GMeta Tutorial

Part 1: Datatype-Generic Programming

ROSAEC Center Workshop

Bruno C. d. S. Oliveira
bruno@ropas.snu.ac.kr
(joint work with Gyesik Lee, Sungkeun Cho and Kwangkeun Yi)

Motivation

“How close are we to a world where every paper on programming languages is accompanied by an electronic appendix with machine- checked proofs?”

The POPL Mark challange

Introduction

- Approaches to formal meta-theory mechanization:
  - Higher-order (almost no overhead)
  - First-order (works in Coq, easy to use & understand)
- GMeta: first-order representations without overhead using datatype-generic programming.

Main Challenge

- Binding: Dealing with binding requires a lot of basic definitions and proofs
    - *Out of a total of around 550 lemmas, approximately 400 were tedious infrastructure lemmas* (Rossberg 2010) - Formalization of ML-modules in Coq
- Problem: How to reuse prior definitions and proofs?

Infrastructure Overhead

- common operations: free & bound variables; substitutions; shifting, etc.
- lemmas about operations: permutation lemmas.
- well-formedness: lemmas that only hold on certain well-formedness conditions.

GMeta

- GMeta: a generic metatheory library for first-order representations
- Infrastructure defined once, and reused for each language.
- Parametrizable over:
  - the object calculus/language
  - the type of the first-order representation

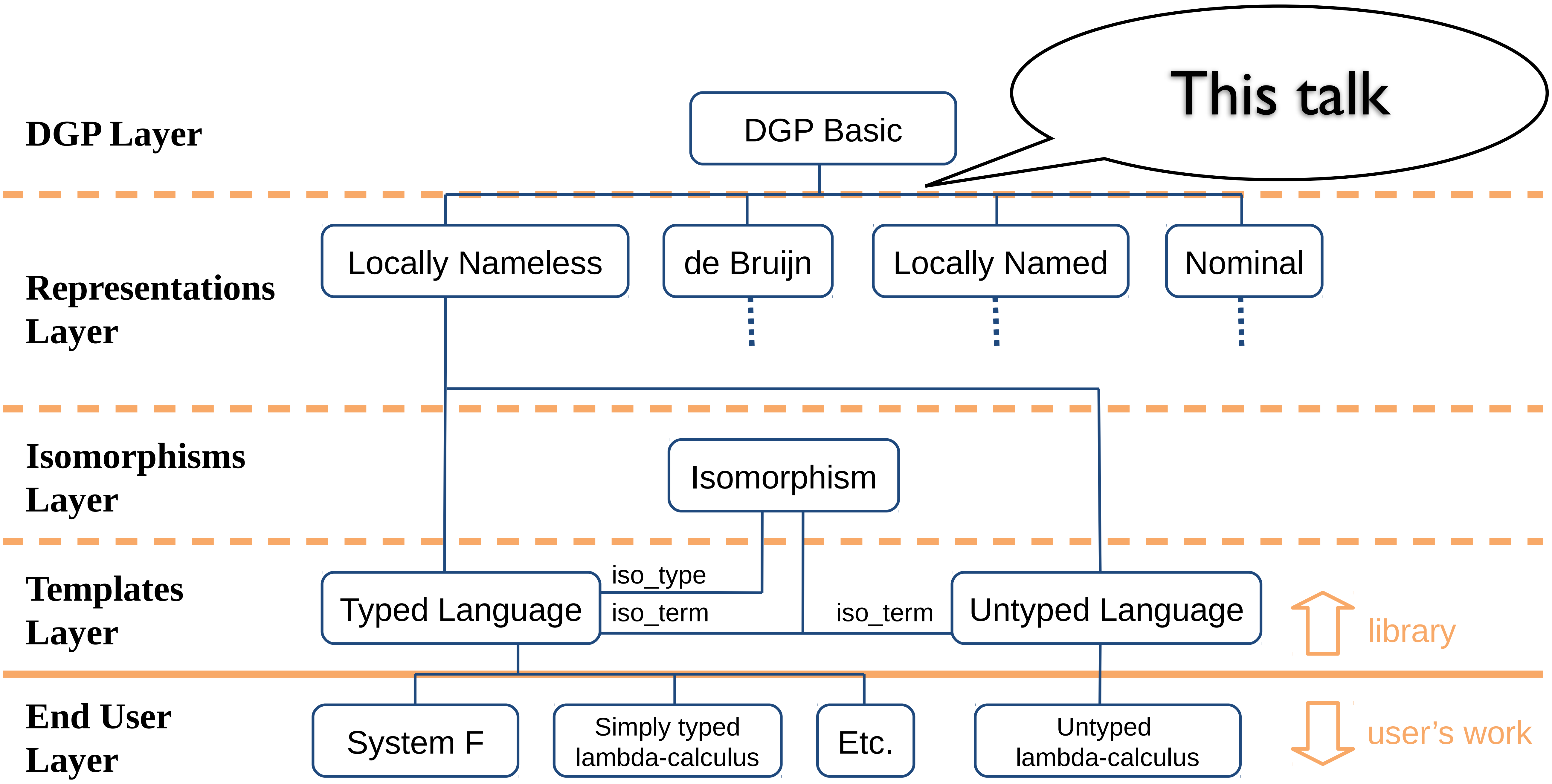
Eliminating Overhead

Used GMeta in several case studies which were compared against reference solutions by Aydemir et al. (2008).

		Savings	
		boilerplate	total
STLC	GMETA basic vs Aydemir et al.	56%	29%
	GMETA adv. vs Aydemir et al.	87.5%	45%
$F_{<:}$	GMETA basic vs Aydemir et al.	70%	43%
	GMETA adv. vs Aydemir et al.	82%	56%

Figure 3. Savings in various formalizations in terms of numbers of definitions and lemmas.

GMeta Overview



Datatype Generic Programming

The Ultimate Goal

Define binding-related operations&lemmas once and reuse them for various different object languages.

$$fv_{r_1} : \forall (r_2 : \mathsf{Rep}).\ [\![r_2]\!] \rightarrow 2^{\mathbb{N}}$$

$$[\cdot \ \rightarrow \ \cdot]\ \cdot : \forall (r_1\ \ r_2 : \mathsf{Rep}).\ \mathbb{N} \rightarrow [\![r_1]\!] \rightarrow [\![r_2]\!] \rightarrow [\![r_2]\!]$$

Examples: Free variables and substitutions functions for any language r_2.

Inductive Datatypes

We all know and love datatypes from functional languages like Haskell or ML.

$$\textsc{Data}\ \mathbb{N} = \mathsf{z} \mid \mathsf{s}\ \mathbb{N}$$

Inductive Families

Naturals using inductive families syntax:

$$\textsc{Data}\ \frac{}{\mathbb{N} : \star}\ \textsc{where} \qquad \frac{}{\mathsf{z} : \mathbb{N}} \qquad \frac{n : \mathbb{N}}{\mathsf{s}\ n : \mathbb{N}}$$

Vectors of size n:

$$\textsc{Data}\ \frac{A : \star \qquad n : \mathsf{Nat}}{\mathsf{Vector}_A\ n : \star}\ \textsc{where}$$

$$\frac{}{\mathsf{vz} : \mathsf{Vector}_A\ \mathsf{z}} \qquad \frac{n : \mathsf{Nat} \qquad a : A \qquad as : \mathsf{Vector}_A\ n}{\mathsf{vs}\ a\ as : \mathsf{Vector}_A\ (\mathsf{s}\ n)}$$

Universes

- Inductive families can capture whole families of datatypes (universes).
- Functions over inductive families work for any datatype in the family.
- Idea: Define a universe defining a family of languages with binders.

A Simple Universe

$\textsc{Data}\ \mathsf{Rep} = \mathsf{1} \mid \mathsf{Rep} + \mathsf{Rep} \mid \mathsf{Rep} \times \mathsf{Rep} \mid \mathsf{K}\ \mathsf{Rep} \mid \mathsf{R}$

$$\textsc{Data}\ \frac{r, s : \mathsf{Rep}}{[\![s]\!]_r : \star}\ \textsc{where}$$

$$\frac{}{() : [\![\mathsf{1}]\!]_r} \qquad \frac{s : \mathsf{Rep} \qquad v : [\![s]\!]}{\mathsf{k}\ v : [\![\mathsf{K}\ s]\!]_r}$$

$$\frac{s_1, s_2 : \mathsf{Rep} \qquad v : [\![s_1]\!]_r}{\mathsf{i}_1\ v : [\![s_1 + s_2]\!]_r} \qquad \frac{s_1, s_2 : \mathsf{Rep} \qquad v : [\![s_2]\!]_r}{\mathsf{i}_2\ v : [\![s_1 + s_2]\!]_r}$$

$$\frac{s_1, s_2 : \mathsf{Rep} \qquad v_1 : [\![s_1]\!]_r \qquad v_2 : [\![s_2]\!]_r}{(v_1, v_2) : [\![s_1 \times s_2]\!]_r} \qquad \frac{v : [\![r]\!]}{\mathsf{r}\ v : [\![\mathsf{R}]\!]_r}$$

$$\textsc{Data}\ \frac{s : \mathsf{Rep}}{[\![s]\!] : \star}\ \textsc{where} \qquad \frac{s : \mathsf{Rep} \qquad v : [\![s]\!]_s}{\mathsf{in}\ v : [\![s]\!]}$$

A Simple Universe

Modeling datatypes with the universe:

$\mathsf{RNat} : \mathsf{Rep}$
$\mathsf{RNat} = 1 + \mathsf{R}$

$\mathsf{RList} : \mathsf{Rep}$
$\mathsf{RList} = 1 + \mathsf{K}\ \mathsf{RNat} \times \mathsf{R}$

$\mathsf{nil} : [\![\mathsf{RList}]\!]$
$\mathsf{nil} = \mathsf{in}\ (\mathsf{i}_1\ ())$

$\mathsf{cons} : [\![\mathsf{RNat}]\!] \rightarrow [\![\mathsf{RList}]\!] \rightarrow [\![\mathsf{RList}]\!]$
$\mathsf{cons}\ n\ ns = \mathsf{in}\ (\mathsf{i}_2\ (\mathsf{k}\ n, \mathsf{r}\ ns))$

Traditional recursive types:

$\mathsf{Nat} = \mu\ R.\ 1 + R$
$\mathsf{List} = \mu\ R.\ 1 + \mathsf{Nat} \times R$

Generic functions

Generic size:

$$size : \forall (r : \mathsf{Rep}).\ [\![r]\!] \rightarrow \mathbb{N}$$
$$size\ (\mathsf{in}\ t) = size\ t$$

$$size : \forall (r, s : \mathsf{Rep}).\ [\![s]\!]_r \rightarrow \mathbb{N}$$
$$\begin{aligned} size\ () &= 0 \\ size\ (\mathsf{k}\ t) &= 0 \\ size\ (\mathsf{i}_1\ t) &= size\ t \\ size\ (\mathsf{i}_2\ t) &= size\ t \\ size\ (t, v) &= size\ t + size\ v \\ size\ (\mathsf{r}\ \ t) &= 1 + size\ t \end{aligned}$$

If r = RNat then *size* is value of the natural number.

If r = RList then *size* is the length of the list.

More generally, *size* works for any r.

Representing Binders

Extended universe:

$$\textsc{Data}\ \mathsf{Rep} = \ldots\ |\ \mathsf{E}\ \mathsf{Rep}\ |\ \mathsf{B}\ \mathsf{Rep}\ \mathsf{Rep}$$

$$\begin{array}{ll} Q : \star & \text{(* Quantifier type *)} \\ V : \star & \text{(* Variable type *)} \end{array}$$

$$\textsc{Data}\ \frac{r, s : \mathsf{Rep}}{[\![s]\!]_r : \star}\ \textsc{where}\ \ \ldots$$

Binders

$$\frac{s : \mathsf{Rep} \qquad v : [\![s]\!]}{\mathsf{e}_s\ v : [\![\mathsf{E}\ s]\!]_r} \qquad \frac{s_1, s_2 : \mathsf{Rep} \qquad q : Q \qquad v : [\![s_2]\!]_r}{\lambda_{s_1} q.v : [\![\mathsf{B}\ s_1\ s_2]\!]_r}$$

$$\textsc{Data}\ \frac{r : \mathsf{Rep}}{[\![r]\!] : \star}\ \textsc{where}\ \ \ldots \qquad \frac{s : \mathsf{Rep} \qquad v : V}{\mathsf{var}\ v : [\![s]\!]}$$

Variables

Lambda Calculus

Representing the lambda calculus:

$\mathsf{RLambda} : \mathsf{Rep}$
$\mathsf{RLambda} = \mathsf{R} \times \mathsf{R} + \mathsf{B}\ \mathsf{R}\ \mathsf{R}$

$\mathsf{fvar} : \mathbb{N} \rightarrow [\![\mathsf{RLambda}]\!]$
$\mathsf{fvar}\ n \quad = \mathsf{var}\ (\mathsf{inl}\ n)$

$\mathsf{bvar} : \mathbb{N} \rightarrow [\![\mathsf{RLambda}]\!]$
$\mathsf{bvar}\ n \quad = \mathsf{var}\ (\mathsf{inr}\ n)$

$\mathsf{app} : [\![\mathsf{RLambda}]\!] \rightarrow [\![\mathsf{RLambda}]\!] \rightarrow [\![\mathsf{RLambda}]\!]$
$\mathsf{app}\ e_1\ e_2 = \mathsf{in}\ (\mathsf{i}_1\ (\mathsf{r}\ \ e_1, \mathsf{r}\ \ e_2))$

$\mathsf{lam} : [\![\mathsf{RLambda}]\!] \rightarrow [\![\mathsf{RLambda}]\!]$
$\mathsf{lam}\ e \quad = \mathsf{in}\ (\mathsf{i}_2\ (\lambda_{\mathsf{R}} \mathbb{1}.\ \mathsf{r}\ \ e))$

Generic Functions

Free variables (locally nameless):

> Instantiation of Q and V:
>
> $Q = \mathbb{1}$
> $V = \mathbb{N} + \mathbb{N}$

$$
\begin{array}{lll}
fv_{r_1} & : \forall (r_2 : \mathsf{Rep}).\ [\![r_2]\!] \to 2^{\mathbb{N}} & \\
fv_{r_1} & (\mathsf{in}\ t) & = fv_{r_1}\ t \\
fv_{r_1} & (\mathsf{var}\ (\mathsf{inl}\ x)) & = \mathbf{if}\ r_1 \equiv r_2\ \mathbf{then}\ \{x\}\ \mathbf{else}\ \emptyset \\
fv_{r_1} & (\mathsf{var}\ (\mathsf{inr}\ y)) & = \emptyset
\end{array}
$$

$$
\begin{array}{lll}
fv_{r_1} & : \forall (r_2, s : \mathsf{Rep}).\ [\![s]\!]_{r_2} \to 2^{\mathbb{N}} & \\
fv_{r_1} & () & = \emptyset \\
fv_{r_1} & (\mathsf{k}\ t) & = \emptyset \\
fv_{r_1} & (\mathsf{e}\ t) & = fv_{r_1}\ t \\
fv_{r_1} & (\mathsf{i}_1\ t) & = fv_{r_1}\ t \\
fv_{r_1} & (\mathsf{i}_2\ t) & = fv_{r_1}\ t \\
fv_{r_1} & (t, v) & = (fv_{r_1}\ t) \cup (fv_{r_1}\ v) \\
fv_{r_1} & (\lambda_{r_3} \mathbb{1}.t) & = fv_{r_1}\ t \\
fv_{r_1} & (\mathsf{r}\ \ t) & = fv_{r_1}\ t
\end{array}
$$

Generic Functions

Substitution for free variables:

$$[\cdot \;\rightarrow\; \cdot]\;\cdot : \forall(r_1\;\, r_2 : \mathsf{Rep}).\, \mathbb{N} \rightarrow [\![r_1]\!] \rightarrow [\![r_2]\!] \rightarrow [\![r_2]\!]$$

$$\begin{aligned}
&[k \;\rightarrow\; u]\;(\mathsf{in}\ t) &&= \mathsf{in}\ ([k \;\rightarrow\; u]\ t)\\
&[k \;\rightarrow\; u]\;(\mathsf{var}\ (\mathsf{inl}\ x)) = \\
&\quad \mathbf{if}\ r_1 \equiv r_2 \wedge k \equiv x\ \mathbf{then}\ u\ \mathbf{else}\ (\mathsf{var}\ (\mathsf{inl}\ x))\\
&[k \;\rightarrow\; u]\;(\mathsf{var}\ (\mathsf{inr}\ y)) &&= \mathsf{var}\ (\mathsf{inr}\ y)
\end{aligned}$$

$$[\cdot \;\rightarrow\; \cdot]\;\cdot : \forall(r_1, r_2, s : \mathsf{Rep}).\, \mathbb{N} \rightarrow [\![r_1]\!] \rightarrow [\![s]\!]_{r_2} \rightarrow [\![s]\!]_{r_2}$$

$$\begin{aligned}
&[k \;\rightarrow\; u]\;() &&= ()\\
&[k \;\rightarrow\; u]\;(\mathsf{k}\ t) &&= \mathsf{k}\ t\\
&[k \;\rightarrow\; u]\;(\mathsf{e}\ t) &&= \mathsf{e}\ ([k \;\rightarrow\; u]\ t)\\
&[k \;\rightarrow\; u]\;(\mathsf{i}_1\ t) &&= \mathsf{i}_1\ ([k \;\rightarrow\; u]\ t)\\
&[k \;\rightarrow\; u]\;(\mathsf{i}_2\ t) &&= \mathsf{i}_2\ ([k \;\rightarrow\; u]\ t)\\
&[k \;\rightarrow\; u]\;(t, v) &&= ([k \;\rightarrow\; u]\ t, [k \;\rightarrow\; u]\ v)\\
&[k \;\rightarrow\; u]\;(\lambda_{r_3}\mathbb{1}.z) &&= \lambda_{r_3}\mathbb{1}.([k \;\rightarrow\; u]\ z)\\
&[k \;\rightarrow\; u]\;(\mathsf{r}\ \ t) &&= \mathsf{r}\ \ ([k \;\rightarrow\; u]\ t)
\end{aligned}$$

Generic Lemmas

It is possible to do generic lemmas too:

$$
\begin{aligned}
& subst\_fresh : \forall (r_1, r_2 : \mathsf{Rep})\ (t : [\![r_1]\!])\ (u : [\![r_2]\!])\ (m : \mathbb{N}). \\
& \quad m \notin (fv_{r_2}\ t) \ \ \Rightarrow \ \ [m \to u]\ t = t \\
& bfsubst\_perm : \forall (r_1, r_2, r_3 : \mathsf{Rep})\ (t : [\![r_1]\!])\ (u : [\![r_2]\!])\ (v : [\![r_3]\!]) \\
& \quad (m\ k : \mathbb{N}).\ (wf_{r_3}\ u) \ \ \Rightarrow \\
& \quad \{k \to ([m \to u]\ v)\}\ ([m \to u]\ t) = [m \to u]\ (\{k \to v\}\ t)
\end{aligned}
$$

Conclusion

- Boring lemmas and definitions can be dealt with generically.

- Gyesik will show how to use this for practical mechanizations of metatheory in Coq.

Related Work

- Several clean settings that deal with binding:
  - Parametric HOAS (POPL 2010)
  - A Universe of Binding and Computation (ICFP 2009)
  - Nominal Datatypes (Pitts 2003)
- But most practical development is done in Coq with traditional first-order approaches:
  - This is where our approach fits in