GMeta: A Generic Formal Metatheory Framework for First-Order Representations*

Gyesik Lee[1], Bruno C.D.S. Oliveira[2], Sungkeun Cho[2], and Kwangkeun Yi[2]

[1] Hankyong National University, Korea
gslee@hknu.ac.kr
[2] ROSAEC Center, Seoul National University, Korea
{bruno,skcho,kwang}@ropas.snu.ac.kr

Abstract. This paper presents GMeta: a generic framework for *first-order representations* of variable binding that provides *once and for all* many of the so-called infrastructure lemmas and definitions required in mechanizations of formal metatheory. The key idea is to employ *datatype-generic programming* (DGP) and *modular programming* techniques to deal with the infrastructure overhead. Using a generic *universe* for representing a large family of object languages we define datatype-generic libraries of infrastructure for first-order representations such as *locally nameless* or *de Bruijn* indices. Modules are used to provide *templates*: a convenient interface between the datatype-generic libraries and the end-users of GMeta. We conducted case studies based on the POPLmark challenge, and showed that dealing with challenging binding constructs, like the ones found in System $F_{<:}$, is possible with GMeta. All of GMeta's generic infrastructure is implemented in the Coq theorem prover. Furthermore, due to GMeta's modular design, the libraries can be easily used, extended, and customized by users.

Keywords: Mechanization, variable binding, first-order representations, POPLmark challenge, datatype-generic programming, Coq.

1 Introduction

A key issue in mechanical developments of formal metatheory for programming languages concerns the representation and manipulation of terms with variable binding. There are two main approaches to address this issue: *first-order* and *higher-order* approaches. In first-order approaches variables are typically encoded using names or natural numbers, whereas higher-order approaches such as higher-order abstract syntax (HOAS) use the function space in the meta-language to encode binding of the object language.

Higher-order approaches are appealing because issues like capture-avoidance and alpha-equivalence can be handled once and for all. This is why such approaches are used in logical frameworks such as Hybrid (Momigliano et al. 2008),

* This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST)/National Research Foundation of Korea(NRF) (Grants R11-2008-007-01002-0 and 2010-0001717).

© Springer-Verlag Berlin Heidelberg 2012

		term	type
term	Variables	$bsubst_{\mathsf{term}\times\mathsf{term}}$	$bsubst_{\mathsf{term}\times\mathsf{type}}$
	Parameters	$fsubst_{\mathsf{term}\times\mathsf{term}}$	$fsubst_{\mathsf{term}\times\mathsf{type}}$
type	Variables	$bsubst_{\mathsf{type}\times\mathsf{term}}$	$bsubst_{\mathsf{type}\times\mathsf{type}}$
	Parameters	$fsubst_{\mathsf{type}\times\mathsf{term}}$	$fsubst_{\mathsf{type}\times\mathsf{type}}$

Fig. 1. Possible variations of substitutions for parameters and variables for a language with two syntactic sorts (term and type) in the locally nameless style

Abella (Gacek 2008), or Twelf (Pfenning and Schürmann 1999); and have also been advocated (Despeyroux et al. 1995; Chlipala 2008) in general-purpose theorem provers like Coq (Coq Development Team 2009).

The main advantage of first-order approaches, and the reason why they are so popular in theorem provers like Coq, is that they are close to pen-and-paper developments and they do not require special support from the theorem prover.

However, the main drawback of first-order approaches is that the tedious infrastructure required for handling variable binding has to be repeated each time for a new object language. For each binding construct in the language, there is a set of *infrastructure operations* and associated *lemmas* that should be implemented. In the locally nameless style (Aydemir et al. 2008) and locally named (McKinna and Pollack 1993) styles we usually need operations like substitution for parameters (free variables) and for (bound) variables as well some associated lemmas. For de Bruijn indices (de Bruijn 1972) we need similar infrastructure, but for operations such as substitution and shifting instead.

Often, the majority of the total number of lemmas and definitions in a formalization consists of basic infrastructure. Figure 1 illustrates the issue using a simple language with two syntactic sorts (types and terms) supporting binding constructs for both type and term variables and assuming a locally nameless style. In the worst case scenario, 8 different types of substitution are needed. We need substitutions for parameters and variables, and for each of these we need to consider all four combinations of substitutions using types and terms. While not all operations are necessary in formalizations, many of them are. For example, System $F_{<:}$, which is the language described in the POPLMark challenge (Aydemir et al. 2005), requires 6 out of the 8 substitutions. Because for each operation we need to also prove a number of associated lemmas, solutions to the POPLMark challenge typically have a large percentage of lemmas and definitions just for infrastructure. In the solution by Aydemir et al. (2008), infrastructure amounts to 65% of the total number of definitions and lemmas (see also Figure 10). In realistic formalizations the situation is often not better: Rossberg et al. (2010) report a combinatorial explosion of infrastructure lemmas and operations as the number of syntactic sorts and binding constructs increases.

Importantly, considering only *homogeneous* operations (like $bsubst_{\mathsf{term}\times\mathsf{term}}$), which perform substitutions of variables on terms of the same sort (term), is insufficient. Generally we must also consider *heterogeneous* operations, like $bsubst_{\mathsf{type}\times\mathsf{term}}$, where the sort of variables being substituted (type) is not of

```
(*@Iso type_iso {
  Parameter type_fvar,
  Variable  type_bvar,
  Binder    type_all _
}*)
Inductive type :=
 | type_fvar   : ℕ → type
 | type_bvar   : ℕ → type
 | type_top    : type
 | type_arrow  : type → type → type
 | type_all    : type → type → type.
```

```
(*@Iso term_iso {
  Parameter term_fvar,
  Variable  term_bvar,
  Binder    term_abs _,
  Binder    term_tabs _ binds type
}*)
Inductive term :=
 | term_fvar : ℕ → term
 | term_bvar : ℕ → term
 | term_app  : term → term → term
 | term_abs  : type → term → term
 | term_tapp : term → type → term
 | term_tabs : type → term → term.
```

Fig. 2. Syntax definitions and GMeta isomorphism annotations for a locally nameless style version of System $F_{<:}$ in Coq

the same as the terms which are being substituted into (term). Languages like System F have type abstractions in terms ($\Lambda X.e$) and require operations like $bsubst_{\mathsf{type}\times\mathsf{term}}$ and $fsubst_{\mathsf{type}\times\mathsf{term}}$ for substituting type variables in terms.

1.1 Our Solution

To deal with the combinatorial explosion of infrastructure operations and lemmas, we propose the use of *datatype-generic programming* (DGP) and *modular programming* techniques. The key idea is that, with DGP, we can define once and for all the tedious infrastructure lemmas and operations in a generic way and, with modules, we can provide a convenient interface for users to instantiate such generic infrastructure to their object languages.

This idea is realized in GMeta: a generic framework for first-order representations of variable binding implemented in Coq[1]. In GMeta, a DGP *universe* (Martin-Löf 1984) is used to represent a large family of object languages and includes constructs for representing the binding structure of those languages. The universe is independent of the particular choice of first-order representations: it can be instantiated, for example, to *locally nameless* or *de Bruijn* representations. GMeta uses that universe to provide libraries with the infrastructure for various first-order representations.

The infrastructure is reused by users through so-called *templates*. Templates are functors parameterized by isomorphisms between the object language and the corresponding representation of that language in the universe. By instantiating templates with isomorphisms, users get access to a module that provides infrastructure tailored for a particular binding construct in their own object language. For example, for System $F_{<:}$, the required infrastructure is provided by 3 modules which instantiate GMeta's locally nameless template:

[1] We also have an experimental Agda implementation.

Module $M_{\mathsf{term}\times\mathsf{term}} := \mathit{LNTemplate\ term\_iso\ term\_iso}$.
Module $M_{\mathsf{type}\times\mathsf{type}} := \mathit{LNTemplate\ type\_iso\ type\_iso}$.
Module $M_{\mathsf{type}\times\mathsf{term}} := \mathit{LNTemplate\ type\_iso\ term\_iso}$.

Each module corresponds to one of the 3 combinations needed in System $F_{<:}$, and contains the relevant lemmas and operations. By using this scheme we can deal with the general case of object languages with N syntactic sorts, just by expressing the combinations needed in that language. Moreover GMeta can also provide some more specialized templates for additional reuse and it is easy for users to define their own types of infrastructure and customized templates.

Since isomorphisms can be mechanically generated from the inductive definition of the object language, provided a few annotations, GMeta also includes optional tool support for generating such isomorphisms automatically. Figure 2 illustrates these annotations for System $F_{<:}$. Essentially, the keyword Iso introduces an isomorphism annotation, while the keywords Parameter, Variable and Binder provide the generator with information about which constructors correspond, respectively, to the parameters, variables or binders. Therefore, at the cost of just a few annotations or explicitly creating an isomorphism by hand, GMeta provides much of the tedious infrastructure boilerplate that would constitute a large part of the whole development otherwise.

1.2 Contributions

Our main contribution is to investigate how DGP techniques can deal with the infrastructure overhead required by formalizations using first-order representations. More concretely, the contributions of this paper are:

– *Sound, generic, reusable and extensible infrastructure for first-order representations*: The main advantages of using DGP are that it allows a library-based approach in which 1) the infrastructure can be defined and verified once and for all *within* the meta-logic itself; and 2) extending the infrastructure is easy since it just amounts to extending the library.
– *Heterogeneous generic operations and lemmas*: Of particular interest is the ability of GMeta to deal with binding constructs involving multiple syntactic sorts, such as binders found in the System F family of languages, using heterogeneous generic operations and lemmas.
– *Case studies using the POPLmark challenge*: To validate our approach in practice, we conducted case studies using the POPLmark challenge. Compared to other solutions, our approach shows significant savings in the number of definitions and lemmas required by formalizations.
– *Coq implementation and other resources*: The GMeta framework Coq implementation is available online[2] along with other resources such as tutorials and more case studies.

[2] `http://ropas.snu.ac.kr/gmeta/`
The implementation is based on Coq Version 8.2pl2.

		style	savings
STLC	GMeta vs Aydemir et al.	LN	52%
$F_{<:}$	GMeta vs Aydemir et al.	LN	38%
	GMeta vs Vouillon	dB	35%

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

2 Case Studies

In order to verify the effectiveness of GMeta in reducing the infrastructure overhead, we conducted case studies using locally nameless and de Bruijn representations. Since the results in terms of savings were similar, and due to space limitations, we mainly discuss the locally nameless case studies in this paper. The details of the de Bruijn case studies can be found on GMeta's online webpage. Our two case studies are a solution to the POPLmark challenge parts 1A+2A, and a formalization of the STLC.

GMeta can reduce the infrastructure overhead because it provides reuse of boilerplate definitions and lemmas. By boilerplate we mean the following:

- *Common operations:* operations such as sets of parameters and (bound) variables, term size or different forms of substitution-like operations (such as substitutions for parameters and variables in the locally nameless style; or shifting in the de Bruijn style).
- *Lemmas about common operations:* lemmas about properties of the common operations, such as several forms of permutation lemmas about substitutions.
- *Lemmas involving well-formedness:* many lemmas about common operations only hold when a term is well-formed under a certain environment. Since well-formedness is a notion that appears in many systems and it is often mechanical, we consider such lemmas boilerplate.

The biggest benefit of GMeta is that it significantly lowers the overheads required in mechanical formalizations by providing reuse of the basic infrastructure. Figure 3 shows the savings that GMeta achieved relative to the reference solutions by Aydemir et al. (2008) and Vouillon (2007). Note that in GMeta only user-defined code is counted. In all case studies more than 35% of the total numbers of definitions were saved. We conducted case studies in both System $F_{<:}$ and STLC. A more detailed discussion and evaluation is given in Section 6.

3 GMeta Design

This section gives a general overview of GMeta's design and discusses the techniques used by us to make GMeta convenient to use.

As depicted in Figure 4, the GMeta framework is structured into 5 layers of modules. The structure is hierarchical, with the more general modules at the top and the more specific modules at the bottom.

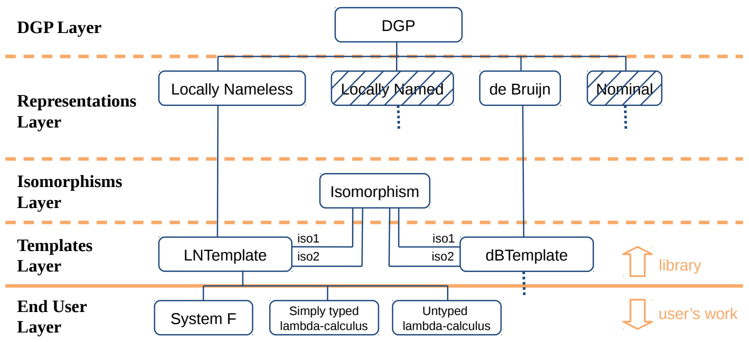

Fig. 4. A simplified modular structure overview of GMETA

- DGP Layer: The core DGP infrastructure is defined at the top-most layer. The main component is a universe that acts as a generic language that the lower-level modules use to define the infrastructure lemmas and definitions.
- Representation Layer: This layer is where the generic infrastructure lemmas and definitions for particular first-order representations are defined. GMETA currently supports locally nameless and de Bruijn representations. However, the DGP library can be extended to cover locally-named approaches (McKinna and Pollack 1993; Sato and Pollack 2010) and other representations.
- Isomorphism Layer: This layer provides simple module signatures for isomorphisms that serve as interfaces between the object language and its representation in the generic language. The adequacy of the object language representation follows from the isomorphism laws.
- Templates Layer: This layer provides templates for the basic infrastructure lemmas and definitions required by particular meta-theoretical developments. Templates are ML-style functors parameterized by isomorphisms between the syntactic sorts of object languages and their corresponding representations in the generic language. In Figure 4 we show only *LNTemplate* and *dBTemplate*, which are the fundamental templates providing reuse for the general infrastructure.
- End User Layer: End users will use GMETA's libraries to develop metatheory for particular object languages, for example, the simply typed lambda calculus (STLC) or System $F_{<:}$ used in our case studies.

The two top layers will be discussed in detail in Sections 4 and 5. They are the most interesting from a technical point of view. More information about other layers and a tutorial are available in GMETA's webpage.

3.1 Making GMETA Convenient to Use

To provide convenience to the user, GMETA employs several techniques. Although DGP plays a fundamental role in the definition of the core libraries of GMETA (at

$$\mathit{fsubst}_{\mathsf{term}\times\mathsf{term}} : \mathbb{N} \to \mathsf{term} \to \mathsf{term} \to \mathsf{term}$$
$$\mathit{fsubst}_{\mathsf{term}\times\mathsf{term}}\ k\ u\ t = \mathit{to}_{\mathsf{term}}\ ([k\ \to\ (\mathit{from}_{\mathsf{term}}\ u)]\ (\mathit{from}_{\mathsf{term}}\ t))$$
$$\mathit{bsubst}_{\mathsf{term}\times\mathsf{term}} : \mathbb{N} \to \mathsf{term} \to \mathsf{term} \to \mathsf{term}$$
$$\mathit{bsubst}_{\mathsf{term}\times\mathsf{term}}\ k\ u\ t = \mathit{to}_{\mathsf{term}}\ (\{k\ \to\ (\mathit{from}_{\mathsf{term}}\ u)\}\ (\mathit{from}_{\mathsf{term}}\ t))$$
$$\mathit{fsubst}_{\mathsf{term}\times\mathsf{type}} : \mathbb{N} \to \mathsf{type} \to \mathsf{term} \to \mathsf{term}$$
$$\mathit{fsubst}_{\mathsf{term}\times\mathsf{type}}\ k\ u\ t = \mathit{to}_{\mathsf{term}}\ ([k\ \to\ (\mathit{from}_{\mathsf{type}}\ u)]\ (\mathit{from}_{\mathsf{term}}\ t))$$
$$\mathit{bsubst}_{\mathsf{term}\times\mathsf{type}} : \mathbb{N} \to \mathsf{type} \to \mathsf{term} \to \mathsf{term}$$
$$\mathit{bsubst}_{\mathsf{term}\times\mathsf{type}}\ k\ u\ t = \mathit{to}_{\mathsf{term}}\ (\{k\ \to\ (\mathit{from}_{\mathsf{type}}\ u)\}\ (\mathit{from}_{\mathsf{term}}\ t))$$
$$\mathit{bsubst}_{\mathsf{type}\times\mathsf{type}} : \mathbb{N} \to \mathsf{type} \to \mathsf{type} \to \mathsf{type}$$
$$\mathit{bsubst}_{\mathsf{type}\times\mathsf{type}}\ k\ u\ t = \mathit{to}_{\mathsf{type}}\ ([k\ \to\ (\mathit{from}_{\mathsf{type}}\ u)]\ (\mathit{from}_{\mathsf{type}}\ t))$$
$$\mathit{wf}_{\mathsf{term}} : \mathsf{term} \to \mathsf{Prop}$$
$$\mathit{wf}_{\mathsf{type}} : \mathsf{type} \to \mathsf{Prop}$$
$$\begin{aligned}&\mathit{thbfsubst\_perm\_core} : \forall (t : \mathsf{term})\ (u, v : \mathsf{type})\ (m\ k : \mathbb{N}),\\ &\quad \mathit{wf}_{\mathsf{type}}\ u \Rightarrow \mathit{bsubst}_{\mathsf{term}\times\mathsf{type}}\ k\ (\mathit{bsubst}_{\mathsf{type}\times\mathsf{type}}\ m\ u\ v)\ (\mathit{fsubst}_{\mathsf{term}\times\mathsf{type}}\ m\ u\ t)\\ &\qquad\qquad = \mathit{fsubst}_{\mathsf{term}\times\mathsf{type}}\ m\ u\ (\mathit{bsubst}_{\mathsf{term}\times\mathsf{type}}\ k\ v\ t)\end{aligned}$$

Fig. 5. Some representations of a template with two sorts: *terms* and *types*

the DGP and representation layers), end users should not need knowledge about DGP for uses of GMeta. However, this is not trivial to achieve because, among other things, end-user proofs generally require unfolding infrastructure operations like substitution, and those operations are written in a datatype-generic way, in a form which is alien to users that do not know about DGP.

Automatically Generated Isomorphisms. GMeta uses automatically generated isomorphisms between the user-defined object language and a corresponding representation of that language of the generic universe. Since information about the binding structure of the language is required to generate isomorphisms, GMeta uses a small annotation language. (see Figure 2 for an example of the annotation language).

Templates. GMeta uses templates to solve the problem of interfacing with the infrastructure DGP libraries.

As already illustrated in Figure 1, a simple language with two syntactic sorts (terms and types) needs two isomorphisms ($\mathit{from}_{\mathsf{term}}, \mathit{to}_{\mathsf{term}}$) and ($\mathit{from}_{\mathsf{type}}, \mathit{to}_{\mathsf{type}}$) between the generic language and the object language. What we mean by isomorphism is explained in the next paragraph about special tactics. In Figure 5, it is demonstrated how the two isomorphisms are used to get an instantiation where several variants of substitution, and well-formedness in the locally nameless style become available for free. The templates include also many lemmas about the operations and some of the lemmas may be true only for well-formedness expressions. For example, the lemma *thbfsubst_perm_core* describes a permutability of two kinds of substitutions where well-formed types ($\mathit{wf}_{\mathsf{type}}$) are involved.

The general form of parameter substitution in the locally nameless template is as follows:

Module $\mathit{LNTemplate}$ $(\mathit{iso}_{\mathsf{S}_1} : \mathsf{Iso}, \mathit{iso}_{\mathsf{S}_2} : \mathsf{Iso})$.

...

$\mathit{fsubst}_{\mathsf{S}_2 \times \mathsf{S}_1} \quad : \mathbb{N} \rightarrow \mathsf{S}_1 \rightarrow \mathsf{S}_2 \rightarrow \mathsf{S}_2$
$\mathit{fsubst}_{\mathsf{S}_2 \times \mathsf{S}_1}\ k\ u\ t = \mathit{to}_{\mathsf{S}_2}\ (\{k\ \rightarrow\ (\mathit{from}_{\mathsf{S}_1}\ u)\}\ (\mathit{from}_{\mathsf{S}_2}\ t))$

Essentially, S_1 and S_2 are supposed to be the types of the syntactic sorts used in object language. These types come from the isomorphisms $\mathit{iso}_{\mathsf{S}_1}$ and $\mathit{iso}_{\mathsf{S}_2}$, which are the parameters of *LNTemplate*. Definitions like $\{\cdot\ \rightarrow\ \cdot\}_T\ \cdot$ are simply using the isomorphism (through the operations $\mathit{to}_{\mathsf{S}_1}$, $\mathit{to}_{\mathsf{S}_2}$, $\mathit{from}_{\mathsf{S}_1}$ and $\mathit{from}_{\mathsf{S}_2}$) to interface with generic operations like $\{\cdot\ \rightarrow\ \cdot\}\ \cdot$ (see Figure 9) defined in the representations layer.

Because of the isomorphisms between the user's object language and the representation of that language in the universe, users do not need to interact directly with the generic universe. Instead, all that a user needs to do is to instantiate the templates with the automatically generated isomorphisms. In Section 1.1, we already described how this technique is used to generate the infrastructure for System $F_{<:}$.

Special Tactics. When proving lemmas for their own formalizations, users may need to unfold operations which are defined in terms of corresponding generic operations. For example, the following lemma is a core lemma in formalization of in the solution to the POPLMark challenge by Aydemir et al. (2008).

Lemma $\mathit{typing\_subst} : \forall E\ F\ U\ t\ T\ z\ u,$
  $(E \mathbin{{+}{+}} (z, U) :: F) \vdash t : T \Rightarrow F \vdash u : U \Rightarrow$
  $(E \mathbin{{+}{+}} F) \vdash ([z\ \rightarrow\ u]_T\ t) : T.$
Proof.
  intros; *dependent induction H*; $\underline{\mathit{gsimpl}}$.
  ...
  $\underline{\mathit{grewrite}}$ *tbfsubst_permutation_var_wf*; *eauto*.
  ...
Qed.

The details of the Coq proof are not relevant. What is important to note is: 1) the key difference to the original proof by Aydemir et al. (2008) is that two different tactics ($\underline{\mathit{gsimpl}}$ and $\underline{\mathit{grewrite}}$) are used; and 2) the lemma *tbfsubst_permutation_var_wf* and the operation $[\cdot\ \rightarrow\ \cdot]_T\ \cdot$ are provided by GMeta's templates.

If the user would try to use *simpl* (the standard Coq tactic to unfold and simplify definitions) directly, the definition of $[\cdot\ \rightarrow\ \cdot]_T\ \cdot$ would be unfolded and he would be presented with parts of the definition of $[\cdot\ \rightarrow\ \cdot]\ \cdot$ (See Figure 9). However, this is clearly undesirable since the expected definition at this point is one similar to a manually defined operation for the object language in hand.

Our solution to this problem is to define some Coq tactics (such as $\underline{\mathit{gsimpl}}$ and $\underline{\mathit{grewrite}}$) that *specialize* operations and lemmas such as $[\cdot\ \rightarrow\ \cdot]_T\ \cdot$ and *tbfsubst_permutation_var_wf* using the isomorphisms provided by the user, and the isomorphism and adequacy laws shown in Figure 6.

$$\begin{aligned} to_{\mathsf{S}_2}\ (from_{\mathsf{S}_2}\ t) &= t \\ from_{\mathsf{S}_2}\ (to_{\mathsf{S}_2}\ t) &= t \\ from_{\mathsf{S}_2}\ ([k \to u]_T\, t) &= [k \to (from_{\mathsf{S}_2}\ u)]\, (from_{\mathsf{S}_2}\ t) \end{aligned}$$

Fig. 6. Isomorphism and adequacy laws

4 DGP for Datatypes with First-Order Binders

This section briefly introduces DGP using inductive families to define universes of datatypes, and shows how to adapt a conventional universe of datatypes to support binders and variables. In our presentation we assume a type theory extended with inductive families, such as the Calculus of Inductive Constructions (CIC) (Paulin-Mohring 1996) or extensions of Martin-Löf type-theory (Martin-Löf 1984) with inductive families (Dybjer 1997).

4.1 Inductive Families

Inductive families are a generalization of conventional datatypes that has been introduced in dependently typed languages such as Epigram (McBride and McKinna 2004), Agda (Norell 2007) or the Coq theorem prover. They are also one of the inspirations for Generalized Algebraic Datatypes (GADTs) (Peyton Jones et al. 2006) in Haskell.

We adopt a notation similar to the one used by Epigram to describe inductive families. For example we can define a family of vectors of size n as follows:

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

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

In this definition the type constructor for vectors has two type arguments. The first argument specifies the type A of elements of the vector, while the second argument n is the size of the vector. We write parametric type arguments in type constructors such as Vector_A using a subscript. Also, if a constructor is not explicitly applied to some arguments (for example $\mathsf{vs}\ a\ as$ is not applied to n), then those arguments are implicitly passed.

4.2 Datatype Generic Programming

The key idea behind DGP is that many functions can be defined generically for whole families of datatype definitions. Inductive families are useful to DGP because they allow us to define universes (Martin-Löf 1984) representing whole families of datatypes. By defining functions over this universe we obtain generic functions that work for any datatypes representable in that universe.

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

$$\textsc{Data}\ \frac{r, s : \mathsf{Rep}}{\llbracket s \rrbracket_r : \star}\ \textsc{where} \quad \frac{}{() : \llbracket 1 \rrbracket_r} \qquad \frac{s : \mathsf{Rep} \qquad v : \llbracket s \rrbracket}{\mathsf{k}\ v : \llbracket \mathsf{K}\ s \rrbracket_r}$$

$$\frac{s_1, s_2 : \mathsf{Rep} \qquad v : \llbracket s_1 \rrbracket_r}{\mathsf{i}_1\ v : \llbracket s_1 + s_2 \rrbracket_r} \qquad \frac{s_1, s_2 : \mathsf{Rep} \qquad v : \llbracket s_2 \rrbracket_r}{\mathsf{i}_2\ v : \llbracket s_1 + s_2 \rrbracket_r}$$

$$\frac{s_1, s_2 : \mathsf{Rep} \qquad v_1 : \llbracket s_1 \rrbracket_r \qquad v_2 : \llbracket s_2 \rrbracket_r}{(v_1, v_2) : \llbracket s_1 \times s_2 \rrbracket_r} \qquad \frac{v : \llbracket r \rrbracket}{\mathsf{r}\ v : \llbracket \mathsf{R} \rrbracket_r}$$

$$\textsc{Data}\ \frac{s : \mathsf{Rep}}{\llbracket s \rrbracket : \star}\ \textsc{where} \quad \frac{s : \mathsf{Rep} \qquad v : \llbracket s \rrbracket_s}{\mathsf{in}\ v : \llbracket s \rrbracket}$$

Fig. 7. A simple universe of types

A Simple Universe. The universe that underlies GMETA is based on a simplified version of the universe for regular tree types by Morris et al. (2004). Morris et al.'s universe is expressive enough to represent recursive types using μ-types (Pierce 2002). However, the presentation of the universe of regular tree types is complicated by the use of *telescopes* (Altenkirch and Reus 1999; McBride and McKinna 2004) for managing μ binders. For presentation purposes and to avoid distractions related to the use of telescopes (which are orthogonal to our purposes), we will use instead a simplified version of regular tree types in which only a single top-level recursive type binder is allowed. This precludes the ability to encode mutually recursive datatypes, which is possible in Morris et al.'s universe. Nevertheless, we have experimental versions of GMETA (both in Coq and Agda) on our online webpage that use the full universe and do support mutually recursive datatypes.

Figure 7 shows the simple universe that is the basis for GMETA. The datatype Rep (defined using the simpler ML-style notation for datatypes) describes the "grammar" of types that can be used to construct the datatypes representable in the universe. The three first constructs represent unit, sum and product types. The K constructor allows the representation of constants of some representable type. The R constructor is the most interesting construct: it is a reference to the recursive type that we are defining. For example, the type representations for naturals and lists of naturals are defined as follows:

$$\begin{array}{ll} \mathsf{RNat} : \mathsf{Rep} & \qquad \mathsf{RList} : \mathsf{Rep} \\ \mathsf{RNat} = 1 + \mathsf{R} & \qquad \mathsf{RList} = 1 + \mathsf{K}\ \mathsf{RNat} \times \mathsf{R} \end{array}$$

The interpretation of the universe is given by two mutually inductive families $\llbracket \cdot \rrbracket_r$ and $\llbracket \cdot \rrbracket$, while the data constructors of these two families provide the syntax to build terms of that universe. The parametric type r in the subscript in $\llbracket \cdot \rrbracket_r$, is the recursive type that is used when interpreting the constructor R. For illustrating the data constructors of terms of the universe, we first define the constructors nil and cons for lists:

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

$Q : \star$ (* Binder type *) $\qquad V : \star$ (* Variable type *)

$$\textsc{Data}\ \frac{r, s : \mathsf{Rep}}{[\![s]\!]_r : \star}\ \textsc{where} \quad \ldots \quad \frac{s : \mathsf{Rep} \qquad v : [\![s]\!]}{\mathsf{e}\ v : [\![\mathsf{E}\ s]\!]_r} \qquad \frac{s_1, s_2 : \mathsf{Rep} \qquad q : Q \qquad v : [\![s_2]\!]_r}{\lambda_{s_1} q.v : [\![\mathsf{B}\ s_1\ s_2]\!]_r}$$

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

Fig. 8. Extending universe with representations of binders and variables

$\mathsf{nil} : [\![\mathsf{RList}]\!]$ $\qquad$ $\mathsf{cons} : [\![\mathsf{RNat}]\!] \to [\![\mathsf{RList}]\!] \to [\![\mathsf{RList}]\!]$
$\mathsf{nil} = \mathsf{in}\ (\mathsf{i}_1\ ())$ $\qquad$ $\mathsf{cons}\ n\ ns = \mathsf{in}\ (\mathsf{i}_2\ (\mathsf{k}\ n, \mathsf{r}\ ns))$

When interpreting $[\![\mathsf{RList}]\!]$, the representation type r in $[\![\cdot]\!]_\mathsf{r}$ stands for $1 + \mathsf{K}\ \mathsf{RNat}\ \times\ \mathsf{R}$. The constructor k takes a value of some interpretation for a type representation s and embeds it in the interpretation for representations of type r. For example, when building values of type $[\![\mathsf{RList}]\!]$, k is used to embed a natural number in the list. Similarly, the constructor r embeds list values in a larger list. The in constructor embeds values of type $[\![\mathsf{r}]\!]_\mathsf{r}$ into a value of inductive family $[\![\mathsf{r}]\!]$, playing the role of a fixpoint. The remaining data constructors (for representing unit, sums and products values) have the expected role, allowing sum-of-product values to be created.

Generic Functions. The key advantage of universes is that we can define (generic) functions that work for any representable datatypes. A simple example is a generic function counting the number of recursive occurrences on a term:

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

To define such generic function, two-mutually inductive definitions are needed. Note that r and s (bound by $\forall$) are implicitly passed in the calls to *size*.

4.3 A Universe for Representing First-Order Binding

We enrich our universe to deal with binders and variables. Figure 7 is insufficient to define generic functions such as substitution and free variables requiring structural information about binders and variables. Figure 8 shows the additional definitions required to support representations of binders, variables, and also deeply embedded terms. The data constructor B of the datatype Rep provides the type for representations of binders. The type Rep is also extended with a constructor E which is the representation type for deeply embedded terms. This constructor is very similar to K. However, the fundamental difference is that generic functions should go inside the terms represented by deeply embedded terms, whereas terms built with K should be treated as constants by generic functions.

The abstract types Q and V represent the types of binders and variables. Depending on the particular first-order representations of binders these types will be instantiated differently.

We illustrate the instantiations of Q and V for 4 of the most popular first-order representations in a table. The last column of the table shows how the lambda term $\lambda x.\ x\ y$ can be encoded in the different approaches. For the nominal approach there is only one sort of variables, which can be represented by a natural number. In this representation, the binders hold information about the bound variables, thus the type Q is the same type as the type of variables V. In the de Bruijn style, the variables are denoted positionally with respect to the current enclosing binder. Thus the type Q is just the unit type and the type V is a natural number. The locally nameless approach can be viewed as a variant of the de Bruijn style. The difference to the de Bruijn style is that parameters and (bound) variables are distinguished. Therefore in the locally nameless style the type V is instantiated to a sum of two natural numbers. Finally, in the locally named style, there are also two sorts of variables and bound variables are represented as in the nominal style. Thus the type Q is a natural number and the type V is a sum type of two naturals. Note that we currently do not support the locally named and nominal style approaches in GMeta as these styles would require special care with issues like alpha-equivalence.

	Q	V	$\lambda x.\ x\ y$
Nominal	$\mathbb{N}$	$\mathbb{N}$	$\lambda x.\ x\ y$
De Bruijn	$\mathbb{1}$	$\mathbb{N}$	$\lambda.\ 0\ 1$
Locally nameless	$\mathbb{1}$	$\mathbb{N}+\mathbb{N}$	$\lambda.\ 0\ y$
Locally named	$\mathbb{N}$	$\mathbb{N}+\mathbb{N}$	$\lambda x.\ x\ a$

The inductive family $[\![\cdot]\!]_\mathsf{r}$ is extended with two new data constructors. The constructor e is similar to the constructor k and is used to build deeply embedded terms. The other constructor uses the standard lambda notation $\lambda_{s_1} q.v$ to denote the constructor for binders. The type representation s_1 is the representation of the syntactic sort of the variables that are bound by the binder, whereas the type representation s_2 is the representation of the syntactic sort of the body of the abstraction. We use $s_1 = \mathsf{R}$ to denote that the syntactic sort of the variables to be bound is the same as that of the body. This distinction is necessary because in certain languages the syntactic sorts of variables to be bound and the body of the abstraction are not the same. For example, in System F, type abstractions in terms such as $\Lambda X.e$ bind type variables X in a term e.

The inductive family $[\![\cdot]\!]$ is also extended with one additional data constructor for variables. This constructor allows terms to be constructed using a variable instead of a concretely defined term.

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

Heterogeneous substitution for (bound) variables:

$$
\begin{array}{ll}
\{\cdot \ \rightarrow \ \cdot\} \ \cdot : \forall (r_1, r_2 : \mathsf{Rep}).\ \mathbb{N} \rightarrow [\![r_1]\!] \rightarrow [\![r_2]\!] \rightarrow [\![r_2]\!] & \\
\{k \ \rightarrow \ u\}\ (\mathsf{in}\ t) & = \mathsf{in}\ (\{k \ \rightarrow \ u\}\ t) \\
\{k \ \rightarrow \ u\}\ (\mathsf{var}\ (\mathsf{inl}\ x)) & = \mathsf{var}\ (\mathsf{inl}\ x) \\
\{k \ \rightarrow \ u\}\ (\mathsf{var}\ (\mathsf{inr}\ y)) & = \textbf{if}\ r_1 \equiv r_2 \wedge k \equiv y\ \textbf{then}\ u\ \textbf{else}\ (\mathsf{var}\ (\mathsf{inr}\ y))
\end{array}
$$

$$
\begin{array}{ll}
\{\cdot \ \rightarrow \ \cdot\} \ \cdot : \forall (r_1, r_2, s : \mathsf{Rep}).\ \mathbb{N} \rightarrow [\![r_1]\!] \rightarrow [\![s]\!]_{r_2} \rightarrow [\![s]\!]_{r_2} & \\
\{k \ \rightarrow \ u\}\ () & = () \\
\{k \ \rightarrow \ u\}\ (\mathsf{k}\ t) & = \mathsf{k}\ t \\
\{k \ \rightarrow \ u\}\ (\mathsf{e}\ t) & = \mathsf{e}\ (\{k \ \rightarrow \ u\}\ t) \\
\{k \ \rightarrow \ u\}\ (\mathsf{i}_1\ t) & = \mathsf{i}_1\ (\{k \ \rightarrow \ u\}\ t) \\
\{k \ \rightarrow \ u\}\ (\mathsf{i}_2\ t) & = \mathsf{i}_2\ (\{k \ \rightarrow \ u\}\ t) \\
\{k \ \rightarrow \ u\}\ (t, v) & = (\{k \ \rightarrow \ u\}\ t, \{k \ \rightarrow \ u\}\ v) \\
\{k \ \rightarrow \ u\}\ (\lambda_{r_3} \mathbb{1}.t) & = \textbf{if}\ (r_3 \equiv \mathsf{R} \wedge r_1 \equiv r_2) \vee (r_3 \not\equiv \mathsf{R} \wedge r_1 \equiv r_3) \\
 & \quad \textbf{then}\ \lambda_{r_3} \mathbb{1}.(\{(k+1) \ \rightarrow \ u\}\ t)\ \textbf{else}\ \lambda_{r_3} \mathbb{1}.(\{k \ \rightarrow \ u\}\ t) \\
\{k \ \rightarrow \ u\}\ (\mathsf{r}\ \ t) & = \mathsf{r}\ \ (\{k \ \rightarrow \ u\}\ t)
\end{array}
$$

Heterogeneous substitution for parameters in the following form are similarly defined:

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

Example of a heterogeneous lemma:

$$\mathit{subst\_fresh} : \forall (r_1, r_2 : \mathsf{Rep})\ (t : [\![r_1]\!])\ (u : [\![r_2]\!])\ (m : \mathbb{N}),\ m \notin (\mathit{fv}_{r_2}\ t) \Rightarrow [m \rightarrow u]\ t = t$$

Fig. 9. Generic definitions for the locally nameless approach

5 Generic Operations and Lemmas

This section shows how generic operations and lemmas defined over the universe presented in Section 4 can be used to provide much of the basic infrastructure boilerplate for the languages representable in the universe.

5.1 Locally Nameless

Figure 9 presents generic definitions for the locally nameless approach. In this approach binders do not bind names, and (bound) variables and parameters (free variables) are distinguished. Thus, as discussed in Section 4.3, the types Q and V are, respectively, the unit type[3] and a sum of two naturals. Using these instantiations for Q and V, the operation for instantiating a (bound) variable with a term can be defined in a generic way. Also, generic lemmas can be defined using the generic operations. The statement for *subst_fresh* – which states that if a parameter does not occur in a term, then substitution of that parameter is the identity – is shown as an example of such generic lemmas.

As explained in Section 4, generic operations are defined over terms of the universe by two mutually-inductive operations defined over the $[\![\cdot]\!]$ and $[\![\cdot]\!]_r$

[3] For convenience, we use $\mathbb{1}$ for both the unit type and the unique term of that type.

(mutually-)inductive families. Note that our generic definition for substitution[4] effectively deals with all the possible combinations for defining a substitution in a multi-sorted syntax.

In the definition of substitutions the most interesting cases are variables and binders. In the case of variables, the condition $r_1 \equiv r_2$ is necessary to check whether the parameter (or variable) and the term to be substituted have the same representation. Note the use of $(\equiv)$ to compare type representations: the universe supports decidable equality, which is crutial for the definition of operations. The subscript r_3 keeps the information about which kind of variables is to be bound. When $r_3 = \mathsf{R}$, the binding is homogeneous, that is, the variable to be bound and the body of the binder have the same representation. For example, the term-level abstraction in terms $(\lambda x : T.e)$ of System F is homogeneous. An example of heterogeneous binding is the type-level abstraction in terms $(\Lambda X.e)$ of System F. In this case r_3 is the representation for System F types. Variable substitution happens when the bound variable and the terms to be substituted have the same representation. Note that, in the case of homogeneous binding $(r_3 \equiv \mathsf{R})$, we compare r_1 with r_2, not with r_3, because the bound variable and the body of the binder have the same representation r_2.

The main advantage of representing the syntax of languages with our generic universe is, of course, that all generic operations are immediately available. For instance, the 8 substitution operations mentioned in Section 1 can be recovered through suitable instantiations of the type representations r_1, r_2, r_3 in the two generic substitutions presented in this section.

5.2 De Bruijn

A key advantage of our modular approach is that we do not have to commit to using a particular first-order representation. Instead, by suitably instantiating the types Q and V, we can define the generic infrastructure for our own favored first-order representation. For example we can use GMeta to define the generic infrastructure for de Bruijn representations. In de Bruijn representations, binders do not bind any names, therefore the type Q is instantiated with the unit type. Also, because there is only one sort of (positional) variable, the type V is instantiated with the type of natural numbers. The implementation of heterogeneous generic shifting follows a pattern similar to that used in the generic operations for the locally nameless style for dealing with homogeneous and heterogeneous binders. The variable and binder cases implement the expected behavior for the de Bruijn indices operations and all the other cases are limited to traversal code. For more details we refer to the GMeta homepage.

6 Discussion and Evaluation

In this section we present the results of the case studies that we conducted. The discussion of these results is done in terms of three criteria proposed by

[4] Note that the notation for substitutions follows Aydemir et al. (2008).

		Definitions	Infrastructure	Core	Overall		
			(lemma + def.)	(lemma + def.)	inf. overhead	total	ratio
STLC	Aydemir et al.	**11**	**13** + 3	**4** + 0	17	31	55%
(locally nameless)	GMeta	7	4 + 0	4 + 0	1	15	7%
System $F_{<:}$	Aydemir et al.	**20**	**48** + 7	**17** + 1	60	93	65%
(locally nameless)	GMeta	13	26 + 1	17 + 1	25	58	43%
System $F_{<:}$	Vouillon	27	24 + 0	50 + 0	41	101	41%
(de Bruijn)	GMeta	12	1 + 0	52 + 0	3	65	5%

Fig. 10. Formalization of POPLmark challenge (part 1A+2A) and STLC in Coq using locally nameless approach and de Bruijn approach with and without GMeta

Aydemir et al. (2005) (*reasonable overheads*, *cost of entry* and *transparency*) for evaluating mechanizations of formal metatheory.

Reasonable Overheads. The biggest benefit of GMeta is that it significantly lowers the overheads required in mechanical formalizations by providing reuse of the basic infrastructure. Figure 10 presents the detailed numbers obtained in our case studies. We follow Aydemir et al. by dividing the whole development into three parts: *definitions*, *infrastructure* and *core*. The numbers on those columns correspond to the number of definitions and lemmas used for each part. The definitions column presents the number of basic definitions about syntax, whereas the core column presents the number of main definitions and lemmas of the formalization (such as, for example, progress and preservation). The infrastructure column is the most interesting because this is where most of the tedious boilerplate lemmas and definitions are. The column boilerplate counts the number of such definitions and lemmas across the formalizations. Although, for the most part, boilerplate comes from the infrastructure part, some boilerplate also exists in the definitions part. This explains why GMeta is able to reduce the number of definitions and lemmas in the two parts. The numbers in bold face are the numbers that were presented by Aydemir et al. (2008). However those numbers did not reflect the real total number of definitions and lemmas in the solutions. For example, in the infrastructure part only the lemmas were counted. Since we are interested in all the boilerplate, our numbers reflect the total number of definitions and lemmas in each part.

In comparison with Aydemir et al.'s reference solutions, the proofs in our approach follow essentially the structure of the original proofs. One minor difference is that instead of some standard Coq tactics, a few more general tactics provided by GMeta should be used. Because this is the only significant difference, the proofs in the GMeta solution and Aydemir et al.'s solution have comparable sizes. This means that most proofs will still be comparable in size although a small number of proofs will be either shorter or longer.

Cost of Entry. One important criterion for evaluating mechanical formalizations of metatheory is the associated cost of entry. That is, how much does a user need to know in order to successfully develop a formalization? We believe that the associated cost of entry of GMeta is comparable to first-order approaches like the one by Aydemir et al. (2008).

One aspect of GMeta that (arguably) requires less knowledge when compared to Aydemir et al. (2008) is that the end-user does not need to know how to prove many basic infrastructure lemmas, since those are provided by GMeta's libraries.

Finally, we should mention that one advantage of generative approaches such as LNgen (Aydemir and Weirich 2009) is that the cost-of-entry, in terms of using the lemmas and definitions provided by LNgen, is a bit lower than in GMeta. This is because the generated infrastructure is directly defined in terms of the object language and the lemmas and definitions can be used as if they had been written by hand. In GMeta, the end-user, while not required to know about DGP, still needs to be aware of some special simplification tactics and, occasionally, he may need to apply adequacy lemmas by hand.

Transparency. The transparency criterion is intended to evaluate how easy it is for humans to understand particular formalization techniques. The issue of transparency is largely orthogonal to GMeta because it usually measures how particular representations of binding (such as locally nameless or de Bruijn), and lemmas and definitions using that approach, are easy to understand by humans. Since we do not introduce any new representation, transparency remains unchanged (the same representation, lemmas and definitions are used).

7 Related Work

Generative Approaches. Closest to our work are *generative* approaches like LNgen, which uses an external tool, based on Ott (Sewell et al. 2010) specifications, to generate the infrastructure lemmas and definitions for a particular language automatically. One advantage of generative approaches is that the generated infrastructure is directly defined in terms of the object language. In contrast, in GMeta, the infrastructure is indirectly defined in terms of generic definitions. This is not entirely ideal, but it is possible to handle the situation in a reasonably effective way in GMeta using tactics (see Section 3.1).

There are two main advantages of a DGP approach over generative approaches: *verifiability*; and *extensibility*. Although a generator allows defining once-and-for all the infrastructure, it would not be a simple task to verify once-and-for all that the generator always generates correct (well-typed) infrastructure. With a generator, we can only verify whether each particular generated set of infrastructure is correct. Another advantage of a libary-based approach is that it is easy to extend. If we wanted to add a new lemma, we would just need to extend a module with a new generic function. With a generator, this would amount to directly changing the generator code. Although there is also a cost to extending libraries, we believe that it is usually easier than changing the generator code.

It is also interesting to compare GMeta and LNgen in terms of which types of infrastructure they can reuse and how hard it is to reuse such infrastructure. The main advantage of LNgen is that dealing with inductive relations is easy. In GMeta, lemmas involving well-formedness require some more effort to be reused. A solution for this problem would be to extend the isomorphism generator to deal with inductive relations as well. On the other hand, the strength of GMeta lies in its extensibility. For example, sometimes there are domain-specific infrastructure lemmas like *thbfsubst_perm_core* in Figure 5. Dealing with such a infrastructure is in conflict with the general-purpose nature of LNgen.

DGP and Binding. DGP techniques have been used before for dealing with binders using a well-scoped de Bruijn index representation (Altenkirch and Reus 1999; McBride and McKinna 2004). Chlipala (2007) used an approach inspired by *proof by reflection techniques* (Boutin 1997) to provide several generic operations on well-typed terms represented by well-scoped de Bruijn indices. Licata and Harper (2009) proposed a universe in Agda that permits definitions that mix binding and computation. The obvious difference is that GMeta works with traditional (non-well-scoped) first-order representations instead of well-scoped de Bruijn indices. This difference of representation means that the universes and generic functions have to deal with significantly different issues and that they are quite different in nature. More fundamentally, Chlipala's (2007) and Licata and Harper's (2009) work can be viewed as trying to develop new ways to formalize metatheory in which many of the invariants hold by construction, that would have to be proved otherwise. This is different from our goal: we are not proposing new ways to formalize metatheory, rather we wish to make well-established ways to formalize metatheory with first-order representations less painful to use.

DGP techniques have also been widely used in conventional functional programming languages (Jansson and Jeuring 1997; Hinze and Jeuring 2003; Rodriguez et al. 2008), and Cheney (2005) explored how to provide generic operations such as substitution or collecting free variables using nominal abstract syntax.

Our work is inspired by the use of *universes* in type-theory (Martin-Löf 1984; Nordström et al. 1990). The basic universe construction presented in Figure 7 is a simple variation of the *regular tree types* universe proposed by Morris et al. (2004, 2009) in Epigram. Nevertheless the extensions for representing variables and binders presented in Figure 8 are new. Dybjer and Setzer (1999, 2001) showed universe constructions within a type-theory with an axiomatization of induction-recursion. Altenkirch and McBride (2003) proposed a universe capturing the datatypes and generic operations of Generic Haskell (Hinze and Jeuring 2003) and Norell (2008) shows how to do DGP with universes in Agda (Norell 2007).

Verbruggen et al. (2008, 2009) formalized a Generic Haskell (Hinze and Jeuring 2003) DGP style in Coq, which can also be used to do generic programming. This approach allows conventional datatypes to be expressed, but it cannot be used to express meta-theoretical generic operations since there are no representations for variables or binders.

Other Techniques for First-Order Approaches. Aydemir et al. (2009) investigated several variations of representing syntax with locally nameless representations aimed at reducing the amount of infrastructure overhead in languages like System $F_{<:}$. One advantage of these techniques is that they are very lightweight in nature and do not require additional tool support. However, while the proposed techniques are effective at achieving significant savings, they require the abstract syntax of the object language to be encoded in a way different from the traditional locally nameless style, potentially collapsing all syntactic sorts into one. In contrast, GMeta allows the syntax to be encoded in the traditional locally nameless style, while at the same time reducing the infrastructure overhead through its reusable libraries of infrastructure.

Higher-Order Approaches and Nominal Logic. Approaches based on higher-order abstract syntax (HOAS) (Pfenning and Elliot 1988; Harper et al. 1993) are used in logical frameworks such as Abella (Gacek 2008), Hybrid (Momigliano et al. 2008) or Twelf (Pfenning and Schürmann 1999). In HOAS, the object-language binding is represented using the binding of the meta-language. This has the important advantage that facts about substitution or alpha-equivalence come for free since the binding infrastructure of the meta-language is reused. It is well-known that in Coq it is not possible to use the usual HOAS encodings, although Despeyroux et al. (1995) and Chlipala (2008) have shown how weaker variations of HOAS can be encoded in Coq. Popescu et al. (2010) investigate how formalizations using HOAS can avoid standard problems by being encoded on top of first-order representations. Approaches like GMeta or LNgen are aimed at recovering many of the properties that one expects from a logical framework for free.

Nominal logic (Pitts 2003) is an extension of first-order logic that allows reasoning about alpha-equivalent abstract syntax in a generic way. Variants of nominal logic have been adopted in the Nominal Isabelle (Urban 2005). However, because Coq does not have a nominal variant, this approach cannot be used in Coq formalizations.

8 Conclusion

There are several techniques for formalizing metatheory using first-order representations, which typically involve developing the whole of the infrastructure by hand each time for a new formalization. GMeta improves on these techniques by providing reusable generic infrastructure in libraries, avoiding the repetition of definitions and lemmas for each new formalization. The DGP approach used by GMeta not only allows an elegant and verifiable formulation of the generic infrastructure which is appealing from the theoretical point of view, but also shows itself useful for conducting realistic formalizations of metatheory.

Acknowledgements. We are especially grateful to Sungwoo Park, which has provided us with many useful comments and suggestions. We thank Hugo Herbelin, Randy Pollack, Stephanie Weirich, the members of ROPAS and the anonymous reviewers for their useful comments and suggestions.

References

Altenkirch, T., McBride, C.: Generic programming within dependently typed programming. In: IFIP TC2/WG2.1 Working Conference on Generic Programming (2003)

Altenkirch, T., Reus, B.: Monadic Presentations of Lambda Terms Using Generalized Inductive Types. In: Flum, J., Rodríguez-Artalejo, M. (eds.) CSL 1999. LNCS, vol. 1683, pp. 453–468. Springer, Heidelberg (1999)

Aydemir, B., Weirich, S., Zdancewic, S.: Abstracting syntax. Technical Report MS-CIS-09-06, University of Pennsylvania (2009)

Aydemir, B.E., Weirich, S.: LNgen: Tool Support for Locally Nameless Representations (2009) (Unpublished manuscript)

Aydemir, B.E., Bohannon, A., Fairbairn, M., Foster, J.N., Pierce, B.C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., Zdancewic, S.: Mechanized Metatheory for the Masses: The PoplMark Challenge. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 50–65. Springer, Heidelberg (2005)

Aydemir, B.E., Charguéraud, A., Pierce, B.C., Pollack, R., Weirich, S.: Engineering formal metatheory. In: POPL 2008 (2008)

Boutin, S.: Using Reflection to Build Efficient and Certified Decision Procedures. In: Ito, T., Abadi, M. (eds.) TACS 1997. LNCS, vol. 1281, pp. 515–529. Springer, Heidelberg (1997)

Cheney, J.: Scrap your nameplate (functional pearl). In: ICFP 2005 (2005)

Chlipala, A.: A certified type-preserving compiler from lambda calculus to assembly language. In: PLDI 2007 (2007)

Chlipala, A.: Parametric higher-order abstract syntax for mechanized semantics. In: ICFP 2008 (2008)

The Coq Development Team. The Coq Proof Assistant Reference Manual, Version 8.2 (2009), `http://coq.inria.fr`

de Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. Indagationes Mathematicae (Proceedings) 75(5), 381–392 (1972)

Despeyroux, J., Felty, A.P., Hirschowitz, A.: Higher-Order Abstract Syntax in Coq. In: Dezani-Ciancaglini, M., Plotkin, G. (eds.) TLCA 1995. LNCS, vol. 902, pp. 124–138. Springer, Heidelberg (1995)

Dybjer, P.: Inductive families. Formal Aspects of Computing 6, 440–465 (1997)

Dybjer, P., Setzer, A.: A Finite Axiomatization of Inductive-Recursive Definitions. In: Girard, J.-Y. (ed.) TLCA 1999. LNCS, vol. 1581, pp. 129–146. Springer, Heidelberg (1999)

Dybjer, P., Setzer, A.: Indexed Induction-Recursion. In: Kahle, R., Schroeder-Heister, P., Stärk, R.F. (eds.) PTCS 2001. LNCS, vol. 2183, pp. 93–113. Springer, Heidelberg (2001)

Gacek, A.: The Abella Interactive Theorem Prover (System Description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 154–161. Springer, Heidelberg (2008)

Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. J. ACM 40(1), 143–184 (1993)

Hinze, R., Jeuring, J.: Generic Haskell: Practice and Theory. In: Backhouse, R., Gibbons, J. (eds.) Generic Programming SS 2002. LNCS, vol. 2793, pp. 1–56. Springer, Heidelberg (2003)

Jansson, P., Jeuring, J.: PolyP—a polytypic programming language extension. In: POPL 1997 (1997)

Licata, D.R., Harper, R.: A universe of binding and computation. In: ICFP 2009 (2009)
Martin-Löf, P.: Intuitionistic Type Theory. Bibliopolis (1984)
McBride, C., McKinna, J.: The view from the left. J. Funct. Program. 14(1), 69–111 (2004)
McKinna, J., Pollack, R.: Pure Type Systems Formalized. In: Bezem, M., Groote, J.F. (eds.) TLCA 1993. LNCS, vol. 664, pp. 289–305. Springer, Heidelberg (1993)
Momigliano, A., Martin, A.J., Felty, A.P.: Two-level hybrid: A system for reasoning using higher-order abstract syntax. Electron. Notes Theor. Comput. Sci. 196, 85–93 (2008)
Morris, P., Altenkirch, T., McBride, C.: Exploring the Regular Tree Types. In: Filliâtre, J.-C., Paulin-Mohring, C., Werner, B. (eds.) TYPES 2004. LNCS, vol. 3839, pp. 252–267. Springer, Heidelberg (2006)
Morris, P., Altenkirch, T., Ghani, N.: A universe of strictly positive families. Int. J. Found. Comput. Sci. 20(1), 83–107 (2009)
Nordström, B., Peterson, K., Smith, J.M.: Programming in Martin-Löf's Type Theory: An Introduction. Oxford Unversity Press (1990)
Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis, Chalmers University of Technology (2007)
Norell, U.: Dependently Typed Programming in Agda. In: Koopman, P., Plasmeijer, R., Swierstra, D. (eds.) AFP 2008. LNCS, vol. 5832, pp. 230–266. Springer, Heidelberg (2009)
Paulin-Mohring, C.: Définitions Inductives en Théorie des Types d'Ordre Supérieur. Habilitation à diriger les recherches, Université Claude Bernard Lyon I (1996)
Peyton Jones, S., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for GADTs. In: ICFP 2006 (2006)
Pfenning, F., Elliot, C.: Higher-order abstract syntax. In: PLDI 1988 (1988)
Pfenning, F., Schürmann, C.: System Description: Twelf - A Meta-Logical Framework for Deductive Systems. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 202–206. Springer, Heidelberg (1999)
Pierce, B.C.: Types and Programming Languages. The MIT Press (2002)
Pitts, A.M.: Nominal logic, a first order theory of names and binding. Inf. Comput. 186(2), 165–193 (2003)
Popescu, A., Gunter, E.L., Osborn, C.J.: Strong Normalization for System F by HOAS on top of FOAS. In: LICS, pp. 31–40 (2010)
Rodriguez, A., Jeuring, J., Jansson, P., Gerdes, A., Kiselyov, O., Oliveira, B.C.d.S.: Comparing libraries for generic programming in Haskell. In: Haskell 2008 (2008)
Rossberg, A., Russo, C.V., Dreyer, D.: F-ing modules. In: TLDI 2010 (2010)
Sato, M., Pollack, R.: External and internal syntax of the lambda-calculus. J. Symb. Comput. 45(5), 598–616 (2010)
Sewell, P., Nardelli, F.Z., Owens, S., Peskine, G., Ridge, T., Sarkar, S., Strniša, R.: Ott: Effective tool support for the working semanticist. J. Funct. Program. 20(01), 71–122 (2010)
Urban, C., Tasson, C.: Nominal Techniques in Isabelle/HOL. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 38–53. Springer, Heidelberg (2005)
Verbruggen, W., de Vries, E., Hughes, A.: Polytypic programming in COQ. In: WGP 2008 (2008)
Verbruggen, W., de Vries, E., Hughes, A.: Polytypic properties and proofs in Coq. In: WGP 2009 (2009)
Vouillon, J.: Poplmark solutions using de bruijn indices (2007), `https://alliance.seas.upenn.edu/~plclub/cgi-bin/poplmark/`