form a countable set.

All PCF-domains defined earlier are actually Scott domains!
Scott Domains

Scott domain $=$ bounded-complete, $\omega$-algebraic domain
Scott domain = bounded-complete, $\omega$-algebraic domain

- **bounded**: has upper bound
Scott Domains

Scott domain $= \text{bounded-complete, } \omega\text{-algebraic domain}$

- **bounded**: has upper bound
- **bounded-complete**: bounded subsets have a join

All PCF-domains defined earlier are actually Scott domains!
Scott Domains

Scott domain $\equiv$ bounded-complete, $\omega$-algebraic domain

- **bounded**: has upper bound
- **bounded-complete**: bounded subsets have a join
- **compact element**: $c$ is compact if for each directed $S$ it holds that $c \sqsubseteq \bigsqcup S \Rightarrow (\exists s \in S) \ c \sqsubseteq s$
Scott Domains

Scott domain = bounded-complete, ω-algebraic domain

- **bounded**: has upper bound
- **bounded-complete**: bounded subsets have a join
- **compact element**: $c$ is compact if for each directed $S$ it holds that $c \sqsubseteq \bigsqcup S \Rightarrow (\exists s \in S) c \sqsubseteq s$
- **algebraic dcpo**: every element of the dcpo is the join the compact elements below it

All PCF-domains defined earlier are actually Scott domains!
Scott Domains

Scott domain = bounded-complete, $\omega$-algebraic domain

- **bounded**: has upper bound
- **bounded-complete**: bounded subsets have a join
- **compact element**: $c$ is compact if for each directed $S$ it holds that $c \sqsubseteq \bigcup S \Rightarrow (\exists s \in S) \ c \sqsubseteq s$
- **algebraic dcpo**: every element of the dcpo is the join the compact elements below it
- **$\omega$-algebraic**: compact elements form a countable set
Scott Domains

Scott domain = bounded-complete, $\omega$-algebraic domain

- **bounded**: has upper bound
- **bounded-complete**: bounded subsets have a join
- **compact element**: $c$ is compact if for each directed $S$ it holds that $c \sqsubseteq \bigsqcup S \Rightarrow (\exists s \in S)\ c \sqsubseteq s$
- **algebraic dcpo**: every element of the dcpo is the join the compact elements below it
- **$\omega$-algebraic**: compact elements form a countable set

All PCF-domains defined earlier are actually Scott domains!
Extend PCF to PCF+ by adding ‘parallel-or’
Extend PCF to PCF+ by adding ‘parallel-or’

- **Syntax:** \( e \in \text{Exp} ::= T \mid \ldots \mid \text{fix } e \mid e \text{ por } e \)
Extend PCF to PCF+ by adding ‘parallel-or’

- **Syntax:** \( e \in \text{Exp} ::= T \mid \ldots \mid \text{fix } e \mid e \text{ por } e \)

- **Typing:**
  \[
  \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool} \\
  \Gamma \vdash e_1 \text{ por } e_2 : \text{bool}
  \]
Extend PCF to PCF\+ by adding ‘parallel-or’

- **Syntax**: \( e \in \text{Exp} ::= T | \ldots | \text{fix } e | e \text{ por } e \)

- **Typing**: \( \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool} \quad \Gamma \vdash e_1 \text{ por } e_2 : \text{bool} \)

- **Operational semantics**: 
  
  \[
  \begin{array}{c}
  e_1 \Downarrow T \\
  e_2 \Downarrow T \\
  e_1 \Downarrow F \\
  e_2 \Downarrow F
  \end{array}
  \]

  \[
  \begin{array}{c}
  e_1 \text{ por } e_2 \Downarrow T \\
  e_1 \text{ por } e_2 \Downarrow T \\
  e_1 \text{ por } e_2 \Downarrow F
  \end{array}
  \]
PCF+ Parallel-Or

Extend PCF to PCF+ by adding ‘parallel-or’

- **Syntax:** \( e \in \text{Exp ::= } T \mid \ldots \mid \text{fix } e \mid e \text{ por } e \)

- **Typing:**
  \[
  \frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \text{ por } e_2 : \text{bool}}
  \]

- **Operational semantics:**
  \[
  \begin{align*}
  e_1 \Downarrow T & \quad e_2 \Downarrow T & \quad e_1 \Downarrow F & \quad e_2 \Downarrow F \\
  e_1 \text{ por } e_2 \Downarrow T & \quad e_1 \text{ por } e_2 \Downarrow T & \quad e_1 \text{ por } e_2 \Downarrow F
  \end{align*}
  \]

- **Denotational semantics:**
  \[
  \llbracket e_1 \text{ por } e_2 \rrbracket \rho = \begin{cases} 
  \text{true} & \text{if } \llbracket e_1 \rrbracket \rho = \text{true} \text{ or } \llbracket e_2 \rrbracket \rho = \text{true} \\
  \text{false} & \text{if } \llbracket e_1 \rrbracket \rho = \text{false} \text{ and } \llbracket e_2 \rrbracket \rho = \text{false} \\
  \bot & \text{otherwise;}
  \end{cases}
  \]
Full Abstraction

Full abstraction fails for PCF:
\[ J e_1 K = J e_2 K \not\Rightarrow e_1 \sim e_2 = \text{PCF} \]

For PCF+ adequacy still holds:
\[ J e_1 K = J e_2 K \Rightarrow e_1 \sim e_2 = \text{PCF+} \]

But also full abstraction:
\[ J e_1 K = J e_2 K \Leftrightarrow e_1 \sim e_2 = \text{PCF+} \]

The proof of full abstraction uses Scott domains by showing:

▶ All compact elements are definable in PCF+.
▶ If all compact elements are definable, then we have full abstraction.

\[(d \text{ is definable if there is an expression } e \text{ such that } J e K = d).\]
**Full Abstraction**

Full abstraction fails for PCF:

\[ \llbracket e_1 \rrbracket = \llbracket e_2 \rrbracket \not\equiv e_1 \sim_{PCF} e_2. \]

For PCF+, adequacy still holds:

\[ \llbracket e_1 \rrbracket = \llbracket e_2 \rrbracket \Rightarrow e_1 \sim_{PCF+} e_2. \]

The proof of full abstraction uses Scott domains by showing:

▶ All compact elements are definable in PCF+.
▶ If all compact elements are definable, then we have full abstraction.
▶ (\( d \) is definable if there is an expression \( e \) such that \( \llbracket e \rrbracket = d \).)
Full Abstraction

Full abstraction fails for PCF

\[[e_1] = [e_2] \not\sim e_1 \sim_{PCF} e_2.\]

For PCF+ adequacy still holds

\[[e_1] = [e_2] \Rightarrow e_1 \sim_{PCF+} e_2\]
Full Abstraction

Full abstraction fails for PCF

\[ [e_1] = [e_2] \not\iff e_1 \simeq_{\text{PCF}} e_2. \]

For PCF⁺ adequacy still holds

\[ [e_1] = [e_2] \implies e_1 \simeq_{\text{PCF}^+} e_2 \]

but also full abstraction

\[ [e_1] = [e_2] \iff e_1 \simeq_{\text{PCF}^+} e_2. \]
Full Abstraction

Full abstraction fails for PCF

\[ [e_1] = [e_2] \not\iff e_1 \equiv_{PCF} e_2. \]

For PCF+ adequacy still holds

\[ [e_1] = [e_2] \Rightarrow e_1 \equiv_{PCF+} e_2 \]

but also full abstraction

\[ [e_1] = [e_2] \iff e_1 \equiv_{PCF+} e_2. \]

The proof of full abstraction uses Scott domains by showing:

- All compact elements are definable in PCF+.
- If all compact elements are definable, then we have full abstraction.
- \((d \text{ is definable if there is an expression } e \text{ such that } [e] = d.)\)
Overview

1. Introduction
2. PCF
   - language and operational semantics
   - denotational semantics
   - parallel-or and full abstraction
3. PCF with names = PNA
   - motivation and language
   - operational semantics
4. Denotational semantics and full abstraction for PNA
PCF can express computation over booleans, numbers. Lists or trees are not hard to add. What about expressing computation over another programming language with binding forms? Often called meta-programming, we have a meta-level and an object-level language. Often tedious issues with $\alpha$-equivalence in object-level syntax. Important for domain-specific languages and mechanised theorem proving. Possible solution: locally nameless representation.
PCF can express computation over
- booleans, numbers
- lists or trees are not hard to add
PCF can express computation over

- booleans, numbers
- lists or trees are not hard to add

What about expressing computation over another programming language with binding forms?
PCF can express computation over

- booleans, numbers
- lists or trees are not hard to add

What about expressing computation over another programming language with binding forms?

- often called meta-programming
PCF can express computation over

▶ booleans, numbers
▶ lists or trees are not hard to add

What about expressing computation over another programming language with binding forms?

▶ often called meta-programming
▶ have a meta-level and an object-level language
PNA – Motivation

PCF can express computation over

- booleans, numbers
- lists or trees are not hard to add

What about expressing computation over another programming language with binding forms?

- often called meta-programming
- have a meta-level and an object-level language
- often tedious issues with $\alpha$-equivalence in object-level syntax
PNA – Motivation

PCF can express computation over

- booleans, numbers
- lists or trees are not hard to add

What about expressing computation over another programming language with binding forms?

- often called meta-programming
- have a meta-level and an object-level language
- often tedious issues with $\alpha$-equivalence in object-level syntax
- important for domain-specific languages and mechanised theorem proving
PCF can express computation over
- booleans, numbers
- lists or trees are not hard to add

What about expressing computation over another programming language with binding forms?
- often called meta-programming
- have a meta-level and an object-level language
- often tedious issues with $\alpha$-equivalence in object-level syntax
- important for domain-specific languages and mechanised theorem proving
- possible solution: locally nameless representation
Our solution: have names and name abstractions explicitly in the meta-level syntax!
Our solution: have names and name abstractions explicitly in the meta-level syntax!

PNA: Programming with Name Abstractions

\[ \tau \in \text{Typ} ::= \text{bool} \mid \text{nat} \mid \tau \times \tau \mid \tau \rightarrow \tau \mid \text{name} \mid \text{term} \mid \delta \tau \]
Our solution: have names and name abstractions explicitly in the meta-level syntax!

PNA: Programming with Name Abstractions

▷ $\tau \in \text{Typ} ::= \text{bool} \mid \text{nat} \mid \tau \times \tau \mid \tau \rightarrow \tau \mid \text{name} \mid \text{term} \mid \delta \tau$

▷ Expressions

$e \in \text{Exp} ::= T \mid F \mid \ldots \mid \text{fix } e \mid \text{as for PCF}$
Our solution: have names and name abstractions explicitly in the meta-level syntax!

PNA: Programming with Name Abstractions

\[ \tau \in \text{Typ} ::= \text{bool} \mid \text{nat} \mid \tau \times \tau \mid \tau \rightarrow \tau \mid \text{name} \mid \text{term} \mid \tilde{\delta} \tau \]

Expressions

\[ e \in \text{Exp} ::= T \mid F \mid \ldots \mid \text{fix } e \mid \text{as for PCF} \]

\[ a \mid \nu a. e \mid \text{atomic name and local name} \]
Our solution: have names and name abstractions explicitly in the meta-level syntax!

PNA: Programming with Name Abstractions

\[ \tau \in \text{Typ} ::= \text{bool} \mid \text{nat} \mid \tau \times \tau \mid \tau \to \tau \mid \text{name} \mid \text{term} \mid \delta \tau \]

Expressions

\[ e \in \text{Exp} ::= T \mid F \mid \ldots \mid \text{fix } e \mid \text{as for PCF} \]

\[ a \mid \nu a. e \mid \text{atomic name and local name} \]

\[ (e \rightleftharpoons e) \mid e = e \mid \text{name swapping and equality} \]
PNA – Syntax

Our solution: have names and name abstractions explicitly in the meta-level syntax!

PNA: Programming with Name Abstractions

- $\tau \in \text{Typ} ::= \text{bool} \mid \text{nat} \mid \tau \times \tau \mid \tau \rightarrow \tau \mid \text{name} \mid \text{term} \mid \delta \tau$
- Expressions
  - $e \in \text{Exp} ::= T \mid F \mid \ldots \mid \text{fix} \ e \mid a \mid \nu a. \ e \mid (e \Leftrightarrow e) \ e \mid e = e \mid V \ e \mid A \ e \ e \mid L \ e \mid$

- as for PCF
- atomic name and local name
- name swapping and equality
- $\lambda$-terms
Our solution: have names and name abstractions explicitly in the meta-level syntax!

PNA: Programming with Name Abstractions

\[ \tau \in \text{Typ} ::= \text{bool} \mid \text{nat} \mid \tau \times \tau \mid \tau \rightarrow \tau \mid \text{name} \mid \text{term} \mid \delta \tau \]

Expressions

\[ e \in \text{Exp} ::= T \mid F \mid \ldots \mid \text{fix } e \mid a \mid \nu a. \ e \mid (e \Rightarrow e) \ e \mid e = e \mid V \ e \mid A \ e \ e \mid L \ e \mid \text{case } e \ \text{of } (V \ x \rightarrow e \mid A \ x \ x \rightarrow e \mid L \ x \rightarrow e) \mid \text{as for PCF} \]

atomic name and local name

name swapping and equality

\[ \lambda \text{-terms} \]

\[ \lambda \text{-term case} \]
PNA – Syntax

Our solution: have names and name abstractions explicitly in the meta-level syntax!

PNA: Programming with Name Abstractions

\[ \tau \in \text{Typ} ::= \text{bool} | \text{nat} | \tau \times \tau | \tau \to \tau | \text{name} | \text{term} | \delta \tau \]

Expressions

\[ e \in \text{Exp} ::= T | F | \ldots | \text{fix } e | a | \nu a. \ e | (e \leftarrow e) \ e | e = e | \text{as for PCF} \]

atomic name and local name

name swapping and equality

\[ V \ e | A \ e \ e | L \ e | \lambda\text{-terms} \]

\[ \text{case } e \text{ of } (V x \to e \mid A x x \to e \mid L x \to e) | \lambda\text{-term case} \]

\[ \alpha a. \ e | e \ @ e \quad \text{name abstraction and concretion} \]
Our solution: have names and name abstractions explicitly in the meta-level syntax!

PNA: Programming with Name Abstractions

\[ \tau \in \text{Typ} ::= \text{bool} \mid \text{nat} \mid \tau \times \tau \mid \tau \rightarrow \tau \mid \text{name} \mid \text{term} \mid \delta \tau \]

Expressions

\[ e \in \text{Exp} ::= T \mid F \mid \ldots \mid \text{fix } e \mid a \mid \nu a. \ e \mid (e \Rightarrow e) \ e \mid e = e \mid V \ e \mid A \ e \ e \mid L \ e \mid \text{name swapping and equality} \]

\[ \text{as for PCF} \]

\[ \lambda\text{-terms} \]

\[ \text{case } e \text{ of } (V \ x \rightarrow e \mid A \ x \ x \rightarrow e \mid L \ x \rightarrow e) \mid \text{\lambda-term case} \]

\[ \alpha a. \ e \mid e @ e \mid \text{name abstraction and concretion} \]

Binding forms: \( \lambda x : \tau \rightarrow _{\_}, \nu a. _{\_}, \alpha a. _{\_}, \) case e of (V x \rightarrow _{\_} | A x x \rightarrow _{\_} | L x \rightarrow _{\_})
We have two kinds of identifiers now:

- Variables: $x \in V$
- Names: $a \in A$

Both can be substituted and permuted, but $A \cap V = \emptyset$.

Both are countably infinite and can be bound.

Finite name permutation action on syntax:

- $\pi \cdot x = x$
- $\pi \cdot a = \pi(a)$
- $\pi \cdot T = T$
- $\pi \cdot (e_1, e_2) = (\pi \cdot e_1, \pi \cdot e_2)$
- $\pi \cdot \nu_a \cdot e = \nu_{\pi(a)} \cdot \pi \cdot e$
PNA – Names and Variables

We have two kinds of identifiers now

<table>
<thead>
<tr>
<th>Variables</th>
<th>Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \in V$</td>
<td>$a \in A$</td>
</tr>
<tr>
<td>can be substituted</td>
<td>can be permuted</td>
</tr>
<tr>
<td>$A \cap V = \emptyset$</td>
<td>$\pi$ ($a$)</td>
</tr>
<tr>
<td>both are countably infinite</td>
<td>both can be bound</td>
</tr>
</tbody>
</table>

Finite name permutation action on syntax:

$$\pi \cdot x = x$$
$$\pi \cdot a = \pi(a)$$
$$\pi \cdot T = T$$
$$\pi \cdot (e_1, e_2) = (\pi \cdot e_1, \pi \cdot e_2)$$
$$\pi \cdot \nu a. e = \nu \pi(a). \pi \cdot e$$
PNA – Names and Variables

We have two kinds of identifiers now

<table>
<thead>
<tr>
<th>Variables</th>
<th>Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \in \mathbb{V}$</td>
<td>$a \in \mathbb{A}$</td>
</tr>
<tr>
<td>can be substituted</td>
<td>can be permuted</td>
</tr>
<tr>
<td>$\mathbb{A} \cap \mathbb{V} = \emptyset$</td>
<td></td>
</tr>
<tr>
<td>both are countably infinite</td>
<td></td>
</tr>
<tr>
<td>both can be bound</td>
<td></td>
</tr>
</tbody>
</table>

Finite name permutation action on syntax:

$$\pi \cdot x = x$$
$$\pi \cdot a = \pi(a)$$
$$\pi \cdot T = T$$
$$\pi \cdot (e_1, e_2) = (\pi \cdot e_1, \pi \cdot e_2)$$
$$\pi \cdot \nu a. e = \nu(\pi \cdot a). \pi \cdot e$$
\( a \in A \Gamma \vdash a : \text{name} \)

\( a \in A \Gamma \vdash e : \tau \)

\( \Gamma \vdash \nu a.e : \tau \)

\( \Gamma \vdash e_1 : \text{name} \)

\( \Gamma \vdash e_2 : \text{name} \)

\( \Gamma \vdash e_3 : \tau \)

\( \Gamma \vdash (e_1 \equiv e_2) e_3 : \tau \)

\( \Gamma \vdash e_1 : \text{name} \)

\( \Gamma \vdash e_2 : \text{name} \)

\( \Gamma \vdash e_1 = e_2 : \text{bool} \)

\( \Gamma \vdash e : \delta \text{term} \)

\( \Gamma \vdash A e_1 e_2 : \text{term} \)

\( \Gamma \vdash e : \alpha \)

\( \Gamma \vdash e_1 : \delta \tau \)

\( \Gamma \vdash e_2 : \text{term} \)

\( \Gamma \vdash x_1 : \text{name} \)

\( \Gamma \vdash e_1 : \tau \)

\( \Gamma, x_2 : \text{term}, x_2' : \text{term} \vdash e_2 : \tau \)

\( \Gamma, x_3 : \delta \text{term} \vdash e_3 : \tau \)

\( \Gamma \vdash \text{case } e \text{ of } (V x_1 e_1 | A x_2 x_2' e_2 | L x_3 e_3) : \tau \)

\( \Gamma \vdash A \in A \Gamma \vdash e : \tau \)

\( \Gamma \vdash \alpha a.e : \delta \tau \)

\( \Gamma \vdash e_1 : \delta \tau \)

\( \Gamma \vdash e_2 : \text{name} \)

\( \Gamma \vdash e_1 @ e_2 : \tau \)
a ∈ A
\[\frac{}{\Gamma ⊢ a : \text{name}}\]
\[
\begin{align*}
\Gamma & \vdash a : \text{name} \\
\Gamma & \vdash e_1 : \text{name} \quad \Gamma \vdash e_2 : \text{name} \\
\Gamma & \vdash e_1 = e_2 : \text{bool}
\end{align*}
\]
PNA – Typing

\[
\begin{align*}
\frac{a \in A}{\Gamma \vdash a : \text{name}} \\
\frac{\Gamma \vdash e_1 : \text{name} \quad \Gamma \vdash e_2 : \text{name}}{
\Gamma \vdash (e_1 \Rightarrow e_2) \ e_3 : \tau}
\end{align*}
\]

\[
\begin{align*}
\frac{a \in A}{\Gamma \vdash e : \tau} \\
\frac{\Gamma \vdash e_1 : \text{name} \quad \Gamma \vdash e_2 : \text{name}}{
\Gamma \vdash e_1 = e_2 : \text{bool}}
\end{align*}
\]
PNA – Typing

\[
\begin{align*}
  a & \in A \\
\quad \Rightarrow \quad & \Gamma \vdash a : \text{name} \\
\end{align*}
\]

\[
\begin{align*}
  a & \in A \\
\quad \Gamma & \vdash e : \tau \\
\quad \Rightarrow \quad & \Gamma \vdash \nu a. e : \tau \\
\end{align*}
\]

\[
\begin{align*}
  \Gamma & \vdash e_1 : \text{name} \\
  \Gamma & \vdash e_2 : \text{name} \\
  \Gamma & \vdash e_3 : \tau \\
\quad \Rightarrow \quad & \Gamma \vdash (e_1 \Rightarrow e_2) e_3 : \tau \\
\end{align*}
\]

\[
\begin{align*}
  \Gamma & \vdash e_1 : \text{name} \\
  \Gamma & \vdash e_2 : \text{name} \\
\quad \Rightarrow \quad & \Gamma \vdash e_1 = e_2 : \text{bool} \\
\end{align*}
\]

\[
\begin{align*}
  a & \in A \\
\quad \Gamma & \vdash e : \tau \\
\quad \Rightarrow \quad & \Gamma \vdash \alpha a. e : \delta \tau \\
\end{align*}
\]

\[
\begin{align*}
  \Gamma & \vdash e_1 : \delta \tau \\
  \Gamma & \vdash e_2 : \text{name} \\
\quad \Rightarrow \quad & \Gamma \vdash e_1 \odot e_2 : \tau \\
\end{align*}
\]
PNA – Typing

\[
\frac{a \in A}{\Gamma \vdash a : \text{name}} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \nu a. e : \tau}
\]

\[
\frac{\Gamma \vdash e_1 : \text{name} \quad \Gamma \vdash e_2 : \text{name}}{\Gamma \vdash (e_1 \Rightarrow e_2) e_3 : \tau} \quad \frac{\Gamma \vdash e_1 : \text{name} \quad \Gamma \vdash e_2 : \text{name}}{\Gamma \vdash e_1 = e_2 : \text{bool}}
\]

\[
\frac{\Gamma \vdash e : \text{name}}{\Gamma \vdash V e : \text{term}} \quad \frac{\Gamma \vdash e_1 : \text{term} \quad \Gamma \vdash e_2 : \text{term}}{\Gamma \vdash A e_1 e_2 : \text{term}} \quad \frac{\Gamma \vdash e : \delta \text{term}}{\Gamma \vdash L e : \text{term}}
\]

\[
\frac{a \in A \quad \Gamma \vdash e : \tau}{\Gamma \vdash \alpha a. e : \delta \tau} \quad \frac{\Gamma \vdash e_1 : \delta \tau \quad \Gamma \vdash e_2 : \text{name}}{\Gamma \vdash e_1 \odot e_2 : \tau}
\]
PNA – Typing

\[
\frac{a \in A}{\Gamma \vdash a : \text{name}} \quad \frac{a \in A}{\Gamma \vdash \nu a. e : \tau}
\]

\[
\frac{\Gamma \vdash e_1 : \text{name} \quad \Gamma \vdash e_2 : \text{name}}{
\Gamma \vdash (e_1 \Rightarrow e_2) e_3 : \tau}
\]

\[
\frac{\Gamma \vdash e_1 : \text{name} \quad \Gamma \vdash e_2 : \text{name}}{
\Gamma \vdash e_1 = e_2 : \text{bool}}
\]

\[
\frac{\Gamma \vdash e : \text{name}}{\Gamma \vdash V e : \text{term}} \quad \frac{\Gamma \vdash e_1 : \text{term} \quad \Gamma \vdash e_2 : \text{term}}{\Gamma \vdash A e_1 e_2 : \text{term}} \quad \frac{\Gamma \vdash e : \delta \text{term}}{\Gamma \vdash \nu e : \text{term}}
\]

\[
\frac{\Gamma \vdash e : \text{term} \quad \Gamma, x_1 : \text{name} \vdash e_1 : \tau}{\Gamma, x_1 : \text{name}} \quad \frac{\Gamma, x_2 : \text{term}, x_2' : \text{term} \vdash e_2 : \tau \quad \Gamma, x_3 : \delta \text{term} \vdash e_3 : \tau}{\Gamma \vdash \text{case } e \text{ of } (V x_1 \to e_1 \mid A x_2 x_2' \to e_2 \mid L x_3 \to e_3) : \tau}
\]

\[
\frac{a \in A \quad \Gamma \vdash e : \tau}{\Gamma \vdash \alpha a. e : \delta \tau} \quad \frac{\Gamma \vdash e_1 : \delta \tau \quad \Gamma \vdash e_2 : \text{name}}{\Gamma \vdash e_1 \odot e_2 : \tau}
\]
PNA – Example Programs

Representation of $(\lambda a a b) c$

Capture-avoiding substitution $y' [x/y]$

$\lambda y :$ term

$\lambda x :$ name

fix $(\lambda (f : \text{term}) \lambda y' : \text{term})$

case $y'$ of $V x_1$
if $x_1 = x$ then $y$ else $y'$

| A $y_2$ |
| $\text{A} (f y_2) (f y'_2)$ |
| L $z$ | $\text{L} (\alpha a . f (z@a))$. |

Steffen Lösch and Andrew M. Pitts
Semantics for a Pure, Functional Prog. Lang. with Names
Representation of $$(\lambda a \rightarrow a \, b) \, c$$

$$A \,(L \,(\alpha \, a \, . \, A \,(V \, a) \,(V \, b))) \,(V \, c)$$
Representation of \((\lambda a \to a \ b) \ c\)

\[
A \ (L \ (\alpha a. \ A \ (V \ a) \ (V \ b))) \ (V \ c)
\]

Capture-avoiding substitution \(y'[x/y]\)

\[
\begin{align*}
\lambda y : \text{term} & \to \lambda x : \text{name} \to \\
\text{fix} (\lambda (f : \text{term} \to \text{term}) & \to \lambda y' : \text{term} \to \\
\text{case } y' \ of \\
& \ V \ x_1 \to \text{if } x_1 = x \ \text{then } y \ \text{else } y' \\
& \ | \ A \ y_2 \ y'_2 \to A \ (f \ y_2) \ (f \ y'_2) \\
& \ | \ L \ z \to L \ (\alpha a. \ f (z \ @ a)))
\end{align*}
\]
\[ a \in A \]
\[ \frac{}{a \Downarrow a} \]
$\frac{a \in A}{a \Downarrow a}$

$\frac{e_1 \Downarrow a \quad e_2 \Downarrow a}{e_1 = e_2 \Downarrow T}$

$\frac{e_1 \Downarrow a \quad e_2 \Downarrow a' \quad a \neq a'}{e_1 = e_2 \Downarrow F}$
PNA – Operational Semantics

\[
\begin{align*}
  a \in A & \quad \Rightarrow \quad a \Downarrow a & \\
  e_1 \Downarrow a_1 & \quad e_2 \Downarrow a_2 & \quad e_3 \Downarrow c \\
  (e_1 \Rightarrow e_2) & \quad e_3 \Downarrow (a_1 \ a_2) \cdot c
\end{align*}
\]

\[
\begin{align*}
  e_1 \Downarrow a & \quad e_2 \Downarrow a \\
  e_1 = e_2 \Downarrow T
\end{align*}
\]

\[
\begin{align*}
  e_1 \Downarrow a & \quad e_2 \Downarrow a' & \quad a \neq a' \\
  e_1 = e_2 \Downarrow F
\end{align*}
\]
\[
\begin{align*}
a \in A & \quad \frac{}{a \Downarrow a} \\
e_1 \Downarrow a \quad e_2 \Downarrow a & \quad \frac{}{e_1 = e_2 \Downarrow T} \\
e_1 \Downarrow c_1 \quad e_2 \Downarrow c_2 & \quad \frac{}{\text{A } e_1 \text{ e}_2 \Downarrow \text{ A } c_1 \text{ c}_2} \\
e_1 \Downarrow c & \quad e \Downarrow c & \quad \frac{}{\text{L e } \Downarrow \text{ L c}} \\
e_1 \Downarrow a_1 \quad e_2 \Downarrow a_2 \quad e_3 \Downarrow c & \quad \frac{}{(e_1 \equiv e_2) \ e_3 \Downarrow (a_1 \ a_2) \cdot c} \\
e_1 \Downarrow a \quad e_2 \Downarrow a' & \quad a \neq a' \quad \frac{}{e_1 = e_2 \Downarrow F} \\
e_1 \Downarrow c & \quad \frac{}{V \ e \Downarrow V \ c}
\end{align*}
\]
PNA – Operational Semantics

\[
\begin{align*}
a \in A \\
\frac{}{a \downarrow a}
\end{align*}
\]

\[
\begin{align*}
e_1 \downarrow a & \quad e_2 \downarrow a \\
\frac{}{e_1 = e_2 \downarrow T}
\end{align*}
\]

\[
\begin{align*}
e_1 \downarrow c_1 & \quad e_2 \downarrow c_2 \\
\frac{}{A e_1 e_2 \downarrow A c_1 c_2}
\end{align*}
\]

\[
\begin{align*}
e_1 \downarrow c & \\
\frac{}{L e \downarrow L c}
\end{align*}
\]

\[
\begin{align*}
e_1 \downarrow c & \quad e_2 \downarrow a' \\
\frac{a \neq a'}{e_1 = e_2 \downarrow F}
\end{align*}
\]

\[
\begin{align*}
e_1 \downarrow a \\
\frac{}{e_1 = e_2 \downarrow a_1}
\end{align*}
\]

\[
\begin{align*}
e_2 \downarrow a_2 \\
\frac{}{e_3 \downarrow c}
\end{align*}
\]

\[
\begin{align*}
(e_1 \Rightarrow e_2) e_3 \downarrow (a_1 a_2) \cdot c
\end{align*}
\]

\[
\begin{align*}
e_1 \downarrow a \\
\frac{}{e_1 = e_2 \downarrow a}
\end{align*}
\]

\[
\begin{align*}
e_2 \downarrow a \\
\frac{}{e_3 \downarrow c}
\end{align*}
\]

\[
\begin{align*}
e_1 \downarrow a \\
\frac{e_2 \downarrow a'}{a \neq a'}
\end{align*}
\]

\[
\begin{align*}
e_1 \downarrow a \\
\frac{}{e_2 \downarrow a_2}
\end{align*}
\]

\[
\begin{align*}
e_1 \downarrow c & \quad e_2 \downarrow c' \\
\frac{e_1[c/x_1] \downarrow c'}{\text{case } e \text{ of } (V x_1 \rightarrow e_1 | \cdots) \downarrow c'}
\end{align*}
\]

\[
\begin{align*}
e_1 \downarrow A c & \quad e_2 \downarrow c'
\end{align*}
\]

\[
\begin{align*}
e_1 \downarrow A c & \quad e_2[c/x_2, c'/x_2'] \downarrow c'' \\
\frac{e_1 \downarrow A c & \quad e_2 \downarrow c'}{\text{case } e \text{ of } (\cdots | A x_2 x_2' \rightarrow e_2 | \cdots) \downarrow c''}
\end{align*}
\]

\[
\begin{align*}
e_1 \downarrow L c & \quad e_2 \downarrow c'
\end{align*}
\]

\[
\begin{align*}
e_1 \downarrow L c & \quad e_2 \downarrow c'
\end{align*}
\]

\[
\begin{align*}
e_3 \downarrow c' & \quad e_3[c/x_3] \downarrow c'
\frac{}{\text{case } e \text{ of } (\cdots | L x_3 \rightarrow e_3) \downarrow c'}
\end{align*}
\]
\[ a \in A \quad \frac{\qquad \quad}{a \downarrow a} \]
\[
\begin{align*}
\frac{e \downarrow c}{\alpha a. \, e \downarrow \alpha a. \, c}
\end{align*}
\]
\[
\begin{align*}
\frac{e_1 \downarrow a \quad e_2 \downarrow a}{e_1 = e_2 \downarrow T}
\end{align*}
\]
\[
\begin{align*}
\frac{e_1 \downarrow c \quad e_2 \downarrow c}{A \, e_1 \, e_2 \downarrow A \, c_1 \, c_2}
\end{align*}
\]
\[
\begin{align*}
\frac{e_1 \downarrow A \, c \, c' \quad e_2[c/x_2, \, c'/x_2'] \downarrow c''}{\text{case} \, e \, \text{of} \, (\cdots \mid A \, x_2 \, x_2' \rightarrow e_2 \mid \cdots) \downarrow c''}
\end{align*}
\]
\[
\begin{align*}
\frac{e \downarrow L \, c \quad e_3[c/x_3] \downarrow c'}{\text{case} \, e \, \text{of} \, (\cdots \mid L \, x_3 \rightarrow e_3 \mid \cdots) \downarrow c'}
\end{align*}
\]
PNA – Operational Semantics

\[
\begin{align*}
    & a \in A \\
    \quad \frac{a \downarrow a}{e \downarrow c \quad a \vdash c := c'} \\
    \quad \frac{\nu a.\ e \downarrow c'}{e_1 \downarrow a_1 \quad e_2 \downarrow a_2 \quad e_3 \downarrow c} \\
    & (e_1 \cong e_2) \ e_3 \downarrow (a_1 \ a_2) \cdot c
\end{align*}
\]

\[
\begin{align*}
    & e \downarrow c \\
    \quad \frac{\alpha a.\ e \downarrow \alpha a.\ c}{e_1 \downarrow a \quad e_2 \downarrow a} \\
    \quad \frac{e_1 = e_2 \downarrow T}{e_1 \downarrow a \quad e_2 \downarrow a'} \\
    & e_1 \downarrow a \quad e_1 \downarrow a \quad e_2 \downarrow a' \quad a \neq a' \\
    \quad \frac{e_1 = e_2 \downarrow F}{e \downarrow c} \\
    \quad \frac{V e \downarrow V c}{V e \downarrow V c}
\end{align*}
\]

\[
\begin{align*}
    & e \downarrow c_1 \quad e_2 \downarrow c_2 \\
    & A e_1 e_2 \downarrow A c_1 c_2 \\
    & L e \downarrow L c \\
    & \text{case } e \text{ of } (V x_1 \rightarrow e_1 | \cdots) \downarrow c' \\
    & e \downarrow A c c' \\
    & e_2[c/x_2, c'/x_2'] \downarrow c'' \\
    & \text{case } e \text{ of } (\cdots | A x_2 x_2' \rightarrow e_2 | \cdots) \downarrow c''
\end{align*}
\]

\[
\begin{align*}
    & e \downarrow L c \\
    & e_3[c/x_3] \downarrow c' \\
    & \text{case } e \text{ of } (\cdots | L x_3 \rightarrow e_3) \downarrow c'
\end{align*}
\]
PNA – Operational Semantics

\[
\begin{align*}
\frac{a \in A}{a \downarrow a} & \quad \frac{e \downarrow c}{a \setminus c := c'} & \quad \frac{e_1 \downarrow a_1 \quad e_2 \downarrow a_2 \quad e_3 \downarrow c}{(e_1 \equiv e_2) \quad e_3 \downarrow (a_1 \ a_2) \cdot c} \\
\nu a. \ e \downarrow c' & \quad \frac{e_1 \downarrow a \quad e_2 \downarrow a' \quad a \neq a'}{\nu a. (a \ a') \cdot c \downarrow c'} & \quad \frac{e_1 \ @ \ e_2 \downarrow c'}{e \downarrow c} \\
\frac{e_1 \downarrow a \quad e_2 \downarrow a}{e_1 = e_2 \downarrow T} & \quad \frac{e_1 \downarrow a \quad e_2 \downarrow a' \quad a \neq a'}{e_1 = e_2 \downarrow F} & \quad \frac{e_1 \downarrow c}{V \ e \downarrow V \ c} \\
\frac{e_1 \downarrow c_1 \quad e_2 \downarrow c_2}{A \ e_1 \ e_2 \downarrow A \ c_1 \ c_2} & \quad \frac{e \downarrow c}{L \ e \downarrow L \ c} & \quad \frac{e \downarrow V \ c \quad e_1[c/x_1] \downarrow c'}{\text{case } e \text{ of } (V \ x_1 \rightarrow e_1 \ | \cdots) \downarrow c'}
\end{align*}
\]

\[
\begin{align*}
\frac{e \downarrow A \ c \ c'}{\text{case } e \text{ of } (\cdots \ | \ A \ x_2 \ x_2' \rightarrow e_2 \ | \cdots) \downarrow c''} & \quad \frac{e_2[c/x_2, c'/x_2'] \downarrow c''}{\text{case } e \text{ of } (\cdots \ | \ A \ x_2 \ x_2' \rightarrow e_2 \ | \cdots) \downarrow c''} \\
\frac{e \downarrow L \ c \quad e_3[c/x_3] \downarrow c'}{\text{case } e \text{ of } (\cdots \ | \ L \ x_3 \rightarrow e_3) \downarrow c'}
\end{align*}
\]
PNA – Name Restriction Operation
PNA – Name Restriction Operation

- Canonical forms: $c \in \text{Can} ::= \ldots \mid a \mid Vc \mid Acc \mid Lc \mid \alpha a. c$
PNA – Name Restriction Operation

- Canonical forms: \( c \in \text{Can} ::= \ldots \mid a \mid V\ c \mid A\ c\ c \mid L\ c \mid \alpha a.\ c \)

- ‘Odersky-style’ partial name restriction operation \( a \setminus c := c' \):

\[
\begin{align*}
  c = T \mid F \mid 0 \mid S\ c' \\
  \frac{}{a \setminus c := c} \\
  a \setminus (e_1, e_2) := (\nu a.\ e_1, \nu a.\ e_2) \\
  a \neq a' \\
  \frac{}{a \setminus a' := a'}
\end{align*}
\]

\[
\begin{align*}
  a \setminus \lambda x : \tau \to e := \lambda x : \tau \to \nu a.\ e \\
  a \setminus c := c' \\
  a \setminus V\ c := V\ c' \\
  a \setminus c_1 := c'_1 \\
  a \setminus A\ c_1\ c_2 := A\ c'_1\ c'_2 \\
  a \setminus L\ c := L\ c' \\
  a \setminus c := c' \\
  a \neq a' \\
  a \setminus \alpha a'.\ c := \alpha a'.\ c'
\end{align*}
\]
Overview

1. Introduction
2. PCF
   ▶ language and operational semantics
   ▶ denotational semantics
   ▶ parallel-or and full abstraction
3. PCF with names = PNA
   ▶ motivation and language
   ▶ operational semantics
4. Denotational semantics and full abstraction for PNA
Nominal Sets

Theme: Remodel the domain theory with nominal sets.

▶ nominal set = set with finite permutation action · and whose elements are finitely supported.

Examples:

▶ finite support = \(x \) is finitely supported by \(A \subseteq f_A\) if all permutations that preserve \(A\) also preserve \(x\).

Crucial construction: name abstraction sets.

▶ For a nominal set \(X\), define an equivalence relation on \(A \times X\) by \((a_1, x_1) \approx (a_2, x_2)\) iff \((a_1 b) \cdot x_1 = (a_2 b) \cdot x_2\) for some fresh \(b\). Write \(\langle a \rangle x\) for the equiv. class of \((a, x)\).

\[
\left[ A \right]_X = (A \times X) / \approx \text{is the name abstraction set of } X.
\]
Theme: Remodel the domain theory with *nominal sets*.

A *nominal set* is a set with a finite permutation action and whose elements are finitely supported.

**Examples:**
- $\text{Exp}$
- $A$

A finite support $x$ is finitely supported by $A \subseteq f$ if all permutations that preserve $A$ also preserve $x$.

**Crucial construction:** Name abstraction sets

For a nominal set $X$, define an equivalence relation on $A \times X$ by $(a_1, x_1) \approx (a_2, x_2)$ iff $(a_1 b) \cdot x_1 = (a_2 b) \cdot x_2$ for some fresh $b$. Write $\langle a \rangle x$ for the equivalence class of $(a, x)$.

$[A] X = (A \times X) / \approx$ is the name abstraction set of $X$. 

Steffen Lösch and Andrew M. Pitts

Semantics for a Pure, Functional Prog. Lang. with Names
Nominal Sets

Theme: Remodel the domain theory with nominal sets.

- nominal set = set with finite permutation action \( \cdot \) and whose elements are finitely supported

Examples: \( \text{Exp} \), \( \mathbb{A} \)
Theme: Remodel the domain theory with nominal sets.

- **nominal set** = set with finite permutation action \( \cdot \) and whose elements are finitely supported

  *Examples*: \( \text{Exp}, \mathbb{A} \)

- **finite support** = \( x \) is finitely supported by \( A \subseteq_f \mathbb{A} \) if all permutations that preserve \( A \) also preserve \( x \)
Nominal Sets

Theme: Remodel the domain theory with *nominal sets*.

- **nominal set** = set with finite permutation action \( \cdot \) and whose elements are finitely supported

*Examples*: \( \text{Exp}, \ A \)

- **finite support** = \( x \) is finitely supported by \( A \subseteq_f \ A \) if all permutations that preserve \( A \) also preserve \( x \)

Crucial construction: *name abstraction sets*

- For a nominal set \( X \), define an equivalence relation on \( \mathcal{A} \times X \) by \( (a_1, x_1) \approx (a_2, x_2) \) iff \( (a_1 b) \cdot x_1 = (a_2 b) \cdot x_2 \) for some fresh \( b \). Write \( \langle a \rangle x \) for the equiv. class of \( (a, x) \).

- \( [\mathcal{A}]X = (\mathcal{A} \times X)/\approx \) is the name abstraction set of \( X \).
Observation: name abstraction does not preserve directed joins of a nominal partial order, but it preserves all uniform-directed joins.

- **nominal partial order**: nominal set with a partial order \( \sqsubseteq \) satisfying
  \[
  d \sqsubseteq d' \Rightarrow \pi \cdot d \sqsubseteq \pi \cdot d'
  \]

- **uniform-directed set**: directed set in which all elements are supported by a common finite set (the uniform support)

- **Nominal Scott domain**: fs-bounded-complete, \( \omega \)-algebraic, uniform-directed complete, nominal partial order with bottom.

  - **fs-bounded-complete**: finitely supported, bounded subsets have a join
  
  - **uniform-compact element**: for all uniform-directed subsets \( S \)
    \[
    c \sqsubseteq \bigsqcup S \Rightarrow (\exists s \in S) c \sqsubseteq s
    \]

- **uniform-continuous functions**: uniform-directed-join-pres. functions; they form the nominal Scott domain \( D_{uc, fs} \).
Observation: name abstraction does not preserve directed joins of a nominal partial order, but it preserves all uniform-directed joins.
Nominal Domain Theory

Observation: name abstraction does not preserve directed joins of a nominal partial order, but it preserves all uniform-directed joins.

- **nominal partial order**: nominal set with a partial order \( \sqsubseteq \) satisfying \( d \sqsubseteq d' \Rightarrow \pi \cdot d \sqsubseteq \pi \cdot d' \).
Nominal Domain Theory

Observation: name abstraction does not preserve directed joins of a nominal partial order, but it preserves all uniform-directed joins.

- **nominal partial order**: nominal set with a partial order $\sqsubseteq$ satisfying $d \sqsubseteq d' \Rightarrow \pi \cdot d \sqsubseteq \pi \cdot d'$

- **uniform-directed set**: directed set in which all elements are supported by a common finite set (the uniform support)
Nominal Domain Theory

Observation: name abstraction does not preserve directed joins of a nominal partial order, but it preserves all uniform-directed joins.

- **nominal partial order**: nominal set with a partial order \( \sqsubseteq \) satisfying \( d \sqsubseteq d' \Rightarrow \pi \cdot d \sqsubseteq \pi \cdot d' \)
- **uniform-directed set**: directed set in which all elements are supported by a common finite set (the uniform support)

Nominal Scott domain = fs-bounded-complete, \( \omega \)-algebraic, uniform-directed complete, nominal partial order with bottom.
Nominal Domain Theory

Observation: name abstraction does not preserve directed joins of a nominal partial order, but it preserves all uniform-directed joins.

- **nominal partial order**: nominal set with a partial order \( \sqsubseteq \) satisfying \( d \sqsubseteq d' \Rightarrow \pi \cdot d \sqsubseteq \pi \cdot d' \)
- **uniform-directed set**: directed set in which all elements are supported by a common finite set (the uniform support)

Nominal Scott domain = fs-bounded-complete, \( \omega \)-algebraic, uniform-directed complete, nominal partial order with bottom.

- **fs-bounded-complete**: finitely supported, bounded subsets have a join
Observation: name abstraction does not preserve directed joins of a nominal partial order, but it preserves all uniform-directed joins.

- **nominal partial order**: nominal set with a partial order \( \sqsubseteq \) satisfying \( d \sqsubseteq d' \Rightarrow \pi \cdot d \sqsubseteq \pi \cdot d' \)

- **uniform-directed set**: directed set in which all elements are supported by a common finite set (the uniform support)

Nominal Scott domain = fs-bounded-complete, \( \omega \)-algebraic, uniform-directed complete, nominal partial order with bottom.

- **fs-bounded-complete**: finitely supported, bounded subsets have a join

- **uniform-compact element**: \( c \sqsubseteq \bigsqcup S \Rightarrow (\exists s \in S) c \sqsubseteq s \) for all uniform-directed subsets \( S \)
Nominal Domain Theory

Observation: name abstraction does not preserve directed joins of a nominal partial order, but it preserves all uniform-directed joins.

- **nominal partial order**: nominal set with a partial order \( \sqsubseteq \) satisfying \( d \sqsubseteq d' \Rightarrow \pi \cdot d \sqsubseteq \pi \cdot d' \)

- **uniform-directed set**: directed set in which all elements are supported by a common finite set (the uniform support)

Nominal Scott domain = fs-bounded-complete, \( \omega \)-algebraic, uniform-directed complete, nominal partial order with bottom.

- **fs-bounded-complete**: finitely supported, bounded subsets have a join

- **uniform-compact element**: \( c \sqsubseteq \bigsqcup S \Rightarrow (\exists s \in S) \ c \sqsubseteq s \) for all uniform-directed subsets \( S \)

- **uniform-continuous functions**: uniform-directed-join-pres. functions; they form the nominal Scott domain \( D \rightarrow_{uc,fs} E \)
Types denote nominal Scott domains:

\[ \mathbb{J} \mathbb{B} \]

\[ \mathbb{J} \mathbb{N} \]

\[ \mathbb{J} \tau_1 \times \tau_2 \]

\[ \mathbb{J} \tau_1 \rightarrow \tau_2 \]

\[ \mathbb{J} \text{name} \]

\[ \mathbb{J} \text{term} \]

\[ \mathbb{J} \delta \tau \]

Steffen Lösch and Andrew M. Pitts

Semantics for a Pure, Functional Prog. Lang. with Names
Types denote nominal Scott domains:
Types denote nominal Scott domains:

- $\llbracket \text{bool} \rrbracket = \{\text{true, false}\}_\perp$
- $\llbracket \text{nat} \rrbracket = \{0, 1, 2, \ldots\}_\perp$
- $\llbracket \tau_1 \times \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket$
Types denote nominal Scott domains:

- \([\text{bool}] = \{\text{true, false}\} \perp\)
- \([\text{nat}] = \{0, 1, 2, \ldots\} \perp\)
- \([\tau_1 \times \tau_2] = [\tau_1] \times [\tau_2]\)
- \([\tau_1 \to \tau_2] = [\tau_1] \rightarrow_{uc,fs} [\tau_2]\)
Types denote nominal Scott domains:

- $[\text{bool}] = \{\text{true}, \text{false}\}$
- $[\text{nat}] = \{0, 1, 2, \ldots\}$
- $[\tau_1 \times \tau_2] = [\tau_1] \times [\tau_2]$
- $[\tau_1 \rightarrow \tau_2] = [\tau_1] \rightarrow_{uc,fs} [\tau_2]$
- $[\text{name}] = \{a, b, c, \ldots\}$
Types denote nominal Scott domains:

- $\lbrack \text{bool} \rbrack = \{ \text{true}, \text{false} \}_{\bot}$
- $\lbrack \text{nat} \rbrack = \{ 0, 1, 2, \ldots \}_{\bot}$
- $\lbrack \tau_1 \times \tau_2 \rbrack = \lbrack \tau_1 \rbrack \times \lbrack \tau_2 \rbrack$
- $\lbrack \tau_1 \rightarrow \tau_2 \rbrack = \lbrack \tau_1 \rbrack \rightarrow_{uc,fs} \lbrack \tau_2 \rbrack$
- $\lbrack \text{name} \rbrack = \{ a, b, c, \ldots \}_{\bot}$
- $\lbrack \text{term} \rbrack = (\{ t ::= a \mid \lambda a. t \mid t \ t \}=_{\alpha})_{\bot}$
Types denote nominal Scott domains:

- $\llbracket \text{bool} \rrbracket = \{ \text{true, false} \}_\bot$
- $\llbracket \text{nat} \rrbracket = \{ 0, 1, 2, \ldots \}_\bot$
- $\llbracket \tau_1 \times \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket$
- $\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \rightarrow_{uc,fs} \llbracket \tau_2 \rrbracket$
- $\llbracket \text{name} \rrbracket = \{ a, b, c, \ldots \}_\bot$
- $\llbracket \text{term} \rrbracket = (\{ t ::= a \mid \lambda a. t \mid t \ t \} /_=\alpha) _\bot$
- $\llbracket \delta \tau \rrbracket = [A][\tau]$
Expressions denote uniform-continuous, finitely-supported functions:
Expressions denote uniform-continuous, finitely-supported functions:

\[
[a] \rho = a
\]
Expressions denote uniform-continuous, finitely-supported functions:

- $[a] \rho = a$
- $[\nu a. e] \rho = a \setminus ([e] \rho)$ if $a \neq \rho$
Expressions denote uniform-continuous, finitely-supported functions:

- $[[a]] \rho = a$
- $[[\nu a. \, e]] \rho = a \setminus ([[e]] \rho)$ if $a \neq \rho$
- $[[e_1 \equiv e_2] \, e_3] \rho = \begin{cases} (a_1 \, a_2) \cdot ([[e_3]] \rho) & \text{if } [[e_i]] \rho = a_i \in \Lambda \ (i = 1, 2) \\ \bot & \text{otherwise} \end{cases}$
Expressions denote uniform-continuous, finitely-supported functions:

- $[[a]]\rho = a$
- $[[\nu a. e]]\rho = a \setminus ([e]_\rho)$ if $a \neq \rho$
- $[[e_1 \equiv e_2] e_3]_\rho = \begin{cases} (a_1, a_2) \cdot ([e_3]_\rho) & \text{if } [e_i]_\rho = a_i \in \Lambda \ (i = 1, 2) \\ \bot & \text{otherwise} \end{cases}$
- $[[e_1 = e_2]]_\rho = \begin{cases} \text{true} & \text{if } [e_i]_\rho = a \in \Lambda \ (i = 1, 2) \\ \text{false} & \text{if } \bot \neq [e_1]_\rho \neq [e_2] \neq \bot \\ \bot & \text{otherwise} \end{cases}$
Expressions denote uniform-continuous, finitely-supported functions:

- $[a] \rho = a$
- $[\nu a. e] \rho = a \setminus ([e] \rho)$ if $a \neq \rho$
- $[ (e_1 \Rightarrow e_2) e_3 ] \rho = \begin{cases} (a_1, a_2) \cdot ([e_3] \rho) & \text{if } [e_i] \rho = a_i \in A \ (i = 1, 2) \\ \bot & \text{otherwise} \end{cases}$
- $[e_1 = e_2] \rho = \begin{cases} \text{true} & \text{if } [e_i] \rho = a \in A \ (i = 1, 2) \\ \text{false} & \text{if } \bot \neq [e_1] \rho \neq [e_2] \neq \bot \\ \bot & \text{otherwise} \end{cases}$
- $[\forall e] \rho = \begin{cases} [a]_\alpha & \text{if } [e] \rho = a \in A \\ \bot & \text{otherwise} \end{cases}$
Expressions denote uniform-continuous, finitely-supported functions:

- $[[a]]\rho = a$
- $[[\nu a.\ e]]\rho = a \setminus (\llbracket e \rrbracket \rho)$ if $a \not\in \rho$
- $[[e_1 \leadsto e_2 \ e_3]]\rho = \begin{cases} (a_1 a_2) \cdot (\llbracket e_3 \rrbracket \rho) & \text{if } \llbracket e_i \rrbracket \rho = a_i \in \mathbb{A} \ (i = 1, 2) \\ \bot & \text{otherwise} \end{cases}$
- $[[e_1 = e_2]]\rho = \begin{cases} \text{true} & \text{if } \llbracket e_i \rrbracket \rho = a \in \mathbb{A} \ (i = 1, 2) \\ \text{false} & \text{if } \bot \neq \llbracket e_1 \rrbracket \rho \neq \llbracket e_2 \rrbracket \neq \bot \\ \bot & \text{otherwise} \end{cases}$
- $[[\forall e]]\rho = \begin{cases} [a]_\alpha & \text{if } \llbracket e \rrbracket \rho = a \in \mathbb{A} \\ \bot & \text{otherwise} \end{cases}$
- $[[\exists e_1 \ e_2]]\rho = \begin{cases} [t_1 \ t_2]_\alpha & \text{if } \llbracket e_i \rrbracket \rho = [t_i]_\alpha \in \Lambda_\alpha \ (i = 1, 2) \\ \bot & \text{otherwise} \end{cases}$
\[ [\lambda e] \rho = \begin{cases} [\lambda a.t] \alpha & \text{if } [e] \rho = \langle a \rangle[t] \alpha \in [\mathbb{A}]\Lambda \alpha \\ \bot & \text{otherwise} \end{cases} \]

This denotational semantics is adequate:
\[ Je_1 K \rho = Je_2 K \rho \Rightarrow e_1 \sim PNA e_2 \]
\[\begin{align*}
\llbracket L \ e \rrbracket \rho &= \begin{cases} 
[\lambda a. t]_\alpha & \text{if } \llbracket e \rrbracket \rho = \langle a \rangle [t]_\alpha \in [\mathbb{A}]\Lambda_\alpha \\
\bot & \text{otherwise}
\end{cases} \\
\llbracket \text{case } e \text{ of } (V \ x_1 \to e_1 \mid A \ x_2 \ x_2' \to e_2 \mid L \ x_3 \to e_3) \rrbracket \rho &= \\
&= \begin{cases} 
\llbracket e_1 \rrbracket \rho [x_1 \to a] & \text{if } \llbracket e \rrbracket \rho = [a]_\alpha \\
\llbracket e_2 \rrbracket \rho [x_2 \to [t]_\alpha, x_2' \to [t']_\alpha] & \text{if } \llbracket e \rrbracket \rho = [t t']_\alpha \\
\llbracket e_3 \rrbracket \rho [x_3 \to \langle a \rangle [t]_\alpha] & \text{if } \llbracket e \rrbracket \rho = [\lambda a. t]_\alpha \\
\bot & \text{otherwise}
\end{cases}
\end{align*}\]
\[ 
L e \]_\rho = \begin{cases} 
[\lambda a. t]_\alpha & \text{if } [e]_\rho = \langle a \rangle [t]_\alpha \in [A] \Lambda_\alpha \\
\bot & \text{otherwise}
\end{cases} 
\]

\[ 
\text{case } e \text{ of } (V x_1 \to e_1 \mid A x_2 x_2' \to e_2 \mid L x_3 \to e_3)] \rho = 
\begin{cases} 
[e_1]_\rho[x_1 \mapsto a] & \text{if } [e]_\rho = [a]_\alpha \\
[e_2]_\rho[x_2 \mapsto [t]_\alpha, x_2' \mapsto [t']_\alpha] & \text{if } [e]_\rho = [t t']_\alpha \\
[e_3]_\rho[x_3 \mapsto \langle a \rangle [t]_\alpha] & \text{if } [e]_\rho = [\lambda a. t]_\alpha \\
\bot & \text{otherwise}
\end{cases} 
\]

\[ 
\alpha a. e \]_\rho = \langle a \rangle ([e]_\rho) \quad \text{if } a \not\in \rho 
\]
This denotational semantics is adequate:

\[ \llbracket e_1 \rrbracket \rho = \llbracket e_2 \rrbracket \rho \Rightarrow e_1 \sim e_2 \]

\[ \llbracket \lambda a. e \rrbracket \rho = \langle a \rangle \llbracket e \rrbracket \rho \quad \text{if } a \not\in \rho \]

\[ \llbracket e_1 \odot e_2 \rrbracket \rho = (\llbracket e_1 \rrbracket \rho) \odot (\llbracket e_2 \rrbracket \rho) \]
PNA – Denotational Semantics (3)

\[ [L \ e]_\rho = \begin{cases} 
  [\lambda a.t]_\alpha & \text{if } [e]_\rho = \langle a \rangle [t]_\alpha \in [\Lambda] \Lambda_\alpha \\
  \bot & \text{otherwise}
\end{cases} \]

\[ [\text{case } e \text{ of } (V\ x_1 \to e_1 \mid A \ x_2 \ x'_2 \to e_2 \mid L \ x_3 \to e_3)]_\rho = \begin{cases} 
  [e_1]_\rho[x_1 \mapsto a] & \text{if } [e]_\rho = [a]_\alpha \\
  [e_2]_\rho[x_2 \mapsto [t]_\alpha, x'_2 \mapsto [t']_\alpha] & \text{if } [e]_\rho = [t \ t']_\alpha \\
  [e_3]_\rho[x_3 \mapsto \langle a \rangle [t]_\alpha] & \text{if } [e]_\rho = [\lambda a.t]_\alpha \\
  \bot & \text{otherwise}
\end{cases} \]

\[ [\alpha a. \ e]_\rho = \langle a \rangle ([e]_\rho) \quad \text{if } a \# \rho \]

\[ [e_1 \circ e_2]_\rho = ([e_1]_\rho) \circ ([e_2]_\rho) \]

This denotational semantics is \textbf{adequate}: 

\[ [e_1] = [e_2] \Rightarrow e_1 \equiv_{\text{PNA}} e_2 \]
PNA+ = PNA + parallel-or + name-exists + definite-description

**Syntax**:
\[ e \in \text{Exp} ::= \ldots | e \text{por} e | \text{ex}.e | \text{the}.e \]

**Typing**:
\[
\begin{align*}
\Gamma & \vdash e : \text{bool} \\
\Gamma, x : \text{name} & \vdash \text{ex}.e : \text{bool} \\
\Gamma & \vdash \text{the}.e : \text{name} \\
\end{align*}
\]

**Operational semantics**:
\[
\begin{align*}
e[a/x] & \Downarrow T \\
\text{ex}.e & \Downarrow T \\
\begin{array}{c}
e[a/x] \\
\text{the}.e & \Downarrow a' \# (e, a) \\
\end{array} \\
F & \Downarrow a' \# (e, a) \\
\end{align*}
\]

Steffen Lösch and Andrew M. Pitts
Semantics for a Pure, Functional Prog. Lang. with Names
PNA+

PNA+ = PNA + parallel-or + name-exists + definite-description
PNA+ = PNA + parallel-or + name-exists + definite-description

- *Syntax:* \( e \in \text{Exp} ::= \ldots | e \text{ por } e | \text{ex}\, x.\, e | \text{the}\, x.\, e \)
PNA+ = PNA + parallel-or + name-exists + definite-description

- **Syntax:** \( e \in \text{Exp} ::= \ldots \mid e \text{ por } e \mid \text{ex } x. e \mid \text{the } x. e \)

- **Typing:**
  \[
  \Gamma, x : \text{name} \vdash e : \text{bool} \quad \quad \Gamma, x : \text{name} \vdash e : \text{bool} \\
  \frac{}{\Gamma \vdash \text{ex } x. e : \text{bool}} \quad \quad \frac{}{\Gamma \vdash \text{the } x. e : \text{name}}
  \]
PNA+ = PNA + parallel-or + name-exists + definite-description

- **Syntax:** \( e \in \text{Exp} ::= \ldots | e \text{ por } e | \text{ex } x . e | \text{the } x . e \)

- **Typing:**
  \[
  \Gamma, x : \text{name} \vdash e : \text{bool} \qquad \Gamma, x : \text{name} \vdash e : \text{bool}
  \]
  \[
  \frac{}{\Gamma \vdash \text{ex } x . e : \text{bool}} \qquad \frac{}{\Gamma \vdash \text{the } x . e : \text{name}}
  \]

- **Operational semantics:**
  \[
  \frac{e[a/x] \Downarrow T}{\text{ex } x . e \Downarrow T}
  \]
  \[
  \frac{a' \neq e \quad (\forall b \in \text{fn}(e) \cup \{a'\})}{e[b/x] \Downarrow F}
  \]
  \[
  \frac{e[a/x] \Downarrow T \quad a' \neq (e, a) \quad (\forall b \in (\text{fn}(e) - \{a\}) \cup \{a'\})}{e[b/x] \Downarrow F}
  \]
  \[
  \frac{\text{the } x . e \Downarrow a}{\text{ex } x . e \Downarrow F}
  \]
PNA+ (2)

- Denotational semantics:
PNA+ (2)

- **Denotational semantics:**

\[
\begin{align*}
\llbracket \text{ex} \; x . \; e \rrbracket_\rho &= \begin{cases} 
  \text{true} & \text{if } (\exists a \in A) (\llbracket e \rrbracket_\rho) a = \text{true} \\
  \text{false} & \text{if } (\forall a \in A) (\llbracket e \rrbracket_\rho) a = \text{false} \\
  \bot & \text{otherwise.}
\end{cases}
\end{align*}
\]

PNA+ is fully abstract:

\[
\llbracket e_1 \rrbracket_\rho = \llbracket e_2 \rrbracket_\rho \iff e_1 \sim =_{PNA+} e_2.
\]

The full abstraction proof is similar to the PCF+ proof, but we only show compact definability at a subset of all types, called the simple types. We use definable retracts to finish the proof.
Denotational semantics:

\[
\begin{align*}
\llbracket \text{ex } x. \ e \rrbracket \rho &= \begin{cases} 
    \text{true} & \text{if } (\exists a \in A) \ (\llbracket e \rrbracket \rho) \ a = \text{true} \\
    \text{false} & \text{if } (\forall a \in A) \ (\llbracket e \rrbracket \rho) \ a = \text{false} \\
    \bot & \text{otherwise.}
\end{cases} \\
\llbracket \text{the } x. \ e \rrbracket \rho &= \begin{cases} 
    a & \text{if } (\llbracket e \rrbracket \rho) = eq_a \text{ for some } a \in A \\
    \bot & \text{otherwise.}
\end{cases}
\end{align*}
\]
Denotational semantics:

\[
\begin{align*}
\llbracket \text{ex} \ x \ . \ \mathit{e} \rrbracket_{\rho} &= \begin{cases} 
\text{true} & \text{if } (\exists a \in A) \ (\llbracket \mathit{e} \rrbracket_{\rho})_a = \text{true} \\
\text{false} & \text{if } (\forall a \in A) \ (\llbracket \mathit{e} \rrbracket_{\rho})_a = \text{false} \\
\bot & \text{otherwise.}
\end{cases} \\
\llbracket \text{the} \ x \ . \ \mathit{e} \rrbracket_{\rho} &= \begin{cases} 
a & \text{if } (\llbracket \mathit{e} \rrbracket_{\rho}) = \text{eq}_a \text{ for some } a \in A \\
\bot & \text{otherwise.}
\end{cases}
\end{align*}
\]

PNA$^+$ is fully abstract:

\[
\llbracket \mathit{e}_1 \rrbracket = \llbracket \mathit{e}_2 \rrbracket \iff \mathit{e}_1 \equiv_{\text{PNA}^+} \mathit{e}_2.
\]
Denotational semantics:

\[ \text{let } \text{the } x \text{. } e \text{ at } \rho = \begin{cases} a & \text{if } (\llbracket e \rrbracket_\rho) = \text{eq}_a \text{ for some } a \in A \\ \bot & \text{otherwise.} \end{cases} \]

PNA+ is fully abstract:

\[ \llbracket e_1 \rrbracket = \llbracket e_2 \rrbracket \iff e_1 \upharpoonright_{\text{PNA+}} e_2. \]

The full abstraction proof is similar to the PCF+ proof, but we only show compact definability at a subset of all types, called the simple types. We use definable retracts to finish the proof.
Conclusions

- Used nominal sets to enhance a semantic theory
- Obtained a more expressive language
- Retrieved classical results
- Complications from nominal sets were somehow orthogonal
Conclusions

- Used nominal sets to enhance a semantic theory
Conclusions

- Used nominal sets to enhance a semantic theory
- Obtained a more expressive language
Conclusions

- Used nominal sets to enhance a semantic theory
- Obtained a more expressive language
- Retrieved classical results
Conclusions

- Used nominal sets to enhance a semantic theory
- Obtained a more expressive language
- Retrieved classical results
- Complications from nominal sets were somehow orthogonal