Compiling Contextual Objects: Bringing Higher-Order Abstract Syntax to Programmers

Francisco Ferreira, Stefan Monnier and Brigitte Pientka

McGill University/Université de Montréal

Motivation

- We want to compile programs that manipulate other programs.

There are several approaches to representing binders, for example:

- De Bruijn indices
- Nominal approaches[Pitts, 2001]
- Higher Order Abstract Syntax (HOAS) [Harper, Honsell, Plotkin 1993]

Related Work

- When writing proofs:
  - Pfenning and Schürmann [1999], the Twelf system.
  - Felty and Momigliano [2012], the Hybrid using HOAS.
  - Urban [2008] presents Isabelle/Nominal to reason about structures with names in Isabelle.
- For programming
  - Pouillard and Pottier [2010] present an abstract representation that can be instantiated with several concrete ones.
  - Shinwell et al. [2003]; Washburn and Weirich [2008]; Westbrook et al. [2011] with powerful support for programming with binders but no support for dependent types.
  - Chlipala [2008] with Parametric HOAS that supports a form of weak-HOAS in an existing proof assistant with support for dependent types, however manipulation of open terms is problematic.
- Prog. languages which support HOAS and dependent types.
  - Powolsky and Schürmann [2008] presented Delphin that uses LF to represent binders with HOAS.
  - Pientka [2008] introduced **Beluga**.

Contributions

- A framework to compile programs manipulating binders with powerful pattern matching under binders, based on the notion of contextual objects. An essential step to add support for HOAS to existing programming languages.
- Helps bridge the gap between higher-order representations and traditional first-order representations by converting first into a high-level first-order representation that leaves the concrete representation abstract.
- The compiler, that generates code where binders use de Bruijn indices or names, sets up the stage for choosing dynamically between the optimal one in each program.

What is Beluga?

- A language that supports specifications in the logical framework LF[Harper, Honsell, Plotkin 1993]. A setting that supports HOAS.
- A dependently typed, functional programming language that:
  - Embeds LF objects together with a context
  - Abstracts over contexts
  - Supports pattern matching over LF terms and contexts.

How it's done?

Dependent Types

Dependency Erasure to transform types into a simply typed language [Harper, 2005]

HOAS

A conversion to a "fresh-style" representation using worlds and links [Pouillard, Pottier, 2010]

Pattern Matching Compilation

Using a traditional approach with more rules supporting the more expressive patterns[Maranget, 2008]

Front end → Dependency erasure → Fresh-Style → Pattern matching → ... ↗ de Bruijn indices ↘ Names

Example: encoding the Simply Typed λ-Calculus

- A standard LF specification using HOAS.

```
datatype exp : tp → type =
| app : exp (arr A B) → exp A → exp B
| lam : (exp A → exp B) → exp (arr A B);

datatype db : tp → type =
| one : db A
| shift: db A → db A
| lam' : db B → db (arr A B)
| app' : db (arr A B) → db A → db B;
```

Contextual objects

- As we traverse binders in terms we deal with open objects. Contextual objects carry contexts that allow us to reason about free variables.

let m = [. lam λx. (lam λy. app y x)];

Contextual objects

- As we traverse binders in terms we deal with open objects. Contextual objects carry contexts that allow us to reason about free variables.

let m = [. lam $\boxed{\lambda\text{x. (lam } \lambda\text{y. app y x}}$)];
let n = [$\boxed{\text{x : exp T}}$. (lam λy. app y x)];

Contextual objects

- As we traverse binders in terms we deal with open objects. Contextual objects carry contexts that allow us to reason about free variables.

let m = [. lam λx. (lam λy. app y x)];
let n = [x : exp T . (lam λy. app y x)];

Contextual Objects

$$\frac{\Psi \vdash M : A}{[\Psi.M] : [\Psi.A]}$$

- LF object M in context Ψ, i.e. all variables occurring in M are within the scope of Ψ.

Translating HOAS to de Bruijn

```
schema ctx = exp T;
rec hoas2db : (g:ctx) [g. exp T] → [. db T] =
```

Translating HOAS to de Bruijn

```
schema ctx = exp T;
rec hoas2db : (g:ctx) [g. exp T] → [. db T] =
fn e ⇒ case e of
        | [g. app (E1 .. ) (E2 .. )] ⇒
             let [. F1] = hoas2db [g. E1 ..] in
             let [. F2] = hoas2db [g. E2 ..] in
             [. app' F1 F2];
```

Translating HOAS to de Bruijn

```
schema ctx = exp T;
rec hoas2db : (g:ctx) [g. exp T] → [. db T] =
fn e ⇒ case e of
      | [g. app (E1 .. ) (E2 .. )] ⇒
          let [. F1] = hoas2db [g. E1 ..] in
          let [. F2] = hoas2db [g. E2 ..] in
          [. app' F1 F2];
      | [g. lam (λx. E .. x)] ⇒
          let [. F] = hoas2db [g,x:exp _ . E .. x] in
          [. lam' F]
```

Translating HOAS to de Bruijn

```
schema ctx = exp T;
rec hoas2db : (g:ctx) [g. exp T] → [. db T] =
fn e ⇒ case e of
      | [g. app (E1 .. ) (E2 .. )] ⇒
          let [. F1] = hoas2db [g. E1 ..] in
          let [. F2] = hoas2db [g. E2 ..] in
          [. app' F1 F2];
      | [g. lam (λx. E .. x)] ⇒
          let [. F] = hoas2db [g,x:exp _ . E .. x] in
          [. lam' F]
      | [g, x:exp T . x] ⇒ [. one]
      | [g, x:exp T . #p ..] ⇒
          let [. F] = hoas2db [g. #p ..] in
          [. shift F]
```

What is the "Fresh-Look Representation"?

An idea adapted from "A fresh look at programming with names and binders" [Pouillard, Pottier, 2010]
An abstract representation of names and binders:

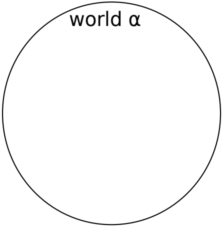
1. "name abstractions cannot be violated" or "the representation of two α-equivalent terms cannot be distinguished"
2. "names do not escape their scope"
3. "names with different scopes cannot be mixed"

What is the "Fresh-Look Representation"?

An idea adapted from "A fresh look at programming with names and binders" [Pouillard, Pottier, 2010]
An abstracted representation of names and binders, with the following characteristics:

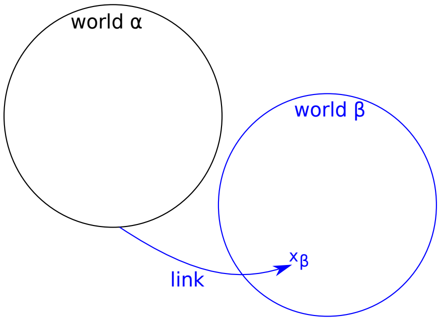
1. ~~"name abstractions cannot be violated" or "the representation of two α-equivalent terms cannot be distinguished"~~
2. ~~"names do not escape their scope"~~
3. ~~"names with different scopes cannot be mixed"~~
4. Easy to convert back to name and de Bruijn style variables generating efficient code.

The Fresh Look Representation



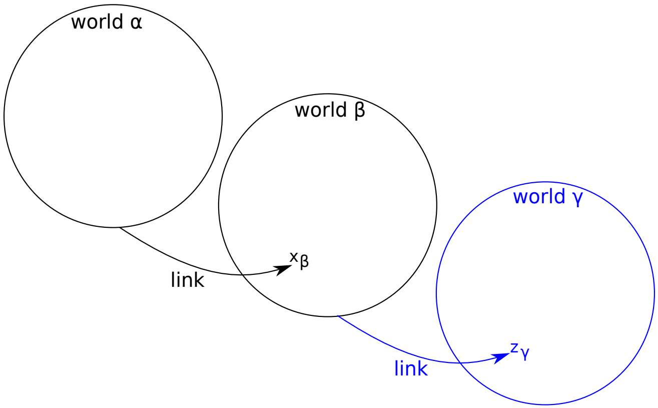
- Worlds are inhabited by names.
- The empty world is the world of closed terms.

The Fresh Look Representation



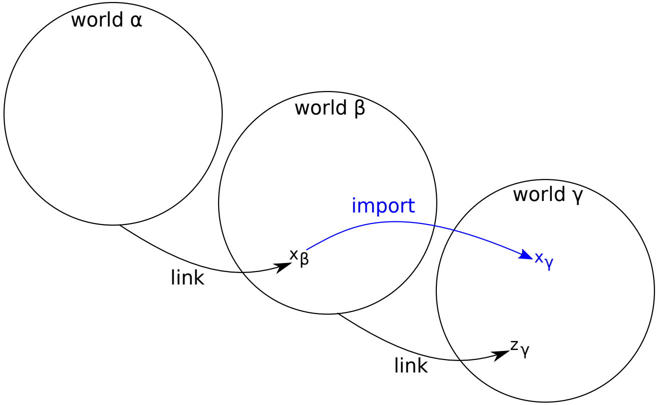
- Links relate a world to a bigger world with one extra name.

The Fresh Look Representation



- Links can be chained to create worlds with many names.

The Fresh Look Representation



- Names from other worlds need to be imported before using them.

The Pipeline

$$\text{Front end} \longrightarrow \text{Dependency erasure} \longrightarrow \text{Fresh-Style} \longrightarrow \text{Pattern matching} \longrightarrow \ldots \begin{array}{l} \nearrow \text{de Bruijn indices} \\ \searrow \text{Names} \end{array}$$

Dependency Erasure

```
rec hoas2db : (g:ctx) [g. exp T] → [. db T] =
fn e ⇒ case e of
        | [g, x:exp T . x] ⇒ [. one]
        | [g, x:exp T . #p ..] ⇒
            let [. F] = hoas2db [g. #p ..] in
            ...
```

- Dependency erasure removes type indices.
- However, it does not remove implicit arguments.

Fresh Look Representation

For contextual objects:

Contexts	$\mapsto$	Chains of links
Binders	$\mapsto$	Links
Variables	$\mapsto$	Names, imported into the corresponding world of the term

The Pipeline

Front end $\longrightarrow$ Dependency erasure $\longrightarrow$ Fresh-Style $\longrightarrow$ Pattern matching $\longrightarrow$ …

$\nearrow$ de Bruijn indices

$\searrow$ Names

Pattern Matching Compilation

It is done in two steps:

- Discriminating on the shape of contexts.
- Building a decision tree for the rest of the pattern. ([Maranget, 2008])

Pattern Matching Compilation

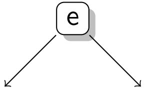
It is done in two steps:

- Discriminating on the shape of contexts.
- Building a decision tree for the rest of the pattern. ([Maranget, 2008])

ML-like Languages	Beluga	
Constructor	Constructor	`[g,x:exp. app _ _]`
Variables	Bound Var.	`[g,x:exp. x]`
	Meta Var.	`[g,x:exp. M..]`
	Parameter Var.	`[g,x:exp. #p..]`
	Context Shape	`[g,x:exp._]`

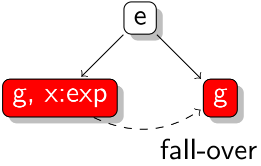
Pattern Matching Compilation

```
rec hoas2db : (g:ctx) [g. exp] → [. db] =
fn e ⇒ case e of
        | [g, x:exp . x] ⇒
        [. one]
        | [g, x:exp . #p ..] ⇒
            let [. F] = hoas2db [g. #p ..] in
            [. shift F]
        | [g. lam (λx. E .. x)] ⇒
            let [. F] = hoas2db [g,x:exp . E .. x] in
            [. lam' F]
        | [g. app (E1 .. ) (E2 .. )] ⇒
            let [. F1] = hoas2db [g. E1 ..] in
            let [. F2] = hoas2db [g. E2 ..] in
            [. app' F1 F2];
```



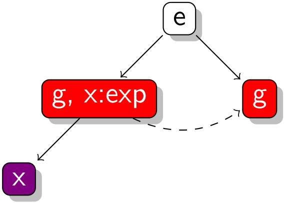
Pattern Matching Compilation

```
rec hoas2db : (g:ctx) [g. exp] → [. db] =
fn e ⇒ case e of
      | [g, x:exp . x] ⇒
      [. one]
      | [g, x:exp . #p ..] ⇒
         let [. F] = hoas2db [g. #p ..] in
         [. shift F]
      | [g. lam (λx. E .. x)] ⇒
         let [. F] = hoas2db [g,x:exp . E .. x] in
         [. lam' F]
      | [g. app (E1 .. ) (E2 .. )] ⇒
         let [. F1] = hoas2db [g. E1 ..] in
         let [. F2] = hoas2db [g. E2 ..] in
         [. app' F1 F2];
```



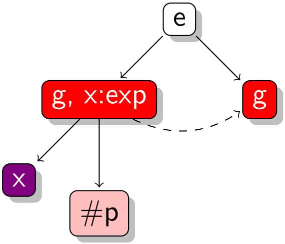
Pattern Matching Compilation

```
rec hoas2db : (g:ctx) [g. exp] → [. db] =
fn e ⇒ case e of
    | [g, x:exp . x] ⇒
    [. one]
    | [g, x:exp . #p ..] ⇒
        let [. F] = hoas2db [g. #p ..] in
        [. shift F]
    | [g. lam (λx. E .. x)] ⇒
        let [. F] = hoas2db [g,x:exp . E .. x] in
        [. lam' F]
    | [g. app (E1 .. ) (E2 .. )] ⇒
        let [. F1] = hoas2db [g. E1 ..] in
        let [. F2] = hoas2db [g. E2 ..] in
        [. app' F1 F2];
```



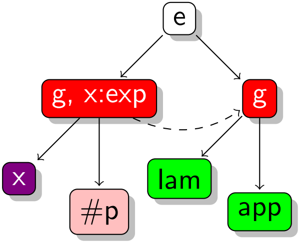
Pattern Matching Compilation

```
rec hoas2db : (g:ctx) [g. exp] → [. db] =
fn e ⇒ case e of
  | [g, x:exp . x] ⇒
    [. one]
  | [g, x:exp . #p ..] ⇒
      let [. F] = hoas2db [g. #p ..] in
      [. shift F]
  | [g. lam (λx. E .. x)] ⇒
      let [. F] = hoas2db [g,x:exp . E .. x] in
      [. lam' F]
  | [g. app (E1 .. ) (E2 .. )] ⇒
      let [. F1] = hoas2db [g. E1 ..] in
      let [. F2] = hoas2db [g. E2 ..] in
      [. app' F1 F2];
```



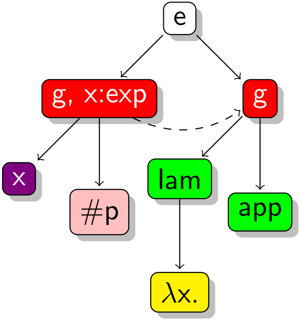
Pattern Matching Compilation

```
rec hoas2db : (g:ctx) [g. exp] → [. db] =
fn e ⇒ case e of
      | [g, x:exp . x] ⇒
      [. one]
      | [g, x:exp . #p ..] ⇒
         let [. F] = hoas2db [g. #p ..] in
         [. shift F]
      | [g. lam (λx. E .. x)] ⇒
         let [. F] = hoas2db [g,x:exp . E .. x] in
         [. lam' F]
      | [g. app (E1 .. ) (E2 .. )] ⇒
         let [. F1] = hoas2db [g. E1 ..] in
         let [. F2] = hoas2db [g. E2 ..] in
         [. app' F1 F2];
```



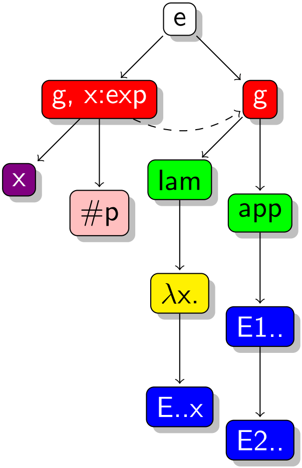
Pattern Matching Compilation

```
rec hoas2db : (g:ctx) [g. exp] → [. db] =
fn e ⇒ case e of
        | [g, x:exp . x] ⇒
        [. one]
        | [g, x:exp . #p ..] ⇒
            let [. F] = hoas2db [g. #p ..] in
            [. shift F]
        | [g. lam (λx. E .. x)] ⇒
            let [. F] = hoas2db [g,x:exp . E .. x] in
            [. lam' F]
        | [g. app (E1 .. ) (E2 .. )] ⇒
            let [. F1] = hoas2db [g. E1 ..] in
            let [. F2] = hoas2db [g. E2 ..] in
            [. app' F1 F2];
```



Pattern Matching Compilation

```
rec hoas2db : (g:ctx) [g. exp] → [. db] =
fn e ⇒ case e of
      | [g, x:exp . x] ⇒
      [. one]
      | [g, x:exp . #p ..] ⇒
         let [. F] = hoas2db [g. #p ..] in
         [. shift F]
      | [g. lam (λx. E .. x)] ⇒
         let [. F] = hoas2db [g,x:exp . E .. x] in
         [. lam' F]
      | [g. app (E1 ..) (E2..)] ⇒
         let [. F1] = hoas2db [g. E1 ..] in
         let [. F2] = hoas2db [g. E2 ..] in
         [. app' F1 F2];
```



The Two Run-times

Front end $\rightarrow$ Dependency erasure $\rightarrow$ Fresh-Style $\rightarrow$ Pattern matching $\rightarrow$ ... $\nearrow$ de Bruijn indices

$\searrow$ Names

Implementation depends on the concrete representation of binders.

- Context Shape (`[g,x:exp.._]`): a context matches a pattern when it contains enough variables to be matched against the pattern.
- Constructors(`[g,x:exp. app _ _]`): match when they are the same.
- Bound Variables (`[g,x:exp. x]`): are compared by position.
- Meta Variables (`[g,x:exp. M..]`): when the inverse substitution can be applied to the matched term.
- Parameter Variables (`[g,x:exp. #p..]`): similarly to meta variables.

Contributions/Future work

What we have:

- A framework for compiling contextual objects.
- Compiling a pattern matching that supports contextual objects. The scheme only supports first-order patterns, e.g. no bound variables or parameter vars in functional position.
- A common intermediate representation to mediate between higher-order and first-order binders.

Future work:

- Type preservation:
  - The fresh-look representation should allow us to keep the types longer
  - Establish statically, that scope is preserved throughout the compiler
- Support the whole Beluga language, i.e. Computational data-types and full pattern matching.
- Fine-grained mixed de Bruijn/named representation

Thank You!