ROSAEC center
Research On Software Analysis for Error-free Computing
소프트웨어 무결점 연구센터 NRF ERC

ROSAEC MEMO
2011-010
April 12, 2011

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

Bruno C. d. S. Oliveira
Seoul National University
bruno@ropas.snu.ac.kr

Gyesik Lee
Seoul National University
gslee@ropas.snu.ac.kr

Sungkeun Cho
Seoul National University
skcho@roaps.snu.ac.kr

Kwangkeun Yi
Seoul National University
kwang@ropas.snu.ac.kr

April 12, 2011

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 then 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.

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 by the meta-logic. This is why such approaches are used in logical frameworks such as Abella (Gacek 2008), Hybrid (Momigliano et al. 2008) or Twelf (Pfenning and Schürmann 1999); and have also been advocated as an interesting alternative (Despeyroux et al. 1995; Chlipala 2008) for formalizing metatheory 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 (based on the nominal approach) and terms with binders are easy to manipulate.

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 typically a set of *infrastructure operations*

		trm	typ
trm	Variables	$bsubst_{\mathsf{trm}\times\mathsf{trm}}$	$bsubst_{\mathsf{trm}\times\mathsf{typ}}$
	Parameters	$fsubst_{\mathsf{trm}\times\mathsf{trm}}$	$fsubst_{\mathsf{trm}\times\mathsf{typ}}$
typ	Variables	$bsubst_{\mathsf{typ}\times\mathsf{trm}}$	$bsubst_{\mathsf{typ}\times\mathsf{typ}}$
	Parameters	$fsubst_{\mathsf{typ}\times\mathsf{trm}}$	$fsubst_{\mathsf{typ}\times\mathsf{typ}}$

Figure 1: Possible variations of substitutions for parameters and variables for a language with two syntactic sorts (trm and typ) in the locally nameless style.

and associated *lemmas* that should be implemented. In the locally nameless style (Aydemir et al. 2008) and locally named (Mckinna and Pollack 1993) styles, for example, we usually need operations like substitutions 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.

The problem is that, in many cases, 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 substitutions will be needed. Basically 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 usually needed in formalizations, many of them are. For example, System $F_{<:}$, which is the language formalized 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).

Importantly, it is insufficient to consider only *homogeneous* operations like $bsubst_{\mathsf{trm}\times\mathsf{trm}}$, which perform substitutions of variables on terms of the same sort (trm). In the general case we also have to consider *heterogeneous* operations, such as $bsubst_{\mathsf{typ}\times\mathsf{trm}}$, in which the sort of variables being substituted (typ) is not of the same sort as the terms which are being substituted (trm). In languages of the System F family, operations like $bsubst_{\mathsf{typ}\times\mathsf{trm}}$ and $fsubst_{\mathsf{typ}\times\mathsf{trm}}$ are needed because those languages support type abstractions in terms ($\Lambda X.e$) and, consequently, we need to provide a way to substitute type variables in terms.

As the number of syntactic sorts and binding constructs increases, there is a combinatorial explosion of the number of infrastructure lemmas and operations. Indeed, reports from larger formalizations confirm that the problem only gets worse with size: in a type-directed translation from an ML-module language to System F_ω using locally nameless style (Rossberg et al. 2010) the authors report that "*Out of a total of around 550 lemmas, approximately 400 were tedious infrastructure lemmas*".

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 forall 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.

Our realization of this idea is GMeta: a generic framework for first-order representations of variable binding, implemented in the Coq theorem prover[1]. In GMeta a DGP *uni-*

[1]We also have an experimental Agda implementation.

```
(*@Iso typ_iso {
  Parameter typ_fvar,
  Variable  typ_bvar,
  Binder    typ_all _
}*)
Inductive typ :=
 | typ_fvar  : ℕ → typ
 | typ_bvar  : ℕ → typ
 | typ_top   : typ
 | typ_arrow : typ → typ → typ
 | typ_all   : typ → typ → typ.
(*@Iso trm_iso {
  Parameter trm_fvar,
  Variable  trm_bvar,
  Binder    trm_abs _,
  Binder    trm_tabs _ binds typ
}*)
Inductive trm :=
 | trm_fvar : ℕ → trm
 | trm_bvar : ℕ → trm
 | trm_app  : trm → trm → trm
 | trm_abs  : typ → trm → trm
 | trm_tapp : trm → typ → trm
 | trm_tabs : typ → trm → trm.
```

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

verse (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 itself 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 operations and lemmas for the various first-order representations.

The infrastructure is reused by users through *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:

Module $M_{\mathsf{trm}\times\mathsf{trm}}$:= *LNTemplate trm_iso trm_iso*.
Module $M_{\mathsf{typ}\times\mathsf{typ}}$:= *LNTemplate typ_iso typ_iso*.
Module $M_{\mathsf{typ}\times\mathsf{trm}}$:= *LNTemplate typ_iso trm_iso*.

Essentially, 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. GMeta shows that DGP is a viable approach to address the problem and that it has some important advantages over alternatives like LNGen (Aydemir and Weirich 2009) (see Section 7). Nevertheless, GMeta is still a proof-of-concept tool and, as discussed in Section 6.4, a bit more support from theorem provers is desirable for improving the usability of the tool.
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. Approaches based on code generation can only provide reuse, but they do not guarantee soundness and they are hard to extend.

- *Heterogeneous generic operations and lemmas*: Of particular interest is the ability of GMeta to deal with challenging 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 Different Types of Infrastructure in GMeta

The purpose of this section is to define what should be considered infrastructure lemmas and definitions and to distinguish between the different varieties of these lemmas and definitions. A classification of infrastructure is desirable for two reasons:

1. Some infrastructure is easier to reuse than other in GMeta. Also some infrastructure is more general than other.
2. There is an informal understanding of what infrastructure is, but the existing literature is not very clear as to what actually constitutes infrastructure (Aydemir et al. 2008; Aydemir and Weirich 2009).

To help with the characterization of infrastructure, we propose two classifications, by difficulty to reuse and by generality of the infrastructure.

2.1 Classification by difficulty to reuse

Type 1. Common operations and lemmas: Operations such as free and bound variables, term size or substitution-like operations (such as substitutions for parameters and variables in the locally nameless style; or shifting in the de Bruijn style) are of this type. For example:

$$\begin{aligned}
&\mathit{fsubst}_{\mathsf{trm}\times\mathsf{trm}} : \mathbb{N} \to \mathsf{trm} \to \mathsf{trm} \to \mathsf{trm}\\
&\mathit{bsubst}_{\mathsf{trm}\times\mathsf{trm}} : \mathbb{N} \to \mathsf{trm} \to \mathsf{trm} \to \mathsf{trm}\\
&\mathit{bsubst}_{\mathsf{typ}\times\mathsf{trm}} : \mathbb{N} \to \mathsf{typ} \to \mathsf{trm} \to \mathsf{trm}
\end{aligned}$$

Also, basic lemmas about the operations (such as several forms of permutation lemmas about substitutions, or lemmas involving some simple conditions about variables) fall in this category. For example:

$$\begin{aligned}
&\mathit{tfsubst\_lemma} : \forall (T\ U\ V : \mathsf{trm})\ (a\ b : \mathbb{N}),\\
&\quad a \neq b \Rightarrow\\
&\quad a \notin (\mathit{fv}_{\mathsf{trm}\times\mathsf{trm}}\ V) \Rightarrow\\
&\quad \mathit{fsubst}_{\mathsf{trm}\times\mathsf{trm}}\ (\mathit{fsubst}_{\mathsf{trm}\times\mathsf{trm}}\ T\ a\ U)\ b\ V =\\
&\quad \mathit{fsubst}_{\mathsf{trm}\times\mathsf{trm}}\ (\mathit{fsubst}_{\mathsf{trm}\times\mathsf{trm}}\ T\ b\ V)\ a\ (\mathit{fsubst}_{\mathsf{trm}\times\mathsf{trm}}\ U\ b\ V)
\end{aligned}$$

Type 2. Lemmas involving unary inductive relations (predicates): When formalizing metatheory it is common to use inductive relations. For example several different forms of well-formedness are usually described in this way. For unary inductive relations, which are essentially predicates, we can use the predicate form instead to provide the associated lemmas in GMeta. An example is the closed terms predicate:

$$\mathit{closed}_{\mathsf{trm}\times\mathsf{trm}} : \mathsf{trm} \to \mathsf{Prop}$$

[2]`http://ropas.snu.ac.kr/gmeta/`

There are many permutations that only hold for closed terms. For example:

$\mathit{tbsubst\_var\_twice\_wf} : \forall (T : \mathsf{trm})\ k\ (U\ \ V : \mathsf{trm}),$
$\quad \mathit{closed}_{\mathsf{trm}\times\mathsf{trm}}\ V \Rightarrow$
$\quad \mathit{bsubst}_{\mathsf{trm}\times\mathsf{trm}}\ T\ k\ V = \mathit{bsubst}_{\mathsf{trm}\times\mathsf{trm}}\ (\mathit{bsubst}_{\mathsf{trm}\times\mathsf{trm}}\ T\ k\ V)\ k\ U$

Type 3. Lemmas involving general inductive relations: General inductive relations can be dealt in GMeta with a corresponding generic inductive relation. However, there is a cost in using these generic inductive relations because they require a mapping between the concrete relation defined directly over the object language and the generic relation. This mapping requires additional work from the user as well as some basic knowledge about DGP, so they are not as easy to reuse as type 1 and 2. The typical example is well-formed terms in an environment:

$\mathit{envT}_{\mathsf{typ}\times\mathsf{trm}} : (\mathit{env}\ \mathsf{typ}) \to \mathsf{trm} \to \mathsf{Prop}$

For which there are many interesting lemmas that are useful in formalizations. For example:

$\mathit{envT\_Twf} : \forall (E : \mathit{env}\ \mathsf{typ})\ (T : \mathsf{trm}),$
$\quad \mathit{envT}_{\mathsf{typ}\times\mathsf{trm}}\ E\ T \Rightarrow$
$\quad \forall (k : \mathbb{N})\ (U : \mathsf{trm}), T = \mathit{bsubst}_{\mathsf{trm}\times\mathsf{trm}}\ T\ k\ U$

In short, it is easy to reuse infrastructure of types 1 and 2 in GMeta, but reusing infrastructure of type 3 requires a bit more work.

2.2 Classification by generality

Some infrastructure is general whereas other is domain-specific (that is, only exist for particular types of languages). We propose the following classification to distinguish between these two types of infrastructure:

Type A. General infrastructure Lemmas and definitions involving only one binder and a single combination of sorts are pervasive: they are definable for any object language with some binding structure. The examples that we gave for infrastructure of type 1 and 2 in Section 2.1 are of this type.

Type B. Domain-specific infrastructure Lemmas and definitions involving multiple binders or multiple combinations of sorts tend to be specific to particular languages. For example the lemma *envT_Twf* uses two different combinations ($\mathsf{trm}\times\mathsf{trm}$ and $\mathsf{typ}\times\mathsf{trm}$). This lemma makes sense when we are talking about some typed languages, but it does not make sense for many other languages. Another example is:

$\mathit{fbfsubst\_perm\_core}_{\mathsf{typ}\times\mathsf{trm}} : \forall (t : \mathsf{trm})\ (u, v : \mathsf{typ})\ (m\ k : \mathbb{N}),$
$\quad \mathit{closed}_{\mathsf{typ}\times\mathsf{typ}}\ u \Rightarrow$
$\quad \mathit{bsubst}_{\mathsf{typ}\times\mathsf{trm}}\ k\ (\mathit{fsubst}_{\mathsf{typ}\times\mathsf{typ}}\ m\ u\ v)\ (\mathit{fsubst}_{\mathsf{typ}\times\mathsf{trm}}\ m\ u\ t)$
$\quad = \mathit{fsubst}_{\mathsf{typ}\times\mathsf{trm}}\ m\ u\ (\mathit{bsubst}_{\mathsf{typ}\times\mathsf{trm}}\ k\ v\ t)$

This lemma involves substitutions on different kind of binders ($\mathit{fsubst}_{\mathsf{typ}\times\mathsf{typ}}$ and $\mathit{fsubst}_{\mathsf{typ}\times\mathsf{trm}}$) and it makes sense when the language has 2 different binders on 2 sorts.

The 2 classifications proposed by us are mostly orthogonal. We will use a combination between the letter A or B and the numbers 1, 2 and 3 to refer to a particular type on which a lemma or definition falls into. For example, *tfsubst_lemma* is of type A1 (that is, it is a general common lemma), whereas $\mathit{fbfsubst\_perm\_core}_{\mathsf{typ}\times\mathsf{trm}}$ is of type B2 (that is, it is a domain-specific

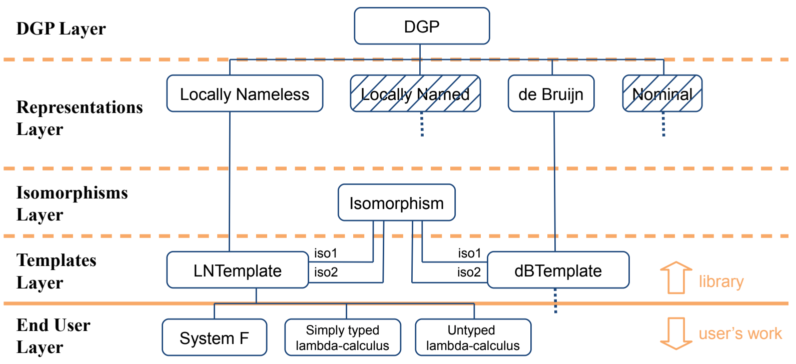

Figure 3: A simplified modular structure overview of GMeta.

lemma using a predicate). As a remark, we noticed that in practice lemmas that fall in the category A3 are rare and we do not have any examples of these in GMeta.

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. The GMeta framework is structured into 5 layers. In this paper we will discuss two layers in detail in Sections 4 and 5. These layers are related to DGP and are the most interesting from a technical point of view. The other layers will not be introduced in detail due to the lack of space, although the key ideas are discussed in this section. These other layers are important because of the convenience aspect of GMeta, but they are less interesting from a technical point of view. More information about these layers is available in GMeta's webpage.

3.1 GMeta's modular structure

Figure 3 shows an overview of the modular structure of GMeta. The structure is hierarchical, with the more general modules at the top and the more specific modules at the bottom.

- **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, and we are planning to support more representations in the future.

- **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 language and their corresponding representations in the generic language. In the Figure 3 we show only *LNTemplate* and *dBTemplate*, which are the foundamental templates providing reuse for the general infrastructure (type A). In GMeta there are also other templates which can be *optionally* used for reusing additional infrastructure. However, those templates tend to be more domain-specific (that is they cover type B infrastructure).

- **End User Layer:** End users will use GMeta's libraries to develop metatheory for particular object languages such as for example, the simply typed lambda calculus (STLC) or System $F_{<:}$ used in our case studies.

3.2 Making GMeta Convenient to Use

The main design goal of GMeta is to make it useful for users wishing to conduct formalizations of metatheory in Coq using conventional first-order representations, without significantly increasing the *cost-of-entry* to users. Importantly, although DGP plays a fundamental role in the definition of the core libraries of GMeta (at the DGP and representation layers), end users should not need knowledge about DGP for basic uses of GMeta. However this is not trivial to achieve because, among other things, end-user proofs (such as soundness) 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. Therefore, to provide convenience to the user and to avoid that users need to know about DGP, GMeta employs the techniques described next.

Automatically generated isomorphisms: Isomorphisms are a well-know technique to allow using DGP with conventional user-defined datatypes (or inductive definitions) such as the ones presented in Figure 2. Like in most libraries for DGP in functional programming languages (Rodriguez et al. 2009), GMeta uses automatically generated isomorphisms between the user-defined object language and a corresponding representation of that language the generic universe. However, unlike the isomorphisms used in DGP libraries in functional programming languages, GMeta's isomorphisms cannot be generated automatically without additional information about the binding structure of the language. To deal with this problem GMeta uses a small annotation language to describe the binding structure of the object language. This allows the generation of isomorphisms which are aware of the correct binding structure for that 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. The definitions and lemmas in templates are very simple. For example, the definition of parameter substitution in the locally nameless template is:

Module *LNTemplate* $(iso_{\mathsf{S}_1} : \mathbf{Iso}, iso_{\mathsf{S}_2} : \mathbf{Iso})$.

$\ldots$

$[\cdot \ \rightarrow \ \cdot]_T \ \cdot \quad : \mathbb{N} \rightarrow \mathsf{S}_1 \rightarrow \mathsf{S}_2 \rightarrow \mathsf{S}_2$

$[k \ \rightarrow \ u]_T \ t = to_{\mathsf{S}_2} \ ([k \ \rightarrow \ (from_{\mathsf{S}_1} \ u)] \ (from_{\mathsf{S}_2} \ t))$

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 iso_{S_1} and iso_{S_2}, which are the parameters of *LNTemplate*. Definitions like $[\cdot \ \rightarrow \ \cdot]_T \ \cdot$ are simply using the isomorphism (through the operations to_{S_2}, $from_{\mathsf{S}_1}$ and $from_{\mathsf{S}_2}$) to interface with generic operations like $[\cdot \ \rightarrow \ \cdot] \ \cdot$ (see Figure 8) 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 $F_{<:}$.

$$\begin{aligned} \mathit{to}_{\mathsf{S}_2}\ (\mathit{from}_{\mathsf{S}_2}\ t) &= t && (1)\\ \mathit{from}_{\mathsf{S}_2}\ (\mathit{to}_{\mathsf{S}_2}\ t) &= t && (2)\\ \mathit{from}_{\mathsf{S}_2}\ ([k \to u]_T\, t) &= [k \to (\mathit{from}_{\mathsf{S}_2}\ u)]\,(\mathit{from}_{\mathsf{S}_2}\ t) && (3) \end{aligned}$$

Figure 4: Isomorphism and adequacy laws.

Special tactics: Templates alone are not enough to avoid dealing with DGP concepts directly. 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 – which states that typing is preserved by substitution – is a core lemma in formalization of in the solution to the POPLMark challenge by Aydemir et al. (2008).

Lemma *typing_subst* : $\forall E\ F\ U\ t\ T\ z\ u,$
$(E \mathbin{++} (z, U) :: F) \vdash t : T \Rightarrow F \vdash u : U \Rightarrow$
$(E \mathbin{++} F) \vdash ([z \to u]_T\ t) : T.$
Proof.
intros; *dependent induction H*; $\underline{\textit{gsimpl}}$.
...
$\underline{\textit{grewrite}}$ *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 ($\underline{\textit{gsimpl}}$ and $\underline{\textit{grewrite}}$) are used; and 2) the lemma *tbfsubst_permutation_var_wf* and the operation $[\cdot \to \cdot]_T\ \cdot$ 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 $[\cdot \to \cdot]_T\ \cdot$ would be unfolded and he would be presented with parts of the definition of $[\cdot \to \cdot]\ \cdot$ (see Figure 8). This is a generic function defined in terms of GMeta's universe. 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 define some Coq tactics (such as $\underline{\textit{gsimpl}}$ and $\underline{\textit{grewrite}}$) that *specialize* operations and lemmas such as $[\cdot \to \cdot]_T\ \cdot$ and *tbfsubst_permutation_var_wf* using the isomorphisms provided by the user, and the isomorphism and adequacy laws shown in Figure 4.

Different use modes GMeta can be used in different use modes, depending on how much knowledge a user has and what the user wishes to accomplish with GMeta. The most common scenario of use of GMeta is to formalize metatheory using the *existing* libraries provided by GMeta. In this scenario users can use GMeta in two modes: basic or advanced. The distinction between this two modes essentially depends on which types of infrastructure can be dealt by GMeta :

- **Basic user:** A basic user can simply use the isomorphism generator, instantiate templates like *LNTemplate* and use the special tactics provided by GMeta to deal with infrastructure of type A1 and A2. It may be even possible to reuse some infrastructure of type B. If some domain-specific templates exist in the libraries for the particular kind of object-language that the user is defining, then that template can be used. There is no knowledge of DGP required and no need to understand the technical details involved in GMeta 's libraries to use this mode.

- **Advanced user:** It is also possible to use GMeta to deal with infrastructure of type B provided some additional knowledge about DGP and understanding of how templates work.

Finally, GMeta can also be used by *library writers* who wish to develop their own libraries of infrastructure:

- **Library writer:** A library writer is someone that wishes to use GMeta to develop his own set of infrastructure (for example writing infrastructure for the locally named approach or for dealing with languages which support dynamic binding). The big advantage of GMeta in comparison with generative approaches such as LNGen is that, provided some basic knowledge about DGP, developing libraries of infrastructure is easy.

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

Functional languages like ML or Haskell support datatype declarations for simple inductive structures such as the natural numbers:

$\textsc{Data}\ \mathsf{Nat} = \mathsf{z} \mid \mathsf{s}\ \mathsf{Nat}$

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. In general, such notation is of the form:

$$\textsc{Data}\ \textit{type-constructor-sig}\ \textsc{where} \qquad \textit{data-constructor-sigs}$$

Using this notation, the definition of natural numbers is:

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

Essentially the same information as the conventional datatype definition is described (albeit in a more detailed form). Both the type and data constructors are written using $\mathsf{sans\ serif}$. Signatures use natural deduction rules, with the context with all the constructor arguments and their types above the line, and the type of the fully applied constructor below the line. Finally, note that the notation $\star$ (in $\mathsf{Nat} : \star$) means the 'type' (or kind) of types.

The expressiveness of this notation reveals itself when we begin defining whole families of datatypes. For example we can define a family of vectors of size n as follows:

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

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

$$\textsc{Data}\ \frac{r, s : \mathsf{Rep}}{[\![s]\!]_r : \star}\ \textsc{where} \qquad \frac{}{() : [\![\mathsf{1}]\!]_r} \qquad \frac{s : \mathsf{Rep} \qquad v : [\![s]\!]}{\mathsf{k}\ v : [\![\mathsf{K}\ s]\!]_r}$$

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

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

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

Figure 5: A simple universe of types.

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. The type of the elements A is *parametric*; that is, it is a (globally visible) parameter of all the constructors (both type and data constructors). In contrast, the type n for the size of vectors varies for each constructor; it is z for the vz constructor and $\mathsf{s}\ n$ for the vs constructor. We write parametric type arguments in type constructors such as Vector_A using 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.

A Simple Universe The universe that underlies GMeta can be viewed as a simplified version of the universe for regular tree types by Morris et al. (2004). Essentially, Morris et al.'s universe is expressive enough to represent recursive types using μ-types (Pierce 2002). However, instead of the nominal approach traditionally used with recursive type binders, the universe of regular tree types uses a well-scoped de Bruijn indices representation (Altenkirch and Reus 1999; McBride and McKinna 2004). As a consequence the presentation of that universe is somehow complicated by the use of telescopes needed in a well-scoped de Bruijn indices representation.

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 reduces the expressive power of the universe (in particular, no mutually inductive definitions are allowed), but it also avoids the use of telescopes and leads to a simpler presentation of the universe.

The complete universe of regular tree types by Morris et al. (which includes the ability to represent mutually-inductive definitions), is used in our experimental versions of GMeta (both in Coq and in Agda) available in the online web page.

Figure 5 shows our simple universe. 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 first three 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:

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

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

Compared with the more familiar definitions of recursive types for naturals and lists using μ-types (Pierce 2002)

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

we can see that the main difference is that there is no μ binder. This is because, as mentioned earlier, since we can only express a single-top level binders, a single variable (R) is enough and there is no need to have a separate binding construct.

The interpretation of the universe is given by two mutually inductive families $[\![\cdot]\!]_r$ and $[\![\cdot]\!]$, while the data constructors of these two families provide the syntax to build terms of that universe. The parametric type[3] r in the subscript in $[\![\cdot]\!]_r$, 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:

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

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

When interpreting $[\![\mathsf{RList}]\!]$, the representation type r in $[\![\cdot]\!]_r$ 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 $[\![r]\!]_r$ into a value of inductive family $[\![r]\!]$, 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.

As a final remark, note that, in this universe, in and $[\![\cdot]\!]$ are redundant because the occurrences of $[\![\cdot]\!]$ in $[\![\cdot]\!]_r$, in the data constructors r and k, could have been replaced, respectively, by $[\![r]\!]_r$ and $[\![\mathsf{R}]\!]_s$. However, the stratification into two inductive families plays an important role in Section 4.3, when the universe is extended with support for variables.

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:

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

[3]Recall that, as explained in Section 4.1, parametric types are visible in both the type and data constructors.

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

$Q : \star$ (* Quantifier type *)
$V : \star$ (* Variable type *)

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

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

Figure 6: Extending universe with representations of binders and variables.

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

To define such generic function, two-mutually inductive definitions are needed: one inductively defined on $[\![r]\!]$; and another inductively defined on $[\![s]\!]_r$. For convenience the same name *size* is used in both definitions. Note that r and s (bound by $\forall$) are implicitly passed in the calls to *size*.

When interpreted on values of type $[\![\mathsf{RNat}]\!]$, *size* computes the value of the represented natural number. When interpreted on values of type $[\![\mathsf{RList}]\!]$, *size* computes the length of the list. What is great about this function is that it works, not only for these types, but for any datatypes representable in the universe.

4.3 A Universe for Representing First-Order Binding

In this paper our goal is to define common infrastructure definitions and lemmas for first-order representations, once and for all, using generic functions and lemmas. However, the universe presented in Figure 5 is insufficient for this purpose because generic functions such as substitution and free variables require structural information about binders and variables. Therefore, we first need to enrich our universe to support these constructs.

Extended Universe Figure 6 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 quantifiers and variables. Depending on the particular first-order representations of binders these types will be instantiated differently. The following table shows the instantiations of Q and V for 4 of the most popular first-order representations:

```
RLambda : Rep
RLambda = R × R + B R R

fvar : ℕ → ⟦RLambda⟧
fvar n      = var (inl n)

bvar : ℕ → ⟦RLambda⟧
bvar n      = var (inr n)

app : ⟦RLambda⟧ → ⟦RLambda⟧ → ⟦RLambda⟧
app e₁ e₂ = in (i₁ (r  e₁, r  e₂))

lam : ⟦RLambda⟧ → ⟦RLambda⟧
lam e       = in (i₂ (λ_R 𝟙. r  e))
```

Figure 7: The untyped lambda calculus using the locally nameless approach.

	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 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 (alternatively a string could be used instead). 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: a natural number. De Bruijn indices do not need to hold information about variables in the binders, because the variables are denoted positionally with respect to the current enclosing binder. Thus, in the de Bruijn style, 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. Like de Bruijn, no information is needed at the binders, thus the type Q is just the unit type. The difference to the de Bruijn style is that parameters and (bound) variables are distinguished: (bound) variables are represented in the same way as de Bruijn variables; but parameters belong to another sort of variables. 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. However, bound variables are represented as in the nominal style instead. Thus the type Q is a natural number and the type V is a sum type of two naturals.

The inductive family $[\![\cdot]\!]_r$ 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.

Untyped lambda calculus using locally nameless representations As a simple example demonstrating the use of the universe to represent languages with binding constructs, we show how the untyped lambda calculus is encoded in Figure 7. The definition RLambda is the representation type for the untyped lambda calculus. Note that the variable case is automatically built in, so the representation only needs to account for the application and abstraction cases. Application consists of a constructor with two recursive arguments and it is represented by the product type on the left side of the sum. Abstraction is a binder with a recursive argument and is represented by the right side of the sum. The four definitions fvar, bvar, app and lam provide shorthands the corresponding parameters, variables, application and abstraction constructors. Terms can be built using these constructors. For example, the identity function is defined as:

$id_{\mathsf{RLambda}} : [\![\mathsf{RLambda}]\!]$
$id_{\mathsf{RLambda}} = \mathsf{lam}\ (\mathsf{bvar}\ 0)$

5 Modular Parameterization on Representations

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 once and for all for the languages representable in the universe for two of the most important first-order representations: locally nameless and de Bruijn.

5.1 Locally Nameless

Figure 8 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[4] and a sum of two naturals. Using these instantiations for Q and V, the *set of parameters* and *substitution* operations can be defined *generically* for the locally nameless approach. Furthermore the operation for instantiating a (bound) variable with a term is also defined in a generic way. Finally, generic lemmas can be defined using the generic operations. The statements for *subst_fresh* – which states that if a parameter does not occur in a term, then substitution of that parameter is the identity – and *bfsubst_perm* – which states that substitutions for parameters and variables can be exchanged under certain well-formedness conditions – are shown as examples 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 $[\![\cdot]\!]_r$ (mutually-)inductive families. Note that, for convenience, we use same function names for mutual definitions.

The operation fv_{r_1} computes the set of parameters in a term. The only interesting case happens with parameters:

$fv_{r_1}\ (\mathsf{var}\ (\mathsf{inl}\ x)) = \textbf{if}\ r_1 \equiv r_2\ \textbf{then}\ \{x\}\ \textbf{else}\ \emptyset$

In this case a singleton set containing the parameter is returned only when the type representations r_1 and r_2 are the same. That is, the computation of parameter sets depends on the representation r_1. For example, in System F, if r_1 is the type representation for System F types, then fv_{r_1} computes the set of type parameters which occur in a term or a type (depending on what r_2 represents). Note that the constructor inl arises from the instantiation of $V = \mathbb{N} + \mathbb{N}$. This constructor signals that the case under consideration is the left case of the sum type, which represents parameters. Variables, on the other hand, are represented by the right case of the sum type, which is signaled by the inr constructor. The other cases of fv_{r_1} are straightforward.

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

Instantiation of Q and V:

$Q = \mathbb{1}$
$V = \mathbb{N} + \mathbb{N}$

Heterogeneous sets of parameters (free variable): Given $r_1 : \mathsf{Rep}$,

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

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

Heterogeneous substitution for parameters:

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

$$\begin{aligned}
&[\cdot\ \to\ \cdot]\ \ \cdot : \forall(r_1, r_2, s : \mathsf{Rep}).\ \mathbb{N} \to [\![r_1]\!] \to [\![s]\!]_{r_2} \to [\![s]\!]_{r_2}\\
&[k\ \to\ u]\ () &&= ()\\
&[k\ \to\ u]\ (\mathsf{k}\ t) &&= \mathsf{k}\ t\\
&[k\ \to\ u]\ (\mathsf{e}\ t) &&= \mathsf{e}\ ([k\ \to\ u]\ t)\\
&[k\ \to\ u]\ (\mathsf{i}_1\ t) &&= \mathsf{i}_1\ ([k\ \to\ u]\ t)\\
&[k\ \to\ u]\ (\mathsf{i}_2\ t) &&= \mathsf{i}_2\ ([k\ \to\ u]\ t)\\
&[k\ \to\ u]\ (t, v) &&= ([k\ \to\ u]\ t, [k\ \to\ u]\ v)\\
&[k\ \to\ u]\ (\lambda_{r_3}\mathbb{1}.z) &&= \lambda_{r_3}\mathbb{1}.([k\ \to\ u]\ z)\\
&[k\ \to\ u]\ (\mathsf{r}\ \ t) &&= \mathsf{r}\ \ ([k\ \to\ u]\ t)
\end{aligned}$$

Heterogeneous substitution for (bound) variables:

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

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

Some heterogeneous lemmas:

$\mathit{subst\_fresh} : \forall(r_1, r_2 : \mathsf{Rep})\ (t : [\![r_1]\!])\ (u : [\![r_2]\!])\ (m : \mathbb{N}).$
$\quad m \notin (fv_{r_2}\ t) \quad \Rightarrow \quad [m \to u]\ t = t$

$\mathit{bfsubst\_perm} : \forall(r_1, r_2, r_3 : \mathsf{Rep})\ (t : [\![r_1]\!])\ (u : [\![r_2]\!])\ (v : [\![r_3]\!])\ (m\ k : \mathbb{N}).$
$\quad (wf_{r_3}\ u) \quad \Rightarrow \quad \{k \to ([m \to u]\ v)\}\ ([m \to u]\ t) = [m \to u]\ (\{k \to v\}\ t)$

Figure 8: Generic definitions for the locally nameless approach.

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

Heterogeneous shifting: Given $r_1 : \mathsf{Rep}$,

$$
\begin{array}{ll}
\uparrow_{\cdot} \cdot : \forall (r_2 : \mathsf{Rep}).\ \mathbb{N} \to [\![r_2]\!] \to [\![r_2]\!] & \\
\uparrow_m (\mathsf{in}\ t) & = \mathsf{in}\ (\uparrow_m t) \\
\uparrow_m (\mathsf{var}\ n) & = \mathbf{if}\ r_1 \equiv r_2 \wedge m \leqslant n\ \mathbf{then}\ (\mathsf{var}\ (n+1))\ \mathbf{else}\ (\mathsf{var}\ n)
\end{array}
$$

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

Figure 9: Heterogeneous shifting for de Bruijn representations.

In the generic definitions for substitutions[5] the 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. The binder case in the heterogeneous substitution for variables is more interesting. 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 shifting 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) variables, the type V is instantiated with natural numbers. For space reasons we only show how to define one operation, *shifting*, in Figure 9. Other generic operations like substitution are similar to the locally nameless style, and generic lemmas about operations on de Bruijn representations are also definable generically. The implementation of heterogeneous generic shifting is quite simple and 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

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

		Definitions	Infrastructure	Core	Overall		
			(lemmas + definitions)	(lemmas + definitions)	inf. overhead	total	ratio
STLC (locally nameless)	Aydemir et al.	**11**	**13** + 3	**4** + 0	17	31	55%
	GMeta basic	7	4 + 0	4 + 0	1	15	7%
System $F_{<:}$ (locally nameless)	Aydemir et al.	**20**	**48** + 7	**17** + 1	60	93	65%
	GMeta basic	13	26 + 1	17 + 1	25	58	43%
	GMeta advanced	11	15 + 0	18 + 1	11	45	24%
System $F_{<:}$ (de Bruijn)	Vouillon	27	24 + 0	50 + 0	41	101	41%
	GMeta basic	12	1 + 0	52 + 0	3	65	5%

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

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

Figure 11: Savings in various formalizations in terms of numbers of definitions and lemmas.

are limited to traversal code. Note that we follow Vouillon (2007)'s definitions.

6 Discussion and Evaluation

In this section we present the results of the case studies we conducted. The discussion of these results is done in terms of three criteria proposed by Aydemir et al. (2005) (*reasonable overheads*, *cost of entry* and *transparency*) for evaluating mechanizations of formal metatheory. We also discuss how theorem provers like Coq could be improved with better support for GMeta.

6.1 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 11 shows the savings that GMeta achieved relative to the reference solutions by Aydemir et al. (2008). In all case studies more than 35% of the total numbers of definitions were saved. We conducted case studies in both System $F_{<:}$ (using locally nameless and de Bruijn representations) and STLC (using locally nameless representations). We also employed GMeta in two modes, basic and advanced (see Section 3.2 for the explanation). In the STLC case there was no need to employ the advanced mode, since with the basic approach reuse was already as good as it could be. In System $F_{<:}$ we need some lemmas of type B3, which cannot be dealt using just the basic approach, so we also used the advanced approach. In the de Bruijn case study, only the basic approach was employed.

Detailed results 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 infrastructure lemmas and definitions are. The column inf. overhead counts the overall number of infrastructure definitions and lemmas used across the formalizations. Although, for the most part, infrastructure comes from what was classified by Aydemir et al. as the infrastructure part, some additional infrastructure definitions and lemmas 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 infrastructure, our numbers reflect the total number of definitions and lemmas in each part.

In same cases we needed a few lemmas simply to slightly adapt the form of a lemma used in GMeta libraries to the form used in (Aydemir et al. 2008). This explains, for example the 1 in the inf. overhead column for STLC.

Basic vs Advanced The main difference between the basic and advanced approaches is that the basic approach allows all types of infrastructure to be reused except for infrastructure of type B3, whereas in the advanced approach all types of infrastructure can be reused. To capture some of the B1 and B2 lemmas used in the basic mode for System $F_{<:}$ we used a domain-specific template for typed languages involve at least 2 syntactic sorts (note that this is a fairly common type of object language).

Proof size In comparison with Aydemir et al. reference solutions, the proofs in the basic approach follow essentially the same 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 (see Section 3.2). Because this is the only difference, the proofs in the GMeta solution and Aydemir et al. solution have comparable sizes. In the advanced approach one additional difference is that the formalization of environments differs from Aydemir et al., in that we use two environments (for types and terms) and Aydemir et al. use a single environment with a disjoint union. This difference means that, while most proofs will still be comparable in size, a small number of proofs will be either shorter or larger.

6.2 Cost of entry

One important criteria 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) (at least for the basic approach).

Essentially GMeta has a pay-as-you-go approach in terms of the cost-of-entry. The basic approach gives us a good cost/benefit ratio. Basically, with developments using the basic mode we get quite a bit of reuse almost for free. reference solution. In the advanced approach additional savings are possible, at the cost of some extra knowledge about DGP and GMeta.

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.

6.3 Transparency

The transparency criteria is intended to evaluate how intuitive a formalization technique is. 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).

6.4 Improving Coq and Other Theorem Provers

We found a few shortcomings in Coq during our experience with GMeta. These affected the development of GMeta and also it's usability. It would be nice to have the following improvements to address those shortcomings.

Improved support for dependently typed programming GMeta makes significant use of dependently typed programming. Being based on the calculus of inductive constructions, Coq has good support for dependent types. However, only very recently more attention has been given to providing better support for dependent pattern matching (Coquand 1992) as done, for example, in the Agda or Epigram dependently typed languages. In this respect the work by Sozeau (2007) on the Program tactic and the other related tactics like *dependent destruction* is certainly a good step forward. Unfortunately, support for these features is still experimental on the current version of Coq (8.3) and there is still some gap to dependently typed languages like Agda or Epigram.

Our experience with GMeta is that there are still a few bugs lying around in the current implementation of Coq which make the development of dependently typed programs that use dependent pattern matching difficult. We found *nested* dependent pattern matching and the use mutual induction with *dependent destruction* to be particularly tricky in Coq. Curiously we only need these features for defining isomorphisms, which (technically speaking) are a fairly uninteresting part of GMeta (since usually the definition of isomorphisms in DGP is supposed to be straightforward and mechanical). More concretely, the need for nested dependent pattern matching arises from the *to* function (which converts from a representation of the object language in the universe to the user defined object language) and the *to_from* lemma (which states that $to\ (from\ t) = t$).

With the very valuable help of Chlipala upcoming book on "Certified Programming with Dependent Types" (Chlipala 2009) we did manage to learn enough techniques to tame Coq and make our programs and proofs pass the type and termination checkers. However in our experimental version of GMeta which is based on the more expressive universe for regular tree types by (Morris et al. 2004) and it requires even more dependent types, we found that the techniques we used earlier no longer worked and we are still looking for a way to implement isomorphisms (although the remaining parts of the framework work as expected).

In contrast, our experience with developing a version of GMeta in Agda was much smoother due to the good support for dependent pattern matching. In Agda, the definitions of isomorphisms were essentially painless. Of course, the main reason not to use Agda in the first place is that the main purpose of GMeta is to assist with theorem proving of mechanized metatheory and Coq support for theorem proving is generally better than Agda, which is focused primarily on programming.

To make a long story short, we believe that support for dependent pattern matching in Coq could be improved in two ways. The first way is simply to polish the current implementation to make it bug free and well supported. We expect that in the next few versions of Coq this will happen. The second improvement would be to provide better assistance when developing programs using dependent pattern matching. The emacs mode for Agda and Epigram are good examples of how the "IDE" can provide valuable help to the user.

Partial evaluation of modules One disadvantage of using DGP for reusing infrastructure is that such infrastructure is written generically. When trying to prove some lemmas users may have to unfold such generic definitions. The way that GMeta currently deals with this problem is through the use of some special tactics like *gsimpl* or *grewrite* (as discussed in Section 3), which make use of isomorphisms and their laws to automatically specialize definitions into the form that the user would expect. However this approach is not ideal because it requires the user to know about the special tactics and when to use them, so a nicer alternative is desirable.

We believe some kind of module-level partial evaluation would be a better solution for specializing infrastructure for object languages and eliminating the DGP indirections. Ideally when instantiating templates like *LNTemplate* with particular isomorphisms we would like to partially evaluate the resulting module so that all the isomorphism indirections would be evaluated and removed. The work by Brady and Hammond (2010) on partial evaluation for dependently typed languages and the work by Alimarine and Smetsers (2004) on partial evaluation of generic functions should be good starting points to investigate this possibility further.

In essence, with module-level partial evaluation we would get the best of DGP and generative approaches: verified generic infrastructure that can be specialized to a form similar to hand-written definitions.

Built-in support for isomorphisms Finally, it would be better to support isomorphism generation directly in Coq, instead of using of an external tool. This could be done for example via a plugin. The idea is simply to support the isomorphism annotations; and to generate and make the isomorphism available to the user directly, rather than calling an external tool that generates a Coq module that needs to be imported later. This could be supported in very much the same way that inductive principles and recursion principles are supported in Coq for inductive definitions. Basically, Coq automatically generates the associated principles when we define an inductive definition, and those principles become immediately available to the user.

7 Related Work

Generative Approaches Closest to our work are *generative* approaches like LNgen (Aydemir and Weirich 2009), 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 3.2). As we have discussed in Section 6.4, allowing some kind of module-level partial evaluation would be nicer and would allow us to recover direct definitions from the generic infrastructure.

There are two main advantages of a DGP approach over generative approaches: 1) *verifiability*; and 2) *extensibility*. Like with DGP, a generator allows defining once-and-forall the infrastructure. However, unlike DGP, we cannot verify once-and-forall 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 disadvantage of generators is that they are hard to extend. If we wanted to add a new lemma to the set of generated infrastructure we would have to directly change the generator code. With a library approach like GMeta this is much easier because it amounts to extending a module with a new generic function.

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 inductive relations is easy. In GMeta, type 3 definitions require some more effort to be reused. A solution for this problem would be to extend the isomorphism generator tool to deal with inductive relations as well. On the other hand the strength of GMeta lies on its extensibility. It is easy to capture some types of domain-specific infrastructure, which are still common enough that we may want to capture in a reusable form. The lemma *thbfsubst_perm_core* presented in Section 2.2 is one example. Dealing with domain-specific infrastructure like that 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 indices 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 (2007) and Licata and Harper (2009) work can be viewed as trying to develop new ways to formalize metatheory, in which many of the invariants that would have to be proved otherwise hold by construction. 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; Lämmel and Peyton Jones 2003; Rodriguez et al. 2009), and Cheney (2005) explored how to provide generic operations such as substitution or 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 5 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 6 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 it's 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); Chlipala (2008) have shown how weaker variations of HOAS can be encoded in Coq. 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 package (Urban 2005). However, because Coq does not have a nominal package, 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 reveals itself useful for conducting realistic formalizations of metatheory.

Acknowledgments

This work benefited from useful comments by Hugo Herberlin, Sungwoo Park, Jaeho Shin and Stephanie Weirich.

References

A. Alimarine and S. Smetsers. Optimizing generic functions. In Dexter Kozen, editor, *Mathematics of Program Construction*, volume 3125, pages 16–31. 2004.

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

T. Altenkirch and B. Reus. Monadic presentations of lambda terms using generalized inductive types. In *CSL '99*, 1999.

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

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

B. E. Aydemir, A. Bohannon, M. Fairbairn, N. Nathan Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized Metatheory for the Masses: The POPLmark Challenge. In *TPHOLs '05*, 2005.

B. E. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *POPL '08*, 2008.

S. Boutin. Using reflection to build efficient and certified decision procedures. In *TACS'97*, 1997.

Edwin C. Brady and Kevin Hammond. Scrapping your inefficient engine: using partial evaluation to improve domain-specific language implementation. In *ICFP '10*, pages 297–308, 2010.

J. Cheney. Scrap your nameplate: (functional pearl). *SIGPLAN Not.*, 40(9):180–191, 2005.

A. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. *SIGPLAN Not.*, 42(6):54–65, 2007.

A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. *SIGPLAN Not.*, 43(9):143–156, 2008.

A. Chlipala. *Certified Programming with Dependent Types.* 2009. URL `http://adam.chlipala.net/cpdt/`.

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

T. Coquand. Pattern matching with dependent types. In B. Nordström, K. Pettersson, and G. Plotkin, editors, *Informal Proceedings Workshop on Types for Proofs and Programs, Båstad, Sweden, 8–12 June 1992*, pages 71–84. 1992.

N. G. de Bruijn. 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.

J. Despeyroux, A. P. Felty, and A. Hirschowitz. Higher-order abstract syntax in coq. In *TLCA '95*, 1995.

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

P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. In *TLCA '99*, 1999.

P. Dybjer and A. Setzer. Indexed induction-recursion. In *PTCS '01*, 2001.

A. Gacek. The abella interactive theorem prover (system description). In *IJCAR '08*, 2008.

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

R. Hinze and J. Jeuring. Generic haskell: Practice and theory. In *Generic Programming '03*, 2003.

P. Jansson and J. Jeuring. Polyp—a polytypic programming language extension. In *POPL '97*, 1997.

R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *SIGPLAN Not.*, 38(3):26–37, 2003.

D. R. Licata and R. Harper. A universe of binding and computation. In *ICFP '09*, 2009.

P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

C. McBride and J. McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004.

J. Mckinna and R. Pollack. Pure type systems formalized. In *TLCA '93*, pages 289–305. Springer-Verlag, 1993.

A. Momigliano, A. J. Martin, and A. P. Felty. Two-level hybrid: A system for reasoning using higher-order abstract syntax. *Electron. Notes Theor. Comput. Sci.*, 196:85–93, 2008.

P. Morris, T. Altenkirch, and C. McBride. Exploring the regular tree types. In *TYPES '04*, 2004.

P. Morris, T. Altenkirch, and N. Ghani. A universe of strictly positive families. *Int. J. Found. Comput. Sci.*, 20(1):83–107, 2009.

B. Nordström, K. Peterson, and J. M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford Unversity Press, 1990.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

U. Norell. Dependently typed programming in agda. In *Advanced Functional Programming '08*, 2008.

C. Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supérieur*. Habilitation à diriger les recherches, Université Claude Bernard Lyon I, 1996.

S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for gadts. *SIGPLAN Not.*, 41(9):50–61, 2006.

F. Pfenning and C. Elliot. Higher-order abstract syntax. *SIGPLAN Not.*, 23(7):199–208, 1988.

F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In *CADE '99*, 1999.

B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

A. M. Pitts. Nominal logic, a first order theory of names and binding. *Inf. Comput.*, 186(2): 165–193, 2003.

A. Rodriguez, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. Comparing libraries for generic programming in haskell. *SIGPLAN Not.*, 44(2):111–122, 2009.

A. Rossberg, C. V. Russo, and D. Dreyer. F-ing modules. In *TLDI '10*, 2010.

P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20(01):71–122, 2010.

M. Sozeau. Program-ing finger trees in coq. In *ICFP 2007*, 2007.

C. Urban. Nominal techniques in isabelle/hol. In *CADE '05*, pages 38–53, 2005.

W. Verbruggen, E. de Vries, and A. Hughes. Polytypic programming in coq. In *WGP '08*, 2008.

W. Verbruggen, E. de Vries, and A. Hughes. Polytypic properties and proofs in coq. In *WGP '09*, 2009.

J. Vouillon. Poplmark solutions using de bruijn indices, 2007. Available at `http://alliance.seas.upenn.edu/~plclub/cgi-bin/poplmark/index.php?title=Submission_by_J%C3%A9r%C3%B4me_Vouillon`.