A Generic Formal Metatheory Framework for First-Order Representations

Gyesik Lee, Bruno C. d. S. Oliveira, Sungkeun Cho and Kwangkeun Yi
ROSAEC Center, Seoul National University
{gslee,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* much of the so-called infrastructure lemmas and definitions required in mechanizations of formal metatheory. The framework employs *datatype-generic programming* and *modular programming* techniques to provide a universe representing a family of datatypes. This universe is generic in two different ways: it is *language*-generic in the sense that several object languages can be represented within the universe; and it is *representation*-generic, meaning that it is parameterizable over the particular choice of first-order representations for binders (for example, *locally nameless* or *de Bruijn*). Using this universe, several libraries providing generic infrastructure lemmas and definitions are implemented. These libraries are used in case studies based on the POPLmark challenge, showing 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, ensuring the soundness of that infrastructure. Furthermore, due to GMETA's modular design, the libraries can be easily used, extended and customized by end 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).

The main advantage of first-order approaches, and the reason why they are so popular in practice, is that terms with binders are easy to manipulate and understand; and they work well in general-purpose theorem provers like Coq (Coq Development Team 2009).

However, the main drawback of first-order approaches is that, without special support, the tedious infrastructure required for handling variable binding has to be repeated each time for a new object language. For example, one of the most popular and state-of-the art first-order approaches for formalizing metatheory consists of a completely manual scheme using a locally nameless style (Aydemir et al. 2008). The key idea is that a number of guidelines can be followed for obtaining the implementation of the basic infrastructure lemmas and definitions. The problem is that, in many cases, the majority of the total number of lemmas and definitions in a formalization consists of basic infrastructure. One example is the solution to the POPLmark challenge (Aydemir et al. 2005) parts 1A+2A by Aydemir et al. (2008). In that solution the number of lemmas and definitions required by the basic infrastructure corresponded to roughly 65% of the development (see also Figure 11). Another example is the development of the metatheory for a type-directed translation from an ML-module language to System F_ω (Rossberg et al. 2010). In that case, as the authors note, "*Out of a total of around 550 lemmas, approximately 400 were tedious infrastructure lemmas*". This lead Rossberg et al. not to recommend Aydemir et al. (2008) locally nameless approach for anything but small, kernel language developments.

This paper proposes GMETA: a generic framework for first-order representations of variable binding, implemented in the Coq theorem prover, which provides *once and for all* much of the infrastructure boilerplate (Lämmel and Peyton Jones 2003) required in mechanized formal metatheory. The key idea is to employ *generic programming* techniques to provide a *universe* representing a family of datatypes. This universe is generic in two different ways:

1. *Language-generic*: several object languages can be represented in the universe. This form of genericity is based on *datatype-generic programming* (DGP) (Gibbons 2007; Jansson and Jeuring 1997).
2. *Representation-generic*: the particular choice of first-order representations (for example, *locally nameless* or *de Bruijn* representations) is parameterizable. This form of generic programming is based on *modular programming* in the style of ML-modules (MacQueen 1988).

With GMETA, developing mechanized metatheory does not involve implementing much of the tedious infrastructure boilerplate by hand for a new language. Instead, such infrastructure can be reused directly from GMETA's libraries. All that is required is an *isomorphism* between the object language and a particular instantiation of the generic language provided by GMETA. Since such an isomorphism can be mechanically generated from the inductive definition of the object language (provided a few annotations) GMETA includes optional tool support that is capable of generating such isomorphisms automatically. 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.

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. Generative approaches have similar benefits to GMETA in terms of savings of infrastructure, and one advantage 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 less convenient than a generative approach if the user has to interact with the generic parts of the libraries. GMETA recovers most of the convenience of generative approaches by providing a set of templates and a few tactics, which allow users to import infrastructure and use it just as if it was specially produced for the object language in hand, avoiding direct interaction with the generic parts of the libraries.

GMETA has some important advantages when compared to generative approaches. Firstly, all the generic infrastructure is implemented inside the meta-logic (Coq) itself, which guarantees *soundness*. In a generative approach it is hard to ensure that the produced code is always sound (LNgen does not offer such guarantees). Secondly, since GMETA is a library-based approach built on top of ML-style modules, the whole infrastructure can be easily extended and customized. In contrast, in a generative approach extending and customizing the generated infrastructure involves modifying the generator, which is likely to be non-trivial (except for the authors of the tool itself). Thirdly, from a theoretical perspective, a generic programming approach allows generic functions and lemmas to be understood as simple, clear specifications of the infrastructure independently of particular object languages. Finally, GMETA works for multiple first-order representations, whereas LNgen only currently supports locally nameless representations.

In order to validate the effectiveness of GMETA in reducing the amount of infrastructure overhead, we conducted several case studies based on the POPLmark challenge. These case studies show that GMETA's generic infrastructure is sufficient to deal with languages with challenging binding constructs like System $F_{<:}$, and that the majority of the boilerplate infrastructure (58%-94%) can be saved.

The contributions of this paper are:

- *Sound, generic, reusable and extensible infrastructure for first-order representations*: The main novelty of our work is the application of generic programming techniques to provide generic infrastructure for first-order representations, allowing such infrastructure to be reused across multiple formalizations. Our universe-based approach has strong theoretical foundations and can be expressed in type-theories with support for *inductive families* (Dybjer 1997).
- *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[1] along with other resources such as tutorials and more case studies.

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

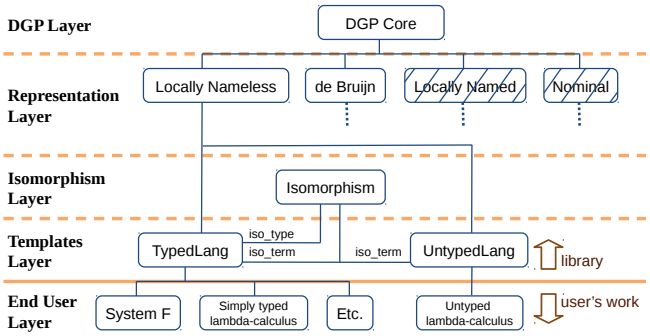

Figure 1. A simplified modular structure overview of GMETA.

2. GMETA in Practice

This section presents an overview of the modular structure of GMETA, shows how GMETA is used to conduct formalizations and summarizes the results of our case studies.

2.1 GMETA's Modular Structure and Paper Organization

Figure 1 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. The paper organization follows the 5 layers in the modular structure of GMETA:

- **DGP Layer:** The core DGP infrastructure is defined at the topmost 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. (Section 3)
- **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. (Section 4)
- **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. (Section 5)
- **Templates Layer:** This layer provides templates for the basic infrastructure lemmas and definitions required by particular meta-theoretical developments. Templates are ML-style functors parametrized by isomorphisms between the syntactic sorts of object language and their corresponding representations in the generic language. Examples of templates are *UntypedLang* (for untyped languages with a single syntactic sort) or *TypedLang* (for typed languages with two syntactic sorts: types and terms). (Section 6)
- **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. (Section 7)

2.2 Using GMETA

Four basic steps are necessary to use GMETA. These are discussed and illustrated next using the STLC as example.

1. **Defining Syntax:** The first step is to define the syntax for the object language that is going to be formalized, using one of the first-order representations supported by GMETA. Figure 2

```
(*@Iso typ_iso*)
Inductive typ :=
| typ_var   : ℕ → typ
| typ_arrow : typ → typ → typ.
(*@Iso trm_iso{
  Parameter trm_fvar,
  Variable  trm_bvar,
  Binder    trm_abs
}*)
Inductive trm :=
| trm_bvar : ℕ → trm
| trm_fvar : ℕ → trm
| trm_abs  : trm → trm
| trm_app  : trm → trm → trm.
```

Figure 2. Syntax definitions and GMETA isomorphism annotations for the STLC in Coq.

illustrates how this can be done in Coq using locally nameless representations.

2. **Defining isomorphisms for interfacing with GMETA:** The second step is to provide annotations to guide the isomorphism generation tool to produce the isomorphism modules automatically. An example of such annotations can be seen in Figure 2. 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[2] or binders. In the case of types, there is no binding structure, so we do not need to specify such information. Note that, alternatively, the isomorphism modules can also be directly implemented by hand.

3. **Importing the infrastructure:** After running the generator tool on the file with the syntax definitions, we can use the generated isomorphism modules to instantiate some of GMETA infrastructure templates, and import these to obtain the desired infrastructure lemmas and definitions. In the case of STLC, this is as simple as adding the following line to the file where the main formalization is going to be implemented:

   Module Import $M := TypedLang\ trm\_iso\ typ\_iso.$

4. **Metatheory formalization:** Now the user can proceed with the formalization using the imported infrastructure. For example, the following lemma – which states that typing is preserved by substitution – is a core lemma in the formalization of the STLC.

   Lemma $typing\_subst : \forall E\ F\ U\ t\ T\ z\ u,$
   $(E \text{ ++ } (z, U)) :: F) \vdash t : T \Rightarrow F \vdash u : U \Rightarrow$
   $(E \text{ ++ } F) \vdash ([z \rightarrow u]\ t) : T.$
   Proof.
   $intros; dependent\ induction\ H; \underline{gsimpl}.$
   ...
   $\underline{grewrite}\ tbfsubst\_permutation\_var\_wf; eauto.$
   ...
   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 (*gsimpl* and *grewrite*) are used; and 2) the lemma *tbfsubst_permutation_var_wf* is provided by GMETA's templates. Essentially, for basic uses of GMETA, the end-user only needs to know a few tactics to be used instead of Coq's own standard tactics; and be familiar with the lemmas provided by the templates and how they are used in typical locally nameless formalizations.

[2] Note that throughout this paper we will follow Aydemir et al. (2008) and use the term *variable* to mean a locally bound variable; and *parameter* to mean a free, or globally bound variable.

		Savings	
		boilerplate	total
STLC	GMETA basic vs Aydemir et al.	59%	32%
	GMETA adv. vs Aydemir et al.	94%	52%
$F_{<:}$	GMETA basic vs Aydemir et al.	58%	38%
	GMETA adv. vs Aydemir et al.	82%	52%

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

Just by following these basic steps a lot of infrastructure can be readily made available to the end user. With some extra knowledge about GMETA and DGP it is possible to obtain additional reuse and extend the GMETA infrastructure that is provided by the libraries.

2.3 Case Studies and Summary of Results

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 only discuss the locally nameless case studies in this paper. The results for 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.

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

1. **Common operations:** operations such as free 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).
2. **Common lemmas about operations:** lemmas about different properties of the operations in 1), such as several forms of permutation lemmas about substitutions.
3. **Lemmas involving well-formedness and common operations:** 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.

For each case study, GMETA was employed in two different ways, according to how much the user needs to know about GMETA and DGP in order to successfully achieve reuse:

- **Basic approach:** In the basic approach, the end-user is required to know about the interfaces of GMETA's templates and a few simple tactics that automatically simplify the templates definitions into a form similar to the hand-written versions of those definitions. Occasionally the user may also need to apply the adequacy lemmas (see Section 5.2) directly. However, the user does not need to know about DGP in order to use GMETA.
- **Advanced approach:** In the advanced approach, it is possible to obtain additional reuse by manually providing a mapping between concrete definitions of well-formed environments and corresponding generic notions defined in the GMETA libraries. This requires some knowledge about DGP and some more tactics in GMETA.

Results The case studies measure the number of definitions and lemmas that have been saved (both the total number and the number of boilerplate definitions and lemmas). The results are shown in Figure 3 and presented in terms of the percentage of definitions and lemmas saved when compared to the reference solutions by Aydemir et al. (2008). In all case studies, the solution using GMETA managed to save more than half of the number of boilerplate definitions and lemmas. The maximum reuse was obtained with the advanced approach for STLC (94%). In terms of the number of total definitions and lemmas, the savings were never below 32% and in the case of System $F_{<:}$ the savings were 52% when using the advanced approach. The detailed numbers and an evaluation of GMETA are presented in Section 7.

3. 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).

3.1 Inductive Families

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

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

Nat is the type of natural numbers, the constructor z represents 0 and the constructor s gives us the successor of a natural.

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. Inductive families are one of the inspirations for Generalized Algebraic Datatypes (GADTs) (Peyton Jones et al. 2006), which has been adopted by Haskell and other languages.

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 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:

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

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

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. 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.

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

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

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

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

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

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

Figure 4. A simple universe of types.

3.2 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 Figure 4 shows a simple universe that is the basis for GMETA's 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 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:

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

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

It is useful to compare these definitions with the more familiar definitions for recursive types using μ-types (Pierce 2002):

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

The definitions RNat and RList are similar to Nat and List. The main difference is that, in the later two definitions, type-level fixpoints are used. Our grammar of types Rep can be seen as a simplified way to describe recursive types, where instead of the full power of type-level fixpoints, we have a single label R representing an occurrence of the recursive type.

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 (the label for the recursive type). 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 $[\![r]\!]$. 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 3.3, 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:

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

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

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.

3.3 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 4 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.

[3] Recall that, as explained in Section 3.1, parametric types are visible in both the type and data constructors.

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

$$Q : \star \qquad \text{(* Quantifier type *)}$$
$$V : \star \qquad \text{(* Variable type *)}$$

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

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

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

Figure 5. Extending universe with representations of binders and variables.

Extended Universe Figure 5 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. In Section 4 we will see why the distinction between K and E is necessary.

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:

	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.

```
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 (λR𝟙. r e))
```

Figure 6. The untyped lambda calculus using the locally nameless approach.

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 6. 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_RLambda : ⟦RLambda⟧
id_RLambda = lam (bvar 0)
```

4. Modular Parametrization on Representations

This section shows how generic operations and lemmas defined over the universe presented in Section 3 can be used to provide much of the basic infrastructure boilerplate once and for all for the languages representable in the universe. We start by motivating the need for *heterogeneous* operations and lemmas for supporting binding constructs in realistic languages and proceed with showing how to define the generic infrastructure for two of the most important first-order representations: locally nameless and de Bruijn.

4.1 The Need for Heterogeneous Generic Operations

Many realistic languages have binding constructs that require heterogeneous operations, which are operations like a substitution that deals with multiple syntactic sorts:

$$hsubst : \mathbb{N} \to \mathsf{Type} \to \mathsf{Term} \to \mathsf{Term}$$

In this case, $hsubst$ is an operation that substitutes a type variable in a term by some type. Operations such as $hsubst$ are needed, for example, to deal with System F type abstractions in terms ($\Lambda X.e$). In general, when the object language has many syntactic sorts, it is possible that several different kinds of heterogeneous substitutions are needed for various combinations of syntactic sorts. For instance, consider the possible substitution operations for the locally nameless style with only two syntactic sorts: types and terms. In this case, there are 8 different kinds of substitution:

		Term	Type
Term	Variables	$tbsubst$	$thbsubst$
	Parameters	$tfsubst$	$thfsubst$
Type	Variables	$yhbsubst$	$ybsubst$
	Parameters	$yhfsubst$	$yfsubst$

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, the System F family of languages will typically need 6 out of the 8 substitutions. We believe that heterogeneous operations are one of the main causes of tedious and burdensome boilerplate that is observed in larger formalizations such as the type-directed translation of ML-modules to System F_ω by Rossberg et al. (2010).

Therefore, in order to support basic infrastructure that deals with realistic languages, we should have generic functions that can be instantiated to operations such as $hsubst$. In other words we should have heterogeneous generic functions and lemmas that can be used with multiple syntactic sorts.

4.2 Locally Nameless

Figure 7 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 3.3, 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 3, 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:

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

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,

[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 $r_1 : \mathsf{Rep}$,

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

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

Heterogeneous substitution for parameters:

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

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

Heterogeneous substitution for (bound) variables:

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

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

Some heterogeneous lemmas:

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

Figure 7. Generic definitions for the locally nameless approach.

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

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

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

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

Figure 8. Heterogeneous shifting for de Bruijn representations.

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.

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 4.1 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.

4.3 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

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

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

5. Isomorphisms and Adequacy

Isomorphisms provide the interface between object languages specified by conventional inductive datatype definitions in Coq and the Coq implementation of the generic language described by the universe presented in Section 3. This section shows how tool support can automate the creation of those isomorphisms and how the adequacy between the object language and its generic language representation can be ensured.

5.1 Isomorphisms and Tool Support

In Section 2.2 we showed how to specify the syntax of an object language in Coq using conventional inductive declarations and how to provide annotations for guiding the automatic creation of isomorphisms between the object language and the generic language provided by GMETA. In this way an end-user does not need to know about DGP in order to enjoy the benefits of GMETA. The isomorphism annotations guide the creation of two different types of isomorphisms:

- **Partial isomorphisms**: This type of isomorphisms is used for syntactic sorts without binding constructs, such as for example types in STLC. The signature for modules for partial isomorphisms contains

  $$\begin{array}{ll} \mathit{from} & : T \to [\![R_T]\!] \\ \mathit{to} & : [\![R_T]\!] \to T \\ \mathit{to\_from} & : \forall (t : T), \mathit{to}\ (\mathit{from}\ t) = t \end{array}$$

  where T and R_T are two parameters of the isomorphism module corresponding to the type of some object language and its universe representation. The functions from and to are just the the two mappings that define the isomorphism and $\mathit{to\_from}$ is one of the isomorphism lemma.

- **Full isomorphisms**: This type of isomorphisms is used for syntactic entities that have binding constructs, such as for example terms in STLC or System $F_{<:}$ or types in System $F_{<:}$. The addition to partial isomorphisms is that we also need the following lemma:

  $$\mathit{from\_to} : \forall (t : [\![R_T]\!]), \mathit{from}\ (\mathit{to}\ t) = t$$

Both types of isomorphisms can be either created by hand or automatically by a prototype generator tool that comes with GMETA. The generator is similar to generators used in DGP libraries for functional languages (Rodriguez et al. 2009): it follows the inductive structure of the object language T to generate all the definitions and proofs. One difference, however, is that the isomorphism annotations are required to provide extra information about the binding structure. For example, in Figure 2, the annotation for the isomorphism $\mathit{typ\_iso}$ is used to create a partial isomorphism, whereas the annotation for $\mathit{trm\_iso}$ is used to create a full isomorphism.

Binding annotations can be more complex than those found in Figure 2. For example, some languages may have multiple binders, and these binders may bind different syntactic sorts. One example of a language where the binding structure is richer than STLC is System $F_{<:}$. In GMETA, the annotations for System $F_{<:}$ terms can be defined as follows:

$$\begin{array}{l} (*@\mathbf{Iso}\ \mathit{trm\_iso}\{ \\ \quad \mathbf{Parameter}\ \mathit{trm\_fvar}, \\ \quad \mathbf{Variable}\ \ \mathit{trm\_bvar}, \\ \quad \mathbf{Binder}\ \ \mathit{trm\_abs}\ \_, \\ \quad \mathbf{Binder}\ \ \mathit{trm\_tabs}\ \_\ \mathbf{binds}\ \mathit{typ} \\ \}*) \end{array}$$

The identifiers $\mathit{trm\_fvar}$ and $\mathit{trm\_bvar}$ are, respectively, the constructors for parameters and variables, whereas the identifiers $\mathit{trm\_abs}$ and $\mathit{trm\_tabs}$ correspond, respectively, to the constructors for term-level abstraction ($\lambda x : T.e$) and type-level abstraction in terms ($\Lambda X.e$). The annotations are richer than STLC in a few aspects. Firstly, there are multiple **Binder** annotations. Secondly, for type-level abstractions in terms, we need to specify that the bounded elements are types rather than terms. Finally, the "_" annotations specify that the body of the lambda is the second argument (and not the first) – this is because System $F_{<:}$ binders are annotated with types, which are provided as the first argument.

The isomorphism generator tool currently supports all these kinds of annotations for the locally nameless style, which is already enough to cope with interesting languages like System $F_{<:}$. However, it is likely that a richer binding language for annotations is required for additional flexibility and for other kinds of sophisticated binding structures. This and support for other first-order approaches is left for future work.

$$\begin{array}{rcll} \mathit{to}\ (\mathit{from}\ t) & = & t & (1) \\ \mathit{from}\ (\mathit{to}\ t) & = & t & (2) \\ \mathit{from}\ ([k \to u]_T\, t) & = & [k \to (\mathit{from}\ u)]\, (\mathit{from}\ t) & (3) \\ \mathit{from}\ (\{k \to u\}_T\, t) & = & \{k \to (\mathit{from}\ u)\}\, (\mathit{from}\ t) & (4) \end{array}$$

Figure 9. Adequacy laws.

5.2 Soundness and Adequacy

Soundness A key advantage of the generic programming approach taken by GMETA over generative approaches such as LNgen is that the generic infrastructure is defined and verified in Coq. In other words, unlike a generative approach, soundness is guaranteed by the meta-logic itself and no external proof obligations are needed. Note, however, that we provide no guarantees that the isomorphism generator always produces correct code for isomorphisms. In this case the situation is similar to LNgen: the tool can be used to generate code that has to be verified afterwards.

Adequacy Whenever we talk about representations of a language, it is desirable to talk about some form of adequacy of the representations (Harper et al. 1993). Because the typical use of GMETA involves two languages – an object language such as the lambda calculus presented in Figure 2; and the representation of that language in the DGP universe – the question of adequacy is also relevant.[6] Informally adequacy requires:

[6] Note that an orthogonal question concerns a more traditional view of adequacy, which is whether the pen-and-paper language is adequately represented by the object language that is used in the mechanization of the formalization. It is, of course, also desirable that the user shows this form of adequacy, but this is a proof obligation that the user has, independently of the use of GMETA.

Simple typed language template:

$tfsubst : \mathbb{N} \to \mathsf{Term} \to \mathsf{Term} \to \mathsf{Term}$
$tfsubst\ k\ u\ t = to\ ([k\ \to\ (from\ u)]\ (from\ t))$

$tbsubst : \mathbb{N} \to \mathsf{Term} \to \mathsf{Term} \to \mathsf{Term}$
$tbsubst\ k\ u\ t = to\ (\{k\ \to\ (from\ u)\}\ (from\ t))$

$twf : \mathsf{Term} \to \mathsf{Prop}$

$tbfsubst\_perm\_core : \forall(t, u, v : \mathsf{Term})\ (m\ k : \mathbb{N}),$
$twf\ u \Rightarrow$
$tbsubst\ k\ (tfsubst\ m\ u\ v)\ (tfsubst\ m\ u\ t)$
$= tfsubst\ m\ u\ (tbsubst\ k\ v\ t)$

$tfsubst\_fresh : \forall(t\ u : \mathsf{Term})\ (m : \mathbb{N}),$
$m\ \notin\ (tfv\ t) \to t = tfsubst\ m\ u\ t.$

Heterogeneous typed language template:

$thfsubst : \mathbb{N} \to \mathsf{Type} \to \mathsf{Term} \to \mathsf{Term}$
$thfsubst\ k\ u\ t = to\ ([k\ \to\ (from_{\mathsf{Type}}\ u)]\ (from\ t))$

$thbsubst : \mathbb{N} \to \mathsf{Type} \to \mathsf{Term} \to \mathsf{Term}$
$thbsubst\ k\ u\ t = to\ (\{k\ \to\ (from_{\mathsf{Type}}\ u)\}\ (from\ t))$

$yfsubst : \mathbb{N} \to \mathsf{Type} \to \mathsf{Type} \to \mathsf{Type}$
$yfsubst\ k\ u\ t = to\ ([k\ \to\ (from_{\mathsf{Type}}\ u)]\ (from_{\mathsf{Type}}\ t))$

$ywf : \mathsf{Type} \to \mathsf{Prop}$

$thbfsubst\_perm\_core : \forall(t : \mathsf{Term})\ (u, v : \mathsf{Type})\ (m\ k : \mathbb{N}),$
$ywf\ u \Rightarrow$
$thbsubst\ k\ (yfsubst\ m\ u\ v)\ (thfsubst\ m\ u\ t)$
$= thfsubst\ m\ u\ (thbsubst\ k\ v\ t)$

Figure 10. Some representative definitions and lemmas for templates.

1. an isomorphism between the terms of the two representations;
2. operations such as substitution to be compositional with respect to this isomorphism.

Figure 9 presents these requirements more formally for locally nameless representations. In this case what has to be shown is that the object language and substitution operations such as $[\cdot\ \to\ \cdot]_T\ \cdot$ and $\{\cdot\ \to\ \cdot\}_T\ \cdot$[7] on the object language are adequately represented by the generic language and the corresponding generic operations. Laws (1) and (2), the standard isomorphism laws, ensure that the term structure remains unchanged by the application of the two operations *from* and *to* consecutively. Laws (3) and (4) ensure that the locally nameless operations preserve the structure between the concrete and generic definitions of these operations. In general we need similar compositionality laws for other operations.

However, in the case of, GMETA only the first two laws are an obligation of the user, and these can be automatically generated by the isomorphism generator. The reason why laws (3) and (4) (and other similar laws for other operations) are automatically satisfied by GMETA is that they trivially follow from the definitions of the concrete operations (which are provided in the template modules)

$[k\ \to\ u]_T\ t\ = to\ ([k\ \to\ (from\ u)]\ (from\ t))$
$\{k\ \to\ u\}_T\ t = to\ (\{k\ \to\ (from\ u)\}\ (from\ t))$

and the first two laws. In the case of heteregeneous operations the only difference is that several isomorphisms are involved and laws similar to (3) and (4) would involve different types of *from* functions. Nevertheless adequacy is shown in the same way. For other first-order representations, similar adequacy laws are required and the user also has similar obligations. In summary, if the isomorphim generator is used, adequacy falls essentially for *free*; and even if the tool is not used only laws (1) and (2) need to be proved in Coq.

Note, however, that this kind of adequacy relies on the user specifying the particular isomorphism that reflects the intended meaning of the syntax correctly. If the user does not correctly specify the isomorphism, then the syntax would be interpreted with a different, unexpected, meaning. For example if, in Figure 2, the parameter is set to trm_bvar and the variable is set to trm_fvar, then the meaning of terms would be different.

6. Templates

Templates are (ML-style) functors that users should instantiate in order to obtain the basic infrastructure for the object language. Because different languages have different requirements in terms of the infrastructure, GMETA provides a number of templates that can be used according to particular categories of languages.

6.1 Templates for Typed Languages

A very common kind of languages that are often formalized are typed languages. In typed languages there are (typically) two different kinds of syntactic sorts: *terms* and *types*. For each sort an isomorphism should be provided. Therefore, a template for typed languages is parametrized by two isomorphisms (one for terms and another for types). GMETA currently supports two different types of templates for typed languages:

- **simple typed languages (STLs)** In simple (or homogeneous) typed languages there are no type variables or binding constructs on types. Examples of such languages are the STLC and many languages of the ML family.
- **heterogeneous typed languages (HTLs)** In heterogeneous typed languages, the language of types contains binding constructs. An example is the System F family of languages.

Figure 10 shows a number of examples of definitions provided by the two types of templates. All of the definitions and lemmas are available for the HTLs template, but only the definitions and lemmas on the left column are available for the STLs template. This is because the infrastructure required for HTLs is a superset of the infrastructure required by STLs. The operations provided by the templates are typically various forms of substitutions (for example: $tfsubst$, $tbsubst$ and $thfsubst$), free or bound variables and various auxiliary functions such as the size of a term. The templates include many lemmas about the operations and some of the lemmas, such as $thbfsubst\_perm\_core\_ywf$, may depend on the well-formedness properties like ywf.

Most of the lemmas and definitions in STLs only involve terms, since there are no binding constructs on types. In contrast, in HTLs, additionally to the lemmas and definitions required by STLs, various kinds of heterogeneous lemmas and operations (see also the discussion in Section 4.1) are also necessary.

6.2 User-Defined Templates

One nice feature of GMETA is that it is easy to extend or add new templates even for users that do not know much about DGP. This is possible because the definitions in the templates are simple instantiations of the generic functions and Coq's module system makes it very easy for users to create new modules from existing ones.

[7] Note that the subscript T is the type of the object language.

		Definitions	Infrastructure (lemmas + definitions)	Core (lemmas + definitions)	Overall		
					boilerplate	total	ratio
STLC	Aydemir et al.	**11**	$\mathbf{13} + 3$	$\mathbf{4} + 0$	17	31	55%
	GMETA basic	8	$8 + 1$	$4 + 0$	7	21	33%
	GMETA advanced	7	$4 + 0$	$4 + 0$	1	15	7%
System $F_{<:}$	Aydemir et al.	**20**	$\mathbf{48} + 7$	$\mathbf{17} + 1$	60	93	65%
	GMETA basic	13	$26 + 1$	$17 + 1$	25	58	43%
	GMETA advanced	11	$15 + 0$	$18 + 1$	11	45	24%

Figure 11. Formalization of POPLmark challenge (part 1A+2A) and STLC in Coq using locally nameless approach with and without GMETA.

7. Discussion and Evaluation

In this section we elaborate on the results of the case studies introduced in Section 2.3 by discussing these results in terms of the three criteria proposed by Aydemir et al. (2005) (*reasonable overheads*, *cost of entry* and *transparency*) for evaluating mechanical formalizations of metatheory.

7.1 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. As we saw in Section 2.3, in all case studies the solution that used GMETA was able to save more than 58% of the boilerplate definitions and more than 32% of the total numbers of definitions.

Detailed results Figure 11 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.

Basic vs Advanced The main difference between the basic and advanced approaches is that, in the advanced approach many lemmas about well-formed terms in an environment can be reused from GMETA's libraries. This is possible because GMETA's libraries provide a generic inductive logical relation that can be used to talk about well-formed terms in an environment. However, in order to interface with such logical relation, it is desirable to define the inductive logical relation on the object language and show the equivalence between the two logical relations (the concrete and the generic one). The problem is that such equivalence proofs are an obligation of the end-user and they require some knowledge about DGP and some tactics in GMETA. In contrast, in the basic approach the user has to manually define the lemmas about well-formed terms in an environment, but he does not need to know about DGP. Note, however, that in the basic approach it is still possible reuse lemmas about well-formedness that do not depend on an environment.[8]

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. Because this is the only significant 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.

Heterogeneous operations and lemmas Key to the ability of GMETA to reuse operations and lemmas for languages with non-trivial binding structures (such as System F), is the support for generic heterogeneous operations and lemmas. As we saw in Section 4, with heterogeneous lemmas and definitions we can capture a large family of definitions and lemmas with just a few definitions.

7.2 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, the developments using the basic approach do not require knowledge of DGP and the knowledge required by the end-user, for the locally nameless approach, is almost the same as in Aydemir et al. (2008). By using the templates provided by GMETA and a few simple tactics (and modulo some minor differences), a lot of infrastructure is provided essentially for free and all the user-defined proofs remain almost the same as the proofs in the reference solution. In the advanced approach additional savings are possible, at the cost of more knowledge about GMETA. Namely, this approach requires a few additional lemmas to interface with the generic libraries, and the proofs of those lemmas require some understanding about DGP. Nonetheless, the additional investment in learning is paid off by significant additional savings.

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.

[8] In Aydemir et al. (2008), a term is called *locally closed* when it is well-formed.

Finally, we should mention that one advantage of generative approaches such as LNgen 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.

7.3 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).

8. Related Work

Throughout the paper we have already discussed closely related work on generative approaches like LNgen and manual schemes such as the locally nameless style proposed by Aydemir et al. (2008). In this section some additional related work is discussed.

Traditional First-Order Approaches De Bruijn indices (de Bruijn 1972) are among the oldest first-order representations being used for mechanical developments. The particular de Bruijn style used in Section 4.3 is based on Vouillon (2007) solution of the POPLmark challenge. Two other important first-order representations are: nominal representations, the standard approach for pen-and-paper formalizations; and the locally named representations (Mckinna and Pollack 1993), which uses different types of names to represent variables and parameters. For a more complete survey of first-order representations we suggest Aydemir et al. (2008). GMETA currently supports locally nameless and de Bruijn representations, and we are planning to add support for locally named and the nominal representations in the future.

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 $F_{<:}$. While the proposed techniques are effective at achieving significant savings, these techniques require that the object language is encoded in a particular way. In contrast GMETA allows the syntax to be encoded in a natural way, 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) (Harper et al. 1993; Pfenning and Elliot 1988) 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 Chlipala (2008); Despeyroux et al. (1995) 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, while at the same time preserving the simplicity of first-order approaches.

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.

DGP and Binding DGP techniques have been widely used in conventional functional programming languages (Hinze and Jeuring 2003; Jansson and Jeuring 1997; 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 4 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 5 are new. Dybjer and Setzer (2001, 1999) showed universes 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) formalize 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.

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-scoped de Bruijn indices. Licata and Harper (2009) proposed a universe in Agda that permits definitions that mix binding and computation. The key difference between our work and these approaches is that our generic libraries are aimed at providing reusable infrastructure for conventional first-order representations of binders, whereas both Chlipala (2007) and Licata and Harper (2009) are focused on the use of DGP techniques for well-scoped de Bruijn indices.

9. Conclusion and Future Work

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 practical for conducting realistic formalizations of metatheory. Our case studies show that GMETA can deal with challenging languages like System $F<:$ and obtain significant savings of infrastructure. Still, there are many ways in which our work could be improved. We mention a few interesting topics for future work next:

- **Extended DGP support:** There are several ways in which GMETA's DGP can be extended to support more languages and binding constructs. One interesting extension would be support for representations for mutually-inductive definitions of syntax. Previous research on very expressive universes such as the universe for strictly positive families by Morris et al. (2009) is likely to be very relevant for such extension. Supporting multi-binders (that is, binders that bind multiple variables) would

allow languages with constructs such as records and pattern matching to be represented in our universe.

- **Better Coq integration:** Better Coq integration would make GMETA more practical. For example, it would be better to support isomorphism generation directly in Coq, instead of using of an external tool. It would also be nice if Coq could support some kind of module-level partial evaluation, which would be useful for specializing modules for object languages and eliminating the DGP indirections. Currently, end-users eliminate those DGP indirections by applying simplification tactics directly, but it would be better if this step could be avoided. In essence, this would gives us the best of DGP and generative approaches: verified generic infrastructure that can be specialized to a form similar to hand-written definitions.
- **Other representations:** It would also be interesting to add support to GMETA for nominal and locally named representations.

Acknowledgments

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

References

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.

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.

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

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. Indexed induction-recursion. In *PTCS '01*, 2001.

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

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

J. Gibbons. Datatype-generic programming. In *SSDGP'06*, 2007.

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.

D. MacQueen. An implementation of standard ml modules. In *LFP '88*, 1988.

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, Th. Altenkirch, and C. McBride. Exploring the regular tree types. In *TYPES '04*, 2004.

P. Morris, Th. 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, Th. Ridge, S. Sarkar, and R. Strniša. Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20(01):71–122, 2010.

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`.