A Locally Nameless Theory of Objects

Ludovic Henrio, Florian Kammüller, Bianca Lutz, and Henry Sudhof

1. Introduction: ς-calculus and De Bruijn notation
2. locally nameless technique
3. formalization in Isabelle and proofs

SAFA workshop – Oct 2010

Context

Calculi abstract away real programming languages:

- Proofs made on the calculus allow optimisation and ensure properties on real programs

Use of theorem prover to increase confidence in those proofs

A general problem is the representation of variables

We focus here on a simple object language

Functional ς-calculus

Syntax

Each method is a function with a parameter: “self”

$a, b ::=$	x_j	variable
	$\mid \; [l_j = \varsigma(x_j)b_j]^{j \in 1..n}$	object definition
	$\mid \; a.l_j$	$(j \in 1..n)$ method call
	$\mid \; a.l_j := \varsigma(x)b$	$(j \in 1..n)$ update

Semantics (Abadi - Cardelli)

Let $o \equiv [l_j = \varsigma(x_j)b_j]^{j \in 1..n}$ (l_j distinct).

o	is an object with method names l_j and methods $\varsigma(x_j)b_j$	
$o.l_j$	$\rightarrow_\beta b_j\{x_j \leftarrow o\}$	$(j \in 1..n)$ selection / method call
$o.l_j := \varsigma(x)b$	$\rightarrow_\beta [l_j = \varsigma(x)b, l_i = \varsigma(x_i)b_i^{i \in (1..n)-\{j\}}]$	$(j \in 1..n)$ update / override

Why functional? ➔ updating a field creates a new object (copy)

An Example

$$\begin{tikzcd}
{[X = \varsigma(x)[], getX = \varsigma(x)x.X].getX} \arrow[d, "\beta"] \\
\varsigma(x)x.X
\end{tikzcd}$$

An Example

$$[X = \varsigma(x)[], getX = \varsigma(x)x.X].getX$$

$$\downarrow_{\beta}$$

$$[X = \boxed{\varsigma(x)[]}, getX = \varsigma(x)x.X].X$$

$$\downarrow_{\beta}$$

$$[]$$

$$\boxed{o.l_j \qquad \rightarrow_{\beta} b_j\{x_j \leftarrow o\}}$$

What are De Bruijn Indices?

De Bruijn indices avoid having to deal with α**-conversion**

$[l = \varsigma(x)x]$ equivalent to $[l = \varsigma(y)y]$

Variables are natural numbers depending on the **depth of the parameter**

$$[l = \varsigma(x)x] \qquad [l = \varsigma(x)[l' = \varsigma(y)x]]$$

$$\downarrow \qquad \downarrow$$

$$[l\ [l = \varsigma(0)]0] \qquad [l = \varsigma[l(x)[l[l' = \varsigma(1)]1]]$$

also represents $[l = \varsigma(y)y]$

Why De Bruijn Indices?

Unique representation -> avoids dealing with alpha conversion

Drawbacks:

- Terms are “ugly” ➔ We are interested in general properties / not for extracting an interpreter …
- Definition of *subst* and *lift:* semantics more complex
- Proofs of many additional (easy) lemmas

Advantages

- Established approach
- Reuse Nipkow’s framework for confluence of the λ-calculus

Alternative approaches, e.g. locally nameless

2 – Locally Nameless technique

What is locally nameless technique?

Bound variables are represented by their De Bruijn index

Free variables are represented by a “usual” variable

$$[l = \varsigma(x)x.l := \varsigma(y)z]$$

$$\downarrow$$

$$[l = 0.l := z]$$

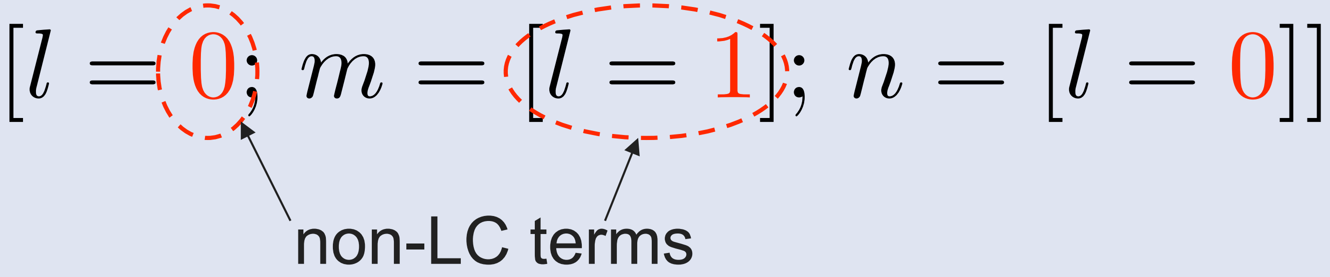
manipulate only *locally closed terms* i.e. all indexed variables must be bound

$[l = 1]$ and $[m = [l = 2]]$ are forbidden

Opening and Closing

open and *close* change between bound and free variables

helps maintain the “locally closed” invariant



$$[l = 0;\ m = [l = 1];\ n = [l = 0]]$$

non-LC terms

open	close
$0^{[x]} \longrightarrow x$	$[x]x \longrightarrow 0$
$[l = 1]^{[x]} \longrightarrow [l = x]$	$[x][l = x] \longrightarrow [l = 1]$
$[l = 0]^{[x]} \longrightarrow [l = 0]$	$[x][l = 0] \longrightarrow [l = 0]$

A method parameter

Syntax

$$
\begin{array}{llr}
a, b ::= & x & \text{Variable} \\
 & \mid [l_j = \varsigma(x_j, y_j) b_j]^{j \in 1..n} & \text{object definition} \\
 & \mid a.l_j(b) & (j \in 1..n)\ \text{method call} \\
 & \mid a.l_j := \varsigma(x, y) b & (j \in 1..n)\ \text{update}
\end{array}
$$

open

$\mathtt{tt}^{\mathtt{[s,p]}}$

close

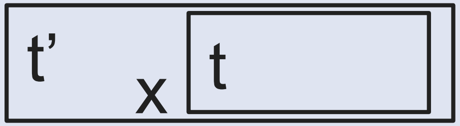
$\mathtt{[s,p]\ t}$

Cofinite Quantification

When specifying semantics or proving properties, we need to open terms: $P(t^{[x]})$

x cannot be taken randomly, an idea:

$$\exists x \notin FV(t).\ P(t^{[x]})$$



Typically, proofs by induction, we must prove:

$$\exists x \notin FV(t).\ P(t^{[x]}) \rightarrow \exists x \notin FV(t').\ P(t'^{[x]})$$

Sometimes impossible if t'≠t, similar problem for: $\forall x \notin FV$

We use cofinite quantification:

$$\exists L \text{ finite}.\forall x \notin L.\ P(t^{[x]})$$

3 – Semantics and Properties

Semantics with cofinite quantification

Reduce inside update (adapted for self+parameter):

$$\frac{t^{[\mathtt{x},\ \mathtt{y}]} \rightarrow_{\varsigma} t'' \qquad t' = \varsigma[\mathtt{x},\ \mathtt{y}]t''' \qquad \mathtt{lc}\ o}{o.l := t \rightarrow_{\varsigma} o.l := t'}$$

⬇

$$\frac{\begin{array}{c} t^{[\mathtt{x},\ \mathtt{y}]} \rightarrow_{\varsigma} t'' \qquad t' = \varsigma[\mathtt{x},\ \mathtt{y}]t'' \qquad \text{finite } L \\ \forall x\ y.\ x{\neq}y \wedge x, y{\notin}\ L \qquad \mathtt{lc}\ o \end{array}}{o.l := t \rightarrow_{\varsigma} o.l := t'}$$

In Isabelle

```
datatype sterm =
        Bvar bVariable
      | Fvar fVariable
      | Obj (Label ⇒_f sterm) type
      | Call sterm Label sterm
      | Upd sterm Label sterm
```

```
| beta_UpdR:
```

$$\llbracket\ \texttt{finite L};\ \forall\ \texttt{s p.}\ \texttt{s} \notin \texttt{L} \land \texttt{p} \notin \texttt{L} \land \texttt{s} \neq \texttt{p} \longrightarrow (\exists\ \texttt{t''.}\ \texttt{t}^{[\texttt{Fvar s,Fvar p}]} \to_{\varsigma} \texttt{t''} \land \texttt{t'=}\ \varsigma\ \texttt{[s,p]t''});\ \texttt{lc u}\ \rrbracket \Longrightarrow \texttt{Upd u l t} \to_{\varsigma} \texttt{Upd u l t'}$$

Properties and Proofs

- Translated proofs for De Bruijn:
  - Confluence
  - Typing: subject reduction and progress
- Different lemmas:
  - lifting and manipulation of indices for De Bruijn
  - Translation between free and bound variables for LN
- Not particularly shorter, but LN more precise
- Induction scheme more complex due to more complex semantics (cofinite quantification)

Conclusion on LN representation

- New concepts wrt de Bruijn:
  - opening and closing
  - locally closed terms (precondition of many lemmas and semantic rules)
  - cofinite quantification
- Better structure, accuracy, and understanding:
  - Distinction between free and bound variables
  - Cofinite quantification
- LN adapted to objects and to multiple parameters
- Terms can be written in a similar manner as paper version (using closing)

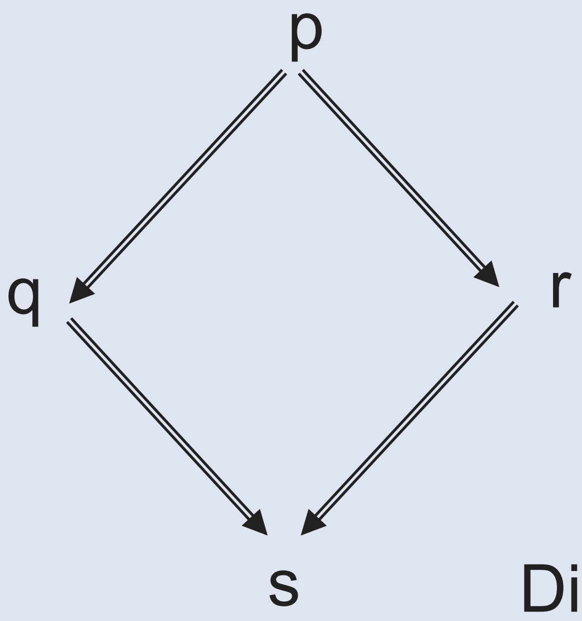
Other techniques?

- Nominal techniques:
  - Terms are identified as a set bijective to all terms factorised by alpha-equivalence
  - There must be a finite support for a term t
  - Well supported in Isabelle but not adapted to finite maps for the moment
- Higher Order Abstract Syntax
  - binders represented by binders of the meta-level
  - not very convenient in our case

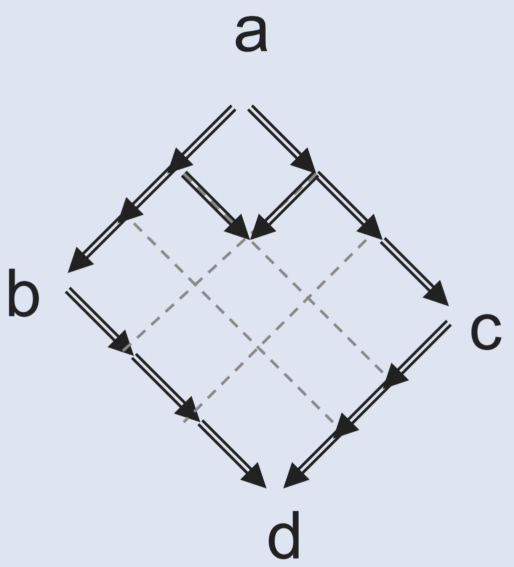
Confluence Principles

Ensures that all computations are equivalent (same result)

Generally based on a **diamond property**:

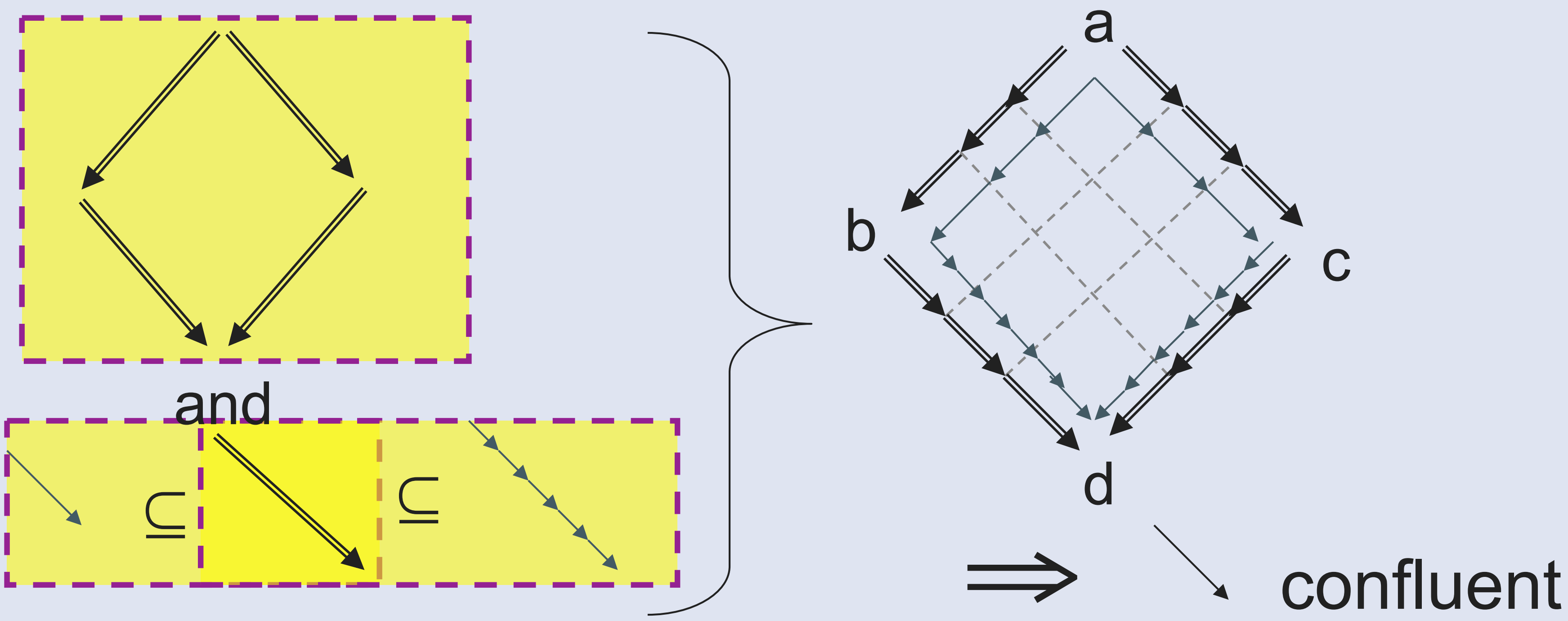


Diamond ⟹ ⇘ confluent:



Confluence Principles (2)

In general we have to introduce a new reduction that verifies the diamond property



$$\llbracket\ \mathtt{diamond}\ \mathtt{R};\ \mathtt{T} \subseteq \mathtt{R};\ \mathtt{R} \subseteq \mathtt{T}^{*}\ \rrbracket \Longrightarrow \mathtt{confluent}\ \mathtt{T}$$

Confluence of the ς-calculus

- Based on Nipkow’s framework: Confluence for the λ-calculus
  - Useful **lemmas**: commute, Church-Rosser, diamond
  - **Structure** of a confluence proof in Isabelle
- Definition of a parallel reduction $\Rightarrow_\beta$ (verifies diamond)
  - Like for λ-calculus, can reduce all sub-terms in parallel

```
upd:  ⟦ s ⇒β s’; t ⇒β t’ ⟧ ⟹ Upd s l t ⇒β Upd s’ l t’
```

  - Also includes $\longrightarrow_\beta$ (semantics of the ς-calculus)

```
upd’: ⟦ Obj s ⇒β Obj s’; t ⇒β t’ ⟧
      ⟹ (Upd (Obj s) l t)  ⇒β (Obj (s’ [l := t’]))
```

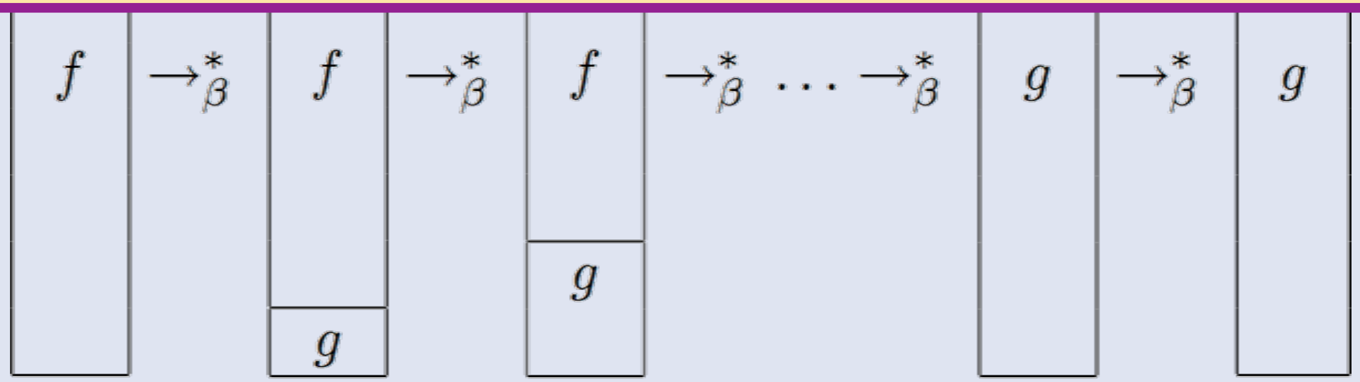
Reducing in Parallel inside Object

Subgoal (looks trivial but proof is tricky):

$\llbracket \texttt{length f = length g; } \forall \texttt{ l < length f. f!l } \rightarrow_\beta^* \texttt{ g!l } \rrbracket \Longrightarrow \texttt{Obj f } \rightarrow_\beta^* \texttt{ Obj g}$

ς-calculus confluence proof similar to Nipkow’s framework but:

- Much less automatic
- Difference of granularity between lists of terms and objects
- More cases for diamond (more constructors/rules)



In the Meantime …

- Objects as **finite maps** from labels to methods instead of **lists** of methods
  - Definition of finite maps and a new induction principle
  - Closer to original ς-calculus (syntax and semantics); new recurrence principle on terms
- Formalization of the basic type system for the functional ς-calculus
  - Typing rules (Abadi - Cardelli)
  - Subject reduction, progress (no stuck configuration)

Todo List

Remove De Bruijn indices → “nominal techniques”?

Introduce methods with a parameter: $\varsigma(x,y)$ / $a.l(b)$

Apply to other results on object languages (concurrence, mobility, …)

→ A base model for Aspect Oriented Programming

Towards Distribution

A model for the ASP calculus in Isabelle; ASP formalizes:

- Active objects (AO) without shared memory
- AO is the entry point and the master object of the activity
- Communicating by asynchronous method calls with futures

Currently:

- Definition of a functional ASP in Isabelle
- Proof of well-formedness of the reduction (no creation of reference to non-existing active objects or futures)

To do ….

- A type system for ASP
- Proof of confluence for the functional ASP
- Extension of the concurrency in the functional calculus
- Case of the imperative ASP calculus …

Conclusion

A formalization of the ς-calculus in Isabelle

A confluence proof for the functional ς-calculus

- Parallel reduction inside objects

A base framework for developments on objects, confluence and concurrency

A lot of possible applications (distribution / typing / AOP …)

Experiments on Isabelle (few months development)

- User-friendly, relatively fast development
- Finding the right structure/representation is crucial
- Difficulties when modifying / reusing code

http://www.cs.tu-berlin.de/~flokam/isabelle/sigma/