INFRAGEN: Binder Boilerplate at Scale

Abstract

A key concern in the mechanization of programming language metatheory is the representation of terms with variable binding. The properties of operations manipulating terms are notoriously burdensome to prove and the amount of work required to scale formalizations to realistic programming languages with rich binding forms is deemed infeasible. This is a pity, because we lose the practical benefits of mechanizing real programming languages.

We present a new solution to generically handle the boilerplate involved in mechanizations that scales to rich binding forms and advanced rules of scoping. We define a new specification language for abstract syntax with binding and implement a code generator that produces Coq code for the representation of the abstract syntax, syntactic operations and proofs of their properties.

We illustrate how our approach removes the burden of variable binding boilerplate in the mechanization of realistic programming languages on a list of example specifications and a solution of the PoplMark challenge based on the generated code.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

Keywords variable binders, mechanized meta-theory, datatype-generic programming, code generation

1. Introduction

The meta-theory of programming language semantics and type-systems is highly complex due to the management of many details. Formal proofs are long and prone to subtle errors that can invalidate large amounts of work. In order to guarantee the correctness of formal meta-theory, techniques for mechanical formalization in proof-assistants have received much attention in recent years.

An important issue that arises in many formalizations is the treatment of variable binding which typically comprises the better part of the whole formalization. Most of this variable binding infrastructure is repetitive and tedious boilerplate. To alleviate researchers from this burden, multiple approaches have been developed to capture the structure of variable binding and generically take care of the associated boilerplate. These include specification languages of syntax with binding and scoping rules, tools that generate code for proof assistants from specifications, generic programming libraries that implement boilerplate using datatype generic functions and proofs, and meta-languages that have built-in support for syntax with binding.

Yet, despite all the existing support, mechanized meta-theory is still only rarely used where it can make the biggest difference: full-scale programming languages. The reason is that it is practically infeasible to mechanize a realistic language entirely by hand. The development cost would be prohibitive, and, when small structural changes can have a snowball effect, maintenance would be a nightmare too. Unfortunately, the available tools that cover binder boilerplate are also inadequate to alleviate part of the burden. The problem is that most do not cover the rich-binding forms (such as patterns or declaration lists) and the advanced scoping rules (like sequential and recursive scopes) that are typical of realistic languages, and those that do still leave of most of the boilerplate up to the developer. As a consequence, only drastic simplifications of realistic languages are mechanized, in order to fit the mold of existing tools and make the development cost affordable. For example, multi-variable binders are replaced by single-variable binders and polymorphic languages by monomorphic sublanguages to avoid dealing with multiple distinct namespaces. Obviously there is a very real danger that these simplifications gloss over actual problems in the full-scale language and give a false sense of security.

This work takes a big step towards making the mechanization of realistic programming languages manageable by taking care of the boilerplate for rich binding forms and advanced scoping rules. For this purpose we provide INFRA, our Ott-style specification language [**?**], to concisely and intuitively specify the abstract syntax and rich binding structure of realistic programming languages. From such an INFRA specification our INFRAGEN tool generates Coq code: the necessary datatypes for a de Bruijn representation of the abstract syntax and the corresponding variable binder boilerplate. This boilerplate covers shifting and substitution operations, as well as properties and proofs of these operations.

Our specific contributions are:

- We present a new generic solution to the automatic treatment of variable binding boilerplate based on the INFRA specification that scales to realistic programming languages with rich-binding forms and advanced forms of scoping like recursive and sequential scopes. This combination of expressiveness and the extent of the boilerplate is not covered by any existing work.
- We mechanically verify the meta-theory of INFRA in Coq. This mechanical formalization gives rise to datatype-generic implementations of boilerplate operations and lemmas.
- We develop the INFRAGEN tool that provides a user-friendly way to generate boilerplate code from an INFRA specification for the Coq proof-assistant.
- We perform a case-study, including parts of the POPLMARK challenge, that shows the expressiveness and benefits of our approach.

The code for INFRAGEN and the Coq developments of our case study are available at `http://goo.gl/zcVPTu`.

[Copyright notice will appear here once 'preprint' option is removed.]

α ::=	type variable
x ::=	term variable
τ ::=	type
$\mid\ \alpha$	type variable
$\mid\ \tau \to \tau'$	function type
$\mid\ \tau_1, \tau_2$	product type
$\mid\ \forall\alpha.\,\tau$	universal type
e ::=	term
$\mid\ x$	variable
$\mid\ \lambda x{:}\tau.\,e$	abstraction
$\mid\ e_1\,e_2$	application
$\mid\ \Lambda\alpha.\,e$	type abstraction
$\mid\ e[\tau]$	type application
$\mid\ e_1, e_2$	product
$\mid\ \mathbf{let}\ p{=}e_1\ \mathbf{in}\ e_2$	pattern binding
p ::=	pattern
$\mid\ x$	variable pattern
$\mid\ p_1, p_2$	product pattern
Γ ::=	type environment
$\mid\ \epsilon$	empty env
$\mid\ \Gamma, \alpha$	type binding
$\mid\ \Gamma, x : \tau$	term binding

Figure 1. $F_\times$ syntax

2. Overview

This section gives an overview of the variable binding boilerplate that arises when proving type preservation of typed programming languages. For this purpose we use $F_\times$ (i.e., System F with products and destructuring pattern bindings) as the running example. In the following, we elaborate the different steps of the formalization and point out where variable binding boilerplate arises.

2.1 Syntax: Variable Representation

Figure 1 shows the first step in the formalization: the syntax of $F_\times$. Notice that patterns can be nested and thus can bind an arbitrary number of variables at once. In this grammar the scoping rules are left implicit. The intended rules are that in a type or term abstraction the given variable scopes over the body e and in a pattern binding the variables bound by the pattern scope over e_2 but not e_1.

The syntax raises the first variable-related issue: how to concretely represent variables, issue that is side-stepped in Figure 1. Traditionally one would use identifiers as the set of variables. However, when formalizing meta-theory this representation requires reasoning modulo α-conversion of bound variables to an extent that is excruciating. It is therefore inevitable to choose a different representation of the abstract syntax.

The goal of this paper is not to develop a new approach to variable binding or to compare advantages or disadvantages of existing ones, but rather to scale the generic treatment of a single approach to realistic languages. We choose to work with de Bruijn representations for mainly two reasons. First, reasoning with de Bruijn representations is well-understood and in particular and the representation of pattern binding and scoping rules is also well-understood [**? ?**]. Second, the functions related to variable binding, the statements of properties of these functions and their proofs have highly regular structures with respect to the abstract syntax and the scoping rules of the language. This helps us in treating boilerplate generically and automating proofs.

The term grammar below encodes a de Bruijn representation of $F_\times$. The variable occurrences of binders have been removed in this representation and the referencing occurrence of type and term variables are replaced by de Bruijn indices n.

$$\begin{array}{ll} T & := n \mid Arr\ T_1\ T_2 \mid Prod\ T_1\ T_2 \mid All\ T \\ p & := pvar \mid pprod\ p_1\ p_2 \\ t & := n \mid abs\ T\ t \mid app\ t_1\ t_2 \mid tabs\ t \mid \\ & \quad\ tapp\ t\ T \mid prod\ t_1\ t_2 \mid \mathbf{let}\ p\ t_1\ t_2 \end{array}$$

The de Bruijn indices point directly to their binders: The index k points to the kth enclosing binding position. For instance, the $F_\times$ expression for the polymorphic swap function

$$\Lambda\alpha.\,\Lambda\beta.\,\lambda x{:}(\alpha,\beta).\,\mathbf{let}\,(x_1,x_2){=}x\,\mathbf{in}\,(x_2,x_1)$$

is represented by the de Bruijn term

$$\begin{array}{l} tabs\ tabs\ abs\ (Prod\ 1\ 0) \\ \quad (\mathbf{let}\ (pprod\ pvar\ pvar)\ 0\ (prod\ 0\ 1)) \end{array}$$

Again the order in which de Bruijn indices are bound and the scoping rules are left implicit in the term grammar. Our specification language INFRA for de Bruijn terms from Section 3 will make order of binding and scoping rules explicit.

A second example representation is $tabs\ tabs\ (abs\ 1\ (abs\ 0\ 1))$ for the polymorphic $const$ function $\Lambda\alpha.\,\Lambda\beta.\,\lambda x{:}\alpha.\,\lambda y{:}\beta.\,x$.

We treat indices for variables from distinct namespaces independently: The index for the type variable β that is used in the inner abs is 0 and not 1, because we only count the number of binders for the corresponding namespace but not binders for other namespaces.

2.2 Semantics: Shifting and Substitution

The next step in the formalization is to develop the typical semantic relations for the language of study. In the case of $F_\times$ these comprise a small-step call-by-value operational semantics, as well as a well-formedness relation for types, a typing relation for terms and a typing relation for patterns.

This requires us to develop our first set of variable boilerplate: type and term substitution operations on the de Bruijn representation. For this, we need an auxiliary operation called *shift* that adapts the indices of the free variables in an expression. More specifically, if a type substitution goes under a binder, as in

$$[\alpha \mapsto \tau_1](\Lambda\beta.\,\tau_2)$$

the substitution is continued in a context that has one more variable. Accordingly we need to increment the index of α and the indices of free variables in the representation of τ_2 so that they still refer to the same variables.

To restrict shifting to free variables but not bound variables, the $shift_\tau$ function below takes a *cutoff* parameter c. Initially the cutoff is zero and when going under a binder that number is incremented with the amount of variables that are being bound. Thus the indices of bound variables are smaller than the cutoff and the indices of free variables are greater or equal to the cutoff.

Definition 1. *The one-place shift of types is recursively defined as*

$$\begin{array}{ll} shift_\tau\ c\ n & = shift_{\mathbb{N}}\ c\ n \\ shift_\tau\ c\ (Arr\ T_1\ T_2) & = Arr\ (shift_\tau\ c\ T_1)\ (shift_\tau\ c\ T_2) \\ shift_\tau\ c\ (All\ T) & = All\ (shift_\tau\ (1+c)\ T) \\ shift_\tau\ c\ (Prod\ T_1\ T_2) & = Prod\ (shift_\tau\ c\ T_1)\ (shift_\tau\ c\ T_2) \end{array}$$

and the one-place shift of an index as

$$shift_{\mathbb{N}}\ c\ n = \begin{cases} 1+n & , n \geqslant c \\ n & , n < c. \end{cases}$$

Substitution is typically used for β-reduction: given a type abstraction applied to a type, we substitute the type variable by the given type in the body and remove the abstraction

$$(\Lambda\alpha.\tau_2)\,\tau_1 \xrightarrow{\beta} [\alpha \mapsto \tau_1]\,\tau_2$$

Because the abstraction is removed we need to adapt the indices of variables. Indices of variables bound in τ_2 can remain unchanged; they have an index that is strictly smaller than that of α. Indices of free variables of τ_2 need to be decremented to account for the fact that α's binder is removed; they have an index that is strictly greater than that of α. We are now able to define type substitution $subst_\tau\ m\ S\ T$ on the de Bruijn representation.

Definition 2. *Type substitution in types is defined as*

$$\begin{aligned}
&subst_\tau\ m\ \tau\ n && = subst_{\mathbb{N}}\ m\ \tau\ n\\
&subst_\tau\ m\ \tau\ (Arr\ \tau_1\ \tau_2) && = Arr\ (subst_\tau\ m\ \tau\ \tau_1)\\
& && \qquad (subst_\tau\ m\ \tau\ \tau_2)\\
&subst_\tau\ m\ \tau\ (Prod\ \tau_1\ \tau_2) && = Prod\ (subst_\tau\ m\ \tau\ \tau_1)\\
& && \qquad (subst_\tau\ m\ \tau\ \tau_2)\\
&subst_\tau\ m\ \tau\ (All\ \tau') && =\\
&\quad All\ (subst_\tau\ (m+1)\ (shift_\tau\ 0\ \tau)\ \tau')
\end{aligned}$$

and the type substitution for indices is defined as

$$subst_N\ m\ \tau\ n = \begin{cases} n & , n < m\\ \tau & , n = m\\ n-1 & , n > m.\end{cases}$$

We also need to define type shifting and type substitution in terms as well as term shifting and term substitution in terms. For the latter two, the interesting case is pattern bindings. Given the function b that counts the number of variables in a pattern p

$$\begin{aligned}
&b\ (pvar) && = 1\\
&b\ (pprod\ p_1\ p_2) && = b\ p_2 + b\ p_1
\end{aligned}$$

we have

$$shift_e\ c\ (\mathbf{let}\ p\ e_1\ e_2) = \mathbf{let}\ p\ (shift_e\ c\ e_1)\ (shift_e\ (b\ p + c)\ e_2)$$

and

$$\begin{aligned}
&subst_e\ m\ s\ (\mathbf{let}\ p\ e_1\ e_2) =\\
&\quad \mathbf{let}\ p\ (subst_e\ m\ s\ e_1)\ (subst_e\ (b\ p + m)\ s\ e_2)
\end{aligned}$$

2.3 Theorems: Commutation, Weakening and Preservation

Given the definitions from the previous subsection, we are ready to define the semantics and type system of $F_\times$ and move on to formulate and prove type soundness for $F_\times$. We refrain from formulating it here explicitly. The proof of type soundness involves the usual lemmas for inversion of values, well-definedness of pattern matching, progress and preservation [**?**]. In order to prove these lemmas we require a second set of variable binding boilerplate: lemmas for various properties of the shift and substitution relations, both on the level of terms and of relations.

- At the level of terms, lemmas include commutation between two operations in the same or distinct namespaces, e.g., the commutation of two shifts in the namespace of types or the commutation of a type substitution with a term substitution.
- At the level of relations, we have weakening lemmas, preservation under type substitution as well as preservation under term substitution.

2.4 Summary

Table 1 summarizes the effort required to formalize type soundness of $F_\times$ in the Coq proof assistant in terms of the de Bruijn representation. It lists the lines of Coq code for the three different parts of the formalization discussed above, divided in binder-related "boilerplate" and the other "useful" code.

The table clearly shows that the boilerplate constitutes more than half of the formalization (57.3%). The boilerplate lemmas in particular, while individually fairly short, make up the bulk of the boilerplate and close to half of the whole formalization (46.2%).

	Useful	**Boilerplate**	
Syntax	23	0	(0%)
Semantics	177	132	(11.0%)
Theorems	311	553	(46.2%)
Total	511	685	(57.3%)

Table 1. Lines of Coq code for the $F_\times$ meta-theory mechanization.

Of course, very similar variable binder boilerplate arises in the formalization of other languages, where it requires a similar unnecessarily large development effort. Rossberg et al. [**?**] report that 400 out of 500 lemmas of their mechanization in the locally-nameless style [**?**] were tedious boilerplate lemmas.

Fortunately there is much regularity to the boilerplate: it follows the structure of the language's abstract syntax and its scoping rules. Many earlier works have already exploited this fact in order to automatically generate or generically define part of the boilerplate for simple languages.

2.5 Objectives and Approach

The aim of this work is to exploit the generic structure of variable binder boilerplate to cater for large-scale languages that contain complex binding structures, like the nested pattern matches of $F_\times$, recursive multi-binders, ... Moreover, because the formalization of large-scale languages would otherwise be prohibitive we cover a larger extent of the boilerplate than earlier works.

Our approach consists of a specification language, called INFRA, that precisely covers the universe of supported languages. We provide generic definitions and lemmas for the variable binding boilerplate that apply to every well-formed INFRA specification. Finally, we complement the generic approach with a code generator, called INFRAGEN, that specializes the generic definitions and allows manual customization and extension.

3. Grammars and Binding Specifications

This section introduces INFRA, the language for specifying the abstract syntax and associated variable binder information of programming languages. The advantage of specifying programming languages in INFRA is straightforward: the variable binder boilerplate comes for free for any well-formed INFRA specification.

The syntax of INFRA allows programming languages to be expressed in terms of different syntactic sorts, term constructors for these sorts and binding specifications for these term constructors. The latter specify the number of variables that are bound by the term constructors as well as their scoping rules.

The semantics of INFRA explains which concrete terms inhabit a particular INFRA specification. As noted in the previous section, INFRA assumes concrete terms make use of the de Bruijn representation for variables.

3.1 INFRA Syntax

Figure 2 shows the grammar of INFRA. An INFRA specification *spec* of a language consists of a declaration of variable namespaces α and of declarations of syntactic sort declarations $sort$.

A sort declaration consists of constructor declarations $ctor$, of which there are two kinds. It is either a variable constructor $C\ \alpha$ that holds a de Bruijn index that references the namespace α or it is a constructor $C\,\overline{nt_i\ bs_i}^{\,i}\ :\ \overline{fn_j = vle_j}^{\,j}$ that has subterms but no indices as immediate children. The i-th subterm is named nt_i

spec	$::=$		Specification
	$\mid$	$\textbf{namespaces}\,\overline{\alpha_i}^{\,i}\,\overline{\mathit{sort}_j}^{\,j}$	
$\alpha,\ \beta$	$::=$		Namespace
sort	$::=$		Sort declaration
	$\mid$	$s := \overline{\mathit{ctor}_i}^{\,i}$	
s	$::=$		Sort names
ctor	$::=$		Constructor decl.
	$\mid$	$C\,\alpha$	
	$\mid$	$C\,\overline{\mathit{nt}_i\ \mathit{bs}_i}^{\,i}\ :\ \overline{\mathit{fn}_j = \mathit{vle}_j}^{\,j}$	
C	$::=$		Constructor name
nt	$::=$		Field name
	$\mid$	$s\ \mathit{suff}$	
suff	$::=$		Suffix
bs	$::=$		Binding specification
	$\mid$	$\overline{\mathit{vle}_i}^{\,i}$	
vle	$::=$		Variable list
	$\mid$	ϵ	empty
	$\mid$	α	singleton
	$\mid$	$\mathit{fn}(\mathit{nt})$	function call
	$\mid$	$\mathit{vle}_1, \mathit{vle}_2$	concatenation
fn	$::=$		Function name

Figure 2. Grammars with binding specifications

and has a binding specification bs_i. A subterm name consists of a sort name s followed by an optional numeric suffix. The former specifies the sort of the subterm and the latter distinguishes it from other subterms of the same sort. The subterm's binding specification bs_i stipulates which variables are bound by the term constructor and brought in scope of the subterm. The binding specification consists of a heterogeneous list of homogeneous variable list expression vle. We will discuss this choice in more detail in Section 8. A variable list expression vle represents a homogeneous list of nameless variables, i.e. variables that live in the same namespace. A variable list expression is either empty, a singleton variable form a given namespace, the concatenation of two variable list expressions or the invocation of an auxiliary function.

Each non-variable constructor is annotated with $\overline{fn_j = vle_j}^{\,j}$, a list of auxiliary function bindings. These auxiliary functions calculate variable lists from terms and serve to specify binders in binding specifications.

3.2 Examples

The following examples of rich binder forms illustrate the expressive power of INFRA.

Nested Patterns Figure 3 shows the INFRA specification of $F_\times$. We start with the declaration of two namespaces: X for type variables and x for term variables, which is followed by the declarations of $F_\times$'s three sorts: types, patterns and terms. For readability reasons we have omitted empty binding specifications from these declarations. The INFRA specification contains only four non-empty binding specifications: universal quantification for types and type abstraction for terms each bind exactly one type variable, the lambda abstraction for terms binds exactly one term variable and the destructuring let binds $b(p)$ variables in t_2 where b is an auxiliary function defined on patterns.

Recursive Scopes Figure 4 shows the specification of a simply-typed lambda calculus with recursive let definitions like the ones found in the Haskell programming language.

```
namespaces X, x
T :=
   | Var    X
   | Arr    T1 T2
   | All    T [X]
   | Prod   T1 T2
p :=
   | pvar          : b = X
   | pprod p1 p2 : b = b(p1), b(p2)
t :=
   | var    x
   | abs    T t [x]
   | app    t1 t2
   | tabs   t1 [X]
   | tapp   t T
   | prod   t1 t2
   | let    p t1 t2 [b(p)]
```

Figure 3. Example specification of $F_\times$

```
namespaces x
T :=
   | Top
   | Arr    T1 T2
d :=
   | nil         : b = ϵ
   | cons   t d : b = x, b(d)
t :=
   | var    x
   | abs    T t [x]
   | app    t1 t2
   | letrec d [b(d)] t [b(d)]
```

Figure 4. Example specification of recursive let

The auxiliary function b collects the variables bound by a declaration list d. In the term constructor $letrec$ we specify that the variables of d are not only bound in the body t but also in d itself.

Interdependent Namespaces Figure 5 shows the specification of a lambda calculus with first-order dependent types as presented by Pierce [**?**]. In this language terms t and types T are mutually recursive and have distinct namespaces. Type variables can be declared in the context with a specific kind K but are never bound in the syntax.

Sequential Scoping Telescopes were invented to model dependently-typed systems [**?**]. A telescope is a list of of variables together with their types

$$x_1 : T_1, \ldots, x_n : T_n$$

where each variable scopes over types that appear later in the list.

The calculus presented in Figure 6 uses telescopic abstractions. In the abstract syntax of telescopes D the variable names are erased but the sequential scoping is captured in the specification. In the lambda abstraction case abs and the dependent function type constructor pi the variables of a telescope are bound simultaneously in the body.

$$
\begin{array}{l}
namespaces\ X, x \\
T := \\
\quad \mid\ Var\ \ X \\
\quad \mid\ Pi\ \ \ T_1\ T_2\ [x] \\
\quad \mid\ App\ \ T\ t \\
t\ \ := \\
\quad \mid\ var\ \ x \\
\quad \mid\ abs\ \ T\ t\ [x] \\
\quad \mid\ app\ \ t_1\ t_2 \\
K := \\
\quad \mid\ star \\
\quad \mid\ pi\ \ \ T\ K\ [x]
\end{array}
$$

Figure 5. Example specification of λLF

$$
\begin{array}{l}
namespaces\ x \\
t\ \ := \\
\quad \mid\ var\ \ x \\
\quad \mid\ abs\ \ D\ t\ [b(D)] \\
\quad \mid\ pi\ \ \ D\ t\ [b(D)] \\
\quad \mid\ app\ \ t_1\ t_2 \\
D := \\
\quad \mid\ nil\ \qquad\qquad\ : b = \epsilon \\
\quad \mid\ cons\ T\ D\ [x] : b = x, b(D)
\end{array}
$$

Figure 6. Example specification of telescopic lambdas

3.3 Well-Formed INFRA Specifications

Figure 7 defines the well-formedness relation $\Phi \vdash spec$ for INFRA specifications $spec$ with respect to the global function environment Φ. This global function environment Φ associates with each function *fn* a signature $s \rightarrow \alpha$. The single rule WFSPEC expresses that a specification is well-formed if each of its sort declarations is.

Rule WFSORT in turn states that a sort declaration is well-formed if each of its constructor declaration is well-sorted with the declaration's sort. The auxiliary well-sorting relation $\Phi \vdash ctor : s$ denotes that constructor declaration $ctor$ has sort s with respect to the global environment Φ. There are two rules for this relation, one for each constructor form.

The grammar of INFRA already syntactically enforces that no auxiliary functions are defined for variable constructors. Rule WF-VAR goes further and requires that in this case there is no function for that sort in the function environment Φ. This implies that, if a sort has a variable constructor, its other constructors cannot have function definitions either. The restriction is necessary to guarantee that auxiliary functions are stable under shifting and substitution. For instance, substituting terms in a list of let-bound declarations does not change the number of variables that is bound by that declaration list.

Rule WFCTOR handles the case of non-variable constructors. This rule requires that the binding specification of each subterm is well-formed and that the function definition is well-typed. The former is regulated by rule WFBINDSPEC, which requires the binding specification to be a (possibly) heterogeneous list of homogeneously typed sublists.

The relation $\Phi, \Gamma \vdash vle : \alpha$ denotes that variable list expression vle is typed homogeneously with elements from namespace α with respect to global function environment Φ and local non-terminal environment Γ. It is defined by four rules. By rule WFVLEEMPTY, the empty list ϵ has elements from any namespace α. Rule WFVLESINGLE states that the singleton list, denoted by α, has elements from the corresponding namespace α. Rule WFVLEAPPEND requires that the two subterms of a concatenation are typed homogeneously. Finally, rule WFVLEAUX makes sure that function calls respect the function signature.

In addition to the explicitly formulated well-formedness requirements of Figure 7, we also require a number of simple consistency properties:

1. Constructor names are not repeated for different constructor declarations.
2. Field names are not repeated in a constructor declaration.
3. For each namespace α there is a unique variable constructor declaration $C\ \alpha$.

The first two requirements avoid ambiguity and follow good practice. The third requirement expresses that every variable belongs to one sort and there is only one way, i.e., one term constructor, to inject it in that sort.

3.4 INFRA Semantics

We are now able to generically define abstract syntax terms for well-formed INFRA specifications. Fix a specification $spec$ and suppose $spec$ is well-formed with function environment Φ. We assume that information about constructors is also available in a global environment. We will use $(C : \alpha \rightarrow s)$ for looking up the namespace α and sort s of a variable constructor and $(C : \overline{nt_i\ bs_i}^{\,i} \rightarrow s)$ for retrieving the field names and binding specification of non-variable constructors. When we are only interested in the sorts of the fields, we will write $(C : \overline{s_i}^{\,i} \rightarrow s)$ instead.

Figure 8 contains a term grammar for raw terms. A term consists of either a term constructor applied to a natural number or a term constructor applied to other terms. Figure 9 contains a judgment for well-sortedness of terms. Rule WFASTVAR states that a term constructor applied to a natural number is well-sorted with s if it was declared as a variable constructor in the declaration of s. Finally, rule WFASTCTOR makes sure that for non-variable constructors the arity and sorts of subterms are respected.

4. Infrastructure Operations

In the previous section we have introduced the INFRA specification language and given it a semantics by declaring the abstract syntax terms that are valid with respect to the specification. In this section we generically define the boilerplate syntax operations on valid abstract syntax terms.

Again, we fix a specification $spec$ with function environment Φ. The first operation we consider is the evaluation of variable list expressions, that return the number of variables bound by a binding construct. Subsequently we define shifting and substitution operations that respect the scoping rules defined by the binding specifications that are part of $spec$.

For the definitions in this chapter we make use of several helper functions. The function $(defOf : fn \rightarrow C \rightarrow vle)$ returns the defining variable list expression of a function *fn* for the constructor C. Furthermore, we need a function $(namespacesOf : s \rightarrow \overline{\alpha_i}^{\,i})$ that associates sorts with the set of namespaces they depend on. For example, in $F_\times$ types do not depend on the namespace for term variables. We will use that in the definition of the operations to not recurse into subterms of sorts on which the operations are the identity function. We require $namespacesOf$ to fulfil two sanity conditions.

1. For each constructor $(C : \alpha \rightarrow s)$

$$\alpha \in namespacesOf\ s$$

$$\begin{array}{lll} \Phi & ::= & & \text{Function environment} \\ & | & \overline{fn_i : s_i \to \alpha_i}^{\,i} & \\ \Gamma & ::= & & \text{Field environment} \\ & | & \overline{nt_i}^{\,i} & \end{array}$$

$\boxed{\Phi \vdash spec}$

$$\frac{\forall j.\ \Phi \vdash sort_j}{\Phi \vdash \textbf{namespaces}\ \overline{\alpha_i}^{\,i}\ \overline{sort_j}^{\,j}}\ \textsc{WfSpec}$$

$\boxed{\Phi \vdash sort}$

$$\frac{\forall i.\ \Phi \vdash ctor_i : s}{\Phi \vdash s ::= \overline{ctor_i}^{\,i}}\ \textsc{WfSort}$$

$\boxed{\Phi \vdash ctor : s}$

$$\frac{\forall \beta.\ \nexists fn.(fn : s \to \beta) \in \Phi}{\Phi \vdash C\,\alpha : s}\ \textsc{WfVar}$$

$$\frac{\begin{array}{l}\forall i.\ \Phi, \overline{nt_i}^{\,i} \vdash bs_i \\ \forall (fn : s \to \alpha) \in \Phi.\ \exists\, i. \\ \qquad fn = fn_i \wedge \Phi, \overline{nt_i}^{\,i} \vdash vle_i : \alpha \\ \forall j.\ \exists\, \alpha.(fn_j : s \to \alpha) \in \Phi\end{array}}{\Phi \vdash C\ \overline{nt_i\ bs_i}^{\,i}\ :\ \overline{fn_j = vle_j}^{\,j} : s}\ \textsc{WfCtor}$$

$\boxed{\Phi, \Gamma \vdash bs}$

$$\frac{\forall i.\ \exists\, \alpha. \Phi, \Gamma \vdash vle_i : \alpha}{\Phi, \Gamma \vdash \overline{vle_i}^{\,i}}\ \textsc{WfBindSpec}$$

$\boxed{\Phi, \Gamma \vdash vle : \alpha}$

$$\frac{}{\Phi, \Gamma \vdash \epsilon : \alpha}\ \textsc{WfVleEmpty}$$

$$\frac{}{\Phi, \Gamma \vdash \alpha : \alpha}\ \textsc{WfVleSingle}$$

$$\frac{\begin{array}{l}(fn : s \to \alpha) \in \Phi \\ (s\ suff) \in \Gamma\end{array}}{\Phi, \Gamma \vdash fn(s\ suff) : \alpha}\ \textsc{WfVleAux}$$

$$\frac{\begin{array}{l}\Phi, \Gamma \vdash vle_1 : \alpha \\ \Phi, \Gamma \vdash vle_2 : \alpha\end{array}}{\Phi, \Gamma \vdash vle_1, vle_2 : \alpha}\ \textsc{WfVleAppend}$$

Figure 7. Well-formed specifications

2. and for each constructor $(C : \overline{s_i}^{\,i} \to s)$

$$\forall i. namespacesOf\ s_i \subseteq namespacesOf\ s.$$

4.1 Variable List Evaluation

The binding specification for a particular subterm of a given term constructor defines how many variables are bound by the constructor in that subterm. The evaluation operator captures the semantics of binding specifications by evaluating the binding specification for a given term and returning the concrete number of variables that are bound.

Figure 10 defines the evaluation operator $eval_\alpha$ for variable list expressions and auxiliary functions. The evaluation of a variable

$$\begin{array}{llll} n,\ m,\ c & ::= & 0 \mid S\ n & \text{De Bruijn index} \\ t & ::= & & \text{Abstract syntax terms} \\ & | & C\ n & \\ & | & C\ \overline{t_i}^{\,i} & \end{array}$$

Figure 8. Grammar for abstract syntax terns

$\boxed{\vdash t : s}$

$$\frac{C : \alpha \to s}{\vdash C\ n : s}\ \textsc{WfAstVar}$$

$$\frac{\begin{array}{l}C : \overline{s_i}^{\,i} \to s \\ \forall i.\ \vdash t_i : s_i\end{array}}{\vdash C\ \overline{t_i}^{\,i} : s}\ \textsc{WfAstCtor}$$

Figure 9. Well-sorted abstract syntax terms

$\boxed{eval_\alpha :: \overline{t_i}^{\,i} \to vle_\alpha \to vl_\alpha}$

$$\begin{array}{ll} eval_\alpha\ \overline{t_i}^{\,i}\ \epsilon & = 0 \\ eval_\alpha\ \overline{t_i}^{\,i}\ \alpha & = 1 \\ eval_\alpha\ \overline{t_i}^{\,i}\ (vle_1, vle_2) & = eval_\alpha\ \overline{t_i}^{\,i}\ vle_2 + eval_\alpha\ \overline{t_i}^{\,i}\ vle_1 \\ eval_\alpha\ \overline{t_i}^{\,i}\ (fn\ nt_i) & = eval_\alpha\ fn\ t_i \end{array}$$

$\boxed{eval_\alpha :: fn \to t \to vl_\alpha}$

$$eval_\alpha\ fn\ (C\ \overline{t_i}^{\,i}) = eval_\alpha\ \overline{t_i}^{\,i}\ (defOf\ \ C\ fn)$$

Figure 10. Evaluation of variable list expressions and functions

list expression is always performed in the context of a particular constructor C. This is taken into account in the call to $eval_\alpha$ by supplying as the first argument the list of subterms for the fields of C. The interesting case of $eval_\alpha$ is that of a function call $fn(nt_i)$, which evaluates fn on the corresponding term t_i. The evaluation of a function pattern matches on the term and looks up the definition of the function for that constructor using $defOf$. Note that we have ruled out function definitions for variable constructors. Thus, we do not need to handle that case here.

4.2 Shifting

We now define the shift operation generically over all abstract syntax terms. We can perform the shift in each namespace independently. Our definitions for shift in Figure 11 are therefore parameterized over a namespace α.

The function $shift_\alpha$ has two arguments: a cutoff and a term t. The case for the variable constructor $C : \alpha \to s$ shifts the index by using $shift_{\mathbb{N}}$ that we defined in Section 2. For variable constructors of other namespaces we keep the index unchanged. In the case of a non-variable constructor we need to calculate the cutoffs for the recursive calls. This is done using the $lift_\alpha$ function. $lift_\alpha$ takes as parameters the binding specification bs of a field nt_i, the subterms $\overline{t_i}^{\,i}$ for all fields and a cutoff c. It will increment c by evaluation every part of the binding specification for the namespace α. Using the calculated cutoffs, the $shift'_\alpha$ function can proceed recursively on the subterms that depend on the namespace α.

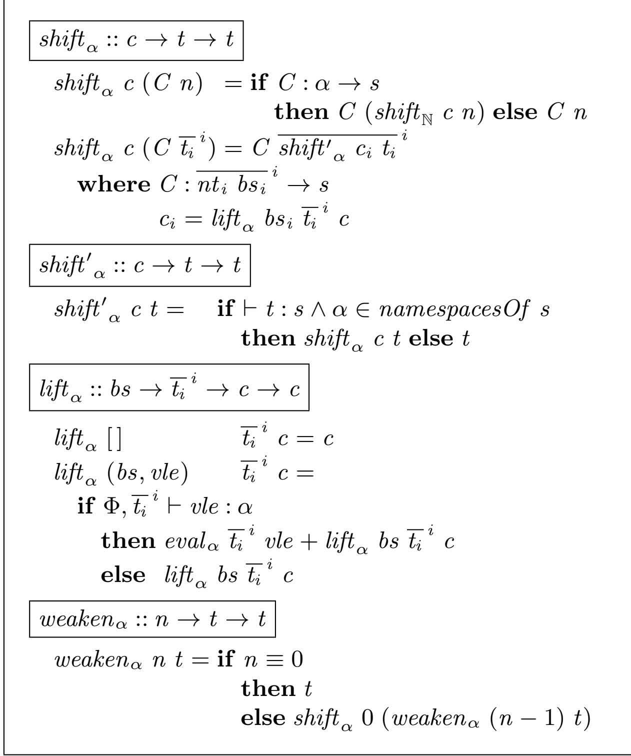

Figure 11. Shifting of terms

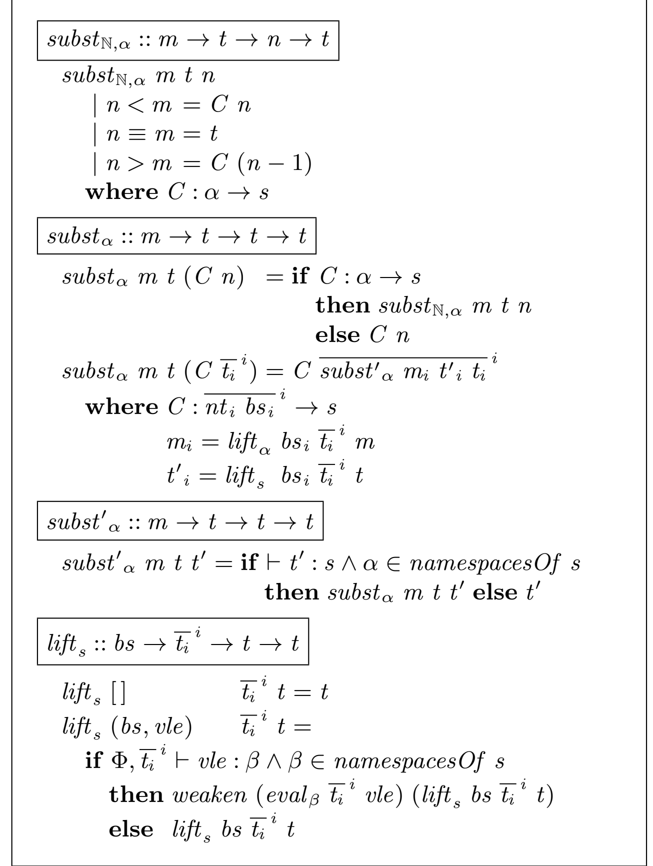

Figure 12. Substitution of terms

4.3 Weakening

As Figure 11 shows, we implement k-place weakening $weaken_\alpha$ by iterating the 1-place $shift_\alpha$ operator. Weakening will be instrumental in hoisting terms under binders in the definition of substitutions.

4.4 Substitution

In the remainder of this section we define a substitution operation generically for all namespaces. Similar to shifting we have three substitution functions and one auxiliary function $lift_s$. In all cases, the index m is to be substituted by the term t given as the second argument.

$subst_{\mathbb{N},\alpha}$ implements the necessary arithmetic to define substitutions for de Bruijn indices. In the case $n > m$ the decrement is needed, because we consider the variable to be removed from the context. The substitution for terms $subst_\alpha$ recurses on the term structure, lifts the index m and term t under the binding specification of the subterms. The variant $subst'_\alpha$ only recurses when when it is necessary to do so.

The auxiliary function $lift_s$ implements the hoisting of terms under binders for the sort s. For each part of a binding specification we weaken the term t if s depends on the namespace of the binder.

5. Infrastructure Lemmas

In this section we look at properties of the shift and substitution operators that are useful in mechanization. We discuss the steps in the proof of a lemma on commutations of shift to explain the reasoning approach that is involved. Understanding each step in detail is important for the correctness of our code generator INFRAGEN.

5.1 Example: Shift Commutation

Lemma 3. *For every namespace α the following holds*

$$shift_\alpha\ 0 \circ shift_\alpha\ c = shift_\alpha\ (1 + c) \circ shift_\alpha\ 0$$

The intuition behind Lemma 3 is the following: We are using shift to adapt indices whenever a new variable is introduced into the context, e.g. by going under a binder. $shift\ c$ adapts indices for an introduction of a variable at position c and $shift\ 0$ for a new variable at the end of the context. The lemma says how these operations commute. Inserting a new variable x at position c and then a new variable y at position 0 is the same as inserting y first at 0 and then accounting for y when inserting x, i.e. inserting it at position $1 + c$.

A problem of Lemma 3 is that we cannot prove it by induction over terms directly. The reason is that the cutoffs change when going under a binder. For instance, when going under a lambda abstraction that binds a single variable, we need to show

$$shift_\alpha\ 1\ (shift_\alpha\ (1 + c)\ t) = shift_\alpha\ (2 + c)\ (shift_\alpha\ 1\ t)$$

for the body t of the lambda abstraction. The induction hypothesis for t would be useless in this case. Hence, in order to obtain a useful induction hypothesis we need to strengthen Lemma 3 to:

Lemma 4. *For every namespace α and natural numbers k, c the following holds*

$$shift_\alpha\ k \circ shift_\alpha\ (k + c) \equiv shift_\alpha\ (k + (1 + c)) \circ shift_\alpha\ k$$

Note that the variable case of Lemma 4 is independent from a concrete specification. Hence, it is worthwhile to capture it in a separate reusable lemma that we prove once and for all:

Lemma 5. *For every natural number k and c the following holds*

$$\mathit{shift}_{\mathbb{N}}\ k \circ \mathit{shift}_{\mathbb{N}}\ (k+c) \equiv \mathit{shift}_{\mathbb{N}}\ (k+(1+c)) \circ \mathit{shift}_{\mathbb{N}}\ k$$

Proof. By arithmetic reasoning. □

To prove Lemma 4 we will need two more auxiliary lemmas about evaluation of binding specifications.

Lemma 6. *For every namespace α, binding specification bs, terms $\overline{t_i}^{\,i}$ and cutoffs $\overline{c_i}^{\,i}$ we have*

$$\mathit{lift}\ bs\ \overline{\mathit{shift}_\alpha\ c_i\ t_i}^{\,i} \equiv \mathit{lift}\ bs\ t_i$$

Proof outline. By induction over binding specification bs and using similar lemmas for the evaluation of variable list expressions and auxiliary functions.

Lemma 7. *For every namespace α, binding specification bs, terms $\overline{t_i}^{\,i}$ and natural number k, c we have*

$$\mathit{lift}\ bs\ t_i\ (k+c) \equiv \mathit{lift}\ bs\ t_i\ k+c$$

Proof outline. By induction over binding specification bs and using the associativity of addition.

The shift and substitution operators only act on variables and leave the rest of the term unchanged. To prove Lemma 6, we needed to rule out auxiliary functions for sorts with variables. A similar lemma holds for substitutions. Lemma 7 reassociates a list of additions

$$n_1 + (... + (n_i + (k+c))...)$$

into

$$(n_1 + (... + (n_i + k)...) + c.$$

Proof outline of Lemma 4. By induction over terms. Use Lemma 5 in the variable case. For the non-variable case we need Lemma 6 to ensure that the same subterms are used in the evaluation of the binding specification to lift the cutoff. Use Lemma 7 to reassociate the lifted cutoffs to match the form of the induction hypotheses. □

In contrast to Lemma 5 which we can implement generically, our code generator INFRAGEN does generate specialized code for Lemmas 4, 6 and 7. However, all difficult arithmetic reasoning that is involved is encapsulated in the generic variable case. The rest of the proof that is generated only involves fairly easy reasoning about associativity.

5.2 Overview: Infrastructure Lemmas

To conclude this section we give a short overview of the lemmas that we have formalized and implemented in INFRAGEN.

Lemma 8. *The following hold for operations in the same namespace.*

1. $\mathit{subst}\ x\ s\ (\mathit{shift}\ x\ t) \equiv t$
2. $\mathit{shift}\ 0\ (\mathit{subst}\ x\ s\ t) \equiv$ $\mathit{subst}\ (1+x)\ (\mathit{shift}\ c\ s)\ (\mathit{shift}\ c\ t)$
3. $\mathit{shift}\ c\ (\mathit{subst}\ 0\ s\ t) \equiv$ $\mathit{subst}\ 0\ (\mathit{shift}\ c\ s)\ (\mathit{shift}\ (1+c)\ t)$
4. $\mathit{subst}\ x\ s_2\ (\mathit{subst}\ 0\ s_1\ t) \equiv$ $\mathit{subst}\ 0\ (\mathit{subst}\ x\ s_2\ s_1)\ (\mathit{subst}\ (1+x)\ (\mathit{shift}\ 0\ s_2)\ t)$

The following lemmas cover the interaction between operations in different namespaces $\alpha \neq \beta$. Note that we use $\mathit{shift'}_\alpha$ in cases where the statement of the lemma depends on the namespaces involved.

5. $\mathit{shift}_\alpha\ c\ (\mathit{shift}_\beta\ c\ t) \equiv$ $\mathit{shift}_\beta\ c\ (\mathit{shift}_\alpha\ c\ t)$
6. $\mathit{shift}_\alpha\ c\ (\mathit{subst}_\beta\ m_\beta\ t\ t) \equiv$ $\mathit{subst}_\beta\ m_\beta\ (\mathit{shift'}_\alpha\ c\ t)\ (\mathit{shift}_\alpha\ c\ t)$
7. $\mathit{subst}_\alpha\ m_\alpha\ t\ (\mathit{subst}_\beta\ m_\beta\ t\ t) \equiv$ $\mathit{subst}_\beta\ m_\beta\ (\mathit{subst'}_\alpha\ m_\alpha\ t\ t)\ (\mathit{subst}_\alpha\ m_\alpha\ (\mathit{shift'}_\beta\ m_\beta\ t)\ t)$

Proof outline. The proofs of the first group follow the same structure as the proof of the shift commutation lemma of Section 5.1. For the second group a simple induction over the terms is sufficient, there is no need to strengthen the statements.

6. INFRAGEN

While the generic Coq definitions presented in the previous sections are satisfactory from a theoretical point of view, they are less so from a pragmatic perspective. The reason is that the generic code only covers the variable binder boilerplate; the rest of a language's formalization still needs to be developed manually. Developing the latter part directly on the generic form is cumbersome. Interacting indirectly with the generic definitions through conversion functions is not much better.

For this reason we also implemented a code generation tool, called INFRAGEN that generates all the boilerplate in a language-specific non-generic form. INFRAGEN takes an INFRA specification and generates Coq code: the inductive definitions of a de Bruijn representation of the object language and the corresponding specialized boilerplate definitions, lemmas and proofs. This generated Coq code is linked against a companion library of reusable functions, proofs and proof tactics.

6.1 Code Generation

Syntax The INFRAGEN tool generates inductive definitions for each sort in the INFRA specification. Furthermore, it analyzes mutually recursive groups, creates mutually recursive definitions for the groups and derives corresponding induction schemes, so that lemmas can be proven by mutual induction.

Function Boilerplate The effect of shifting on de Bruijn indices is generically defined in the companion library. INFRAGEN generates a function that traverses a term to its variable positions and updates the cutoff whenever recursing into a field with a binding specification.

The variable case of the substitution operator is also generically implemented in the library. It is parameterized over the term datatype T for the sort that is substituted, the variable constructor $\mathit{var} : T \to T$ and the shift operation $\mathit{shift} : \mathit{nat} \to T \to T$.

Proof Boilerplate To reduce the implementation effort, INFRAGEN generates proof scripts rather than proof terms for boilerplate lemmas. These scripts are backed by dedicated tactics in the companion library that capture our knowledge of how such proofs proceed.

We have pushed the generic boilerplate for the variable case of lemmas into the library in the same manner as we did for syntax operations. The library contains a proof of the shift commutation lemma for indices. The full proof of commutation for shifts on terms

$$\forall c, \mathit{shift}\ (1+c) \circ \mathit{shift}\ 0 = \mathit{shift}\ 0 \circ \mathit{shift}\ c$$

is merely a congruence proof that can be proven by straightforward induction.

The library also contains two modules for generic proofs for the variable case of the commutation lemmas. The first one covers commutations between a shift and a substitution and is parameterized over three properties

1. The effect of shift on variables

$$\forall c, \mathit{shift}\ c \circ \mathit{var} = \mathit{var} \circ \mathit{shift}_{\mathbb{N}}\ c.$$

2. The commutation lemma for shift.
3. The effect of subst on variables

$$\forall x\ t, \mathit{subst}\ x\ t \circ \mathit{var} = \mathit{subst}_{\mathbb{N}}\ x\ t.$$

Using the first module the code generator will derive the commutation lemmas for terms which are additional inputs to the second module that derives the variable case of the commutation lemma for two substitutions.

6.2 Soundness

We have not formally established that INFRAGEN always generates type-correct code or that the proof scripts always succeed. Nevertheless, a number of important implementation choices bolster the confidence in INFRAGEN's correctness:

- Firstly, we have established that type-sound boilerplate definitions and provable boilerplate lemmas exist for every language specified with INFRA. This proof consists of the Coq formalization of INFRA and the corresponding generic boilerplate definitions and lemmas.
- Secondly, the generated functions and proofs follow the structure of the generic functions and proofs in the formalization. Thus the reasoning steps in the generated proofs are the appropriate steps to take in order to establish the desired properties.
- Thirdly, the variable cases of lemmas are generically handled in the companion library. This means that we have established their correctness once and for all. Furthermore, the variable case is usually the difficult case in the proofs, whereas all other cases are merely congruences. More specifically, most of the reasoning about arithmetic happens in the variable case.
- Finally and more pragmatically, we have implemented a test suite of INFRA specifications for INFRAGEN that contains a number of languages with advanced binding constructs including dependently-typed languages and languages with recursive scoping for which correct code is generated. This test suite is covered in more detail in Section 7.

Nevertheless, the above does not rule out trivial points of failure like name clashes between definitions in the code and the Coq standard library. Fortunately, when the generate code is loaded in Coq, Coq still performs a type soundness check to catch any issues; soundness never has to be taken at face value.

6.3 Binding Specification Design

While several design decisions for INFRA were made with INFRAGEN in mind, this is particularly true for the nature of binding specifications.

Binding specifications consist of nested lists of bound variables. Their grammar is set up in such a way that the structure of the outer heterogeneous list is always statically known. In contrast, the structure of the inner homogeneous lists may not be statically known when it is expressed in terms of auxiliary function calls which depend on runtime values. INFRAGEN exploits the statically known structure to fully inline the code for the heterogeneous lists, including the functions $\mathit{namespacesOf}$, lift_α, lift_s, $\mathit{shift'}_\alpha$ and $\mathit{subst'}_\alpha$. As a consequence we are left with homogeneous operations only.

While we have briefly considered to support only homogeneous binding specifications, we have quickly rejected them as inadequate. For instance GADT pattern matching in System FC [**?**] involves binding term, type and coercion variables simultaneously. Hence, an adequate formalization requires heterogeneous binding specifications.

Another alternative is to have flat heterogeneous lists as binding specifications. That allows arbitrarily interleaved binding of variables from different namespaces. However, to support binding an arbitrary number of variables, the structure of these lists cannot be known statically. This requires a dynamic representation. Because of modularity concerns the usual way to do it, is to define a traversal function that is parameterized over an operation for each namespace in $\mathit{namespacesOf}$. For example when traversing a heterogeneous binder of type and term variables we call a parameter function f for each bound term variable and a function g for each type variable. This is in effect a Church encoding of a heterogeneous variable list. The problem with this is reasoning: all of the commutation lemmas are strengthened with a heterogeneous variable list. In the variable case the lemma is proved by induction over that variable list. However, performing induction over Church encodings is problematic. One can still use this version by performing the reasoning over the patterns instead, i.e. establish the lemmas for the Church encoded lists by induction over the piece of data that created them. This has 2 big disadvantages. Firstly, each lemma now needs to be proved for every kind of binding specification separately, which is a considerable blow up in terms of code. Secondly, the variable case cannot be proven generically upfront. This is especially bad because the variable case contains the difficult arithmetic reasoning. This would undermine the correctness of the code generator.

In summary, our design of binding specifications sits in a sweet spot between flat homogeneous and flat heterogeneous lists. On the one hand, even though it forces the user to choose the order in which variables from different namespaces are bound, it is still highly expressive. On the other hand, it is still very suitable for reasoning; the only price to pay is the reasoning about commutation of operations where variables of different namespaces are bound.

6.4 Inductive relations

INFRAGEN also has some preliminary support for contexts and inductive relations. It is possible to declare the typing context of $F_\times$ like this

$$\begin{aligned}&\mathit{ctx}\ G := \\ &\quad \mid \epsilon\ \ : \mathit{empty} \\ &\quad \mid x\ : \mathit{etermvar}\ G\ T \\ &\quad \mid X : \mathit{etypevar}\ G\end{aligned}$$

A well-formed context has exactly one empty constructor without fields and several linear constructors that bind one variable in the namespace indicated before the constructor name. From such a declaration INFRAGEN derives predicates for insertion of variables into contexts, predicates for context lookups and weakening lemmas for context lookups. Furthermore, one can declare inductive relations where one can use substitutions of the 0 index and shifting with the 0 cutoff in the judgements. For languages that feature only single-variable binding, INFRAGEN generates code for the weakening lemmas of the relations.

7. Case study

To show the expressiveness of our specification language and the benefits of our approach we have performed a case study consisting of two parts. We have implemented a range of language specifications of languages with rich bindings forms and we have performed

mechanizations of type-safety proofs for 4 different languages that use the code generated by our tool INFRAGEN.

7.1 Language suite

The language specification that we have implemented are part of INFRAGEN's test suite. The binding forms that it covers includes pattern bindings, declaration lists with various scoping rules, the combination of pattern bindings with declaration lists and languages featuring intricate dependencies between namespaces and sorts like λLF and the calculus of constructions.

The number of lemmas generated lies in $O(m^2 n)$ where m is the number of namespaces and n is the number of syntactic sorts. For languages with two namespaces like λLF this amounts to about 1000 lines of Coq code for the variable binding boilerplate and additionally 300 lines of code for the definition of context lookups, insertions and weakening lemmas of λLF 's type system [?]. Nevertheless, the code for λLF is checked by Coq in less than 4 seconds on a modern laptop.

7.2 Mechanizations

We have performed mechanizations of type-safety proofs for 4 different languages:

- λ — the simply-typed lambda calculus,
- $F_\times$ — System F with products,
- $F_{<:}$ — System F with subtyping as in the POPLmark challenge, and
- $F_{<:,\times}$ — System F with subtyping, binary products and nested patterns (similar to the variant with records and record patterns featured in the POPLmark challenge).

For each language we have a formalization in Coq developed without any tool support and a version that uses the code generated by INFRAGEN. Table 2 gives a detailed overview of the code size of different parts for each language formalization. The column *Syntax* shows the size of the abstract syntax declaration, including binding specifications.

The useful definitions in the *Semantics* column are the evaluation rules, typing contexts and typing rules. The boilerplate in the semantic definitions are context lookups for the variable typing rule as well as shifting and substitution operators, that are necessary to define β-reduction and, if supported by the language, type application. As can be seen in Table 2 all semantic boilerplate is generated by INFRAGEN.

The interesting meta-theoretical lemmas in the type-safety proofs are pattern-matching definedness, value inversion lemmas and progress and preservation lemmas. For the languages with subtyping this also includes the reflexivity and transitivity of the subtyping relation. We separate the binder boilerplate that arises during the formalizations into three classes: term, context and relation related boilerplate.

Term related boilerplate consists of commutation lemmas for two consecutive syntactic operations that we discussed in Section 5. Note that the mechanization of λ does not make use of any commutation lemmas. In the other cases INFRAGEN derives all lemmas that are necessary for the formalizations. With one exception, these lemmas cover solely the operations on type variables. This is about 140 lines of code for each language. The size depends on the number of namespaces, the number of syntactic sorts and the dependency structure between them, which is roughly the same for these languages.

For contexts, the boilerplate are definitions and proofs related to context insertion and weakening of context lookup. INFRAGEN generates inductive relations characterizing an insertion of a single variable somewhere in the middle of a context and weakening of context lookups by insertion. For the simply-typed lambda calculus and $F_{<:}$ all necessary lemmas are generated.

What INFRAGEN does not yet generate are lemmas for context lookup weakening for concatenating contexts that are used in the languages with pattern-bindings. One can of course always define context concatenation and the necessary proofs generically. The difficulty lies in inferring when a context is really used as a context and when it is used as a pattern environment which is later used as a context extension.

Finally, the boilerplate related to relations are weakening lemmas and lemmas for preservation of typing under substitutions. INFRAGEN can derive the weakening lemmas for relations that only feature single variable binders. This covers the typing relation of the simply-typed lambda calculus and the subtyping and typing relation of $F_{<:}$. In these two cases we can strip away about 30% of the boilerplate. In the two remaining cases we do not cover the typing relation, so that the achieved saving is down to 10%.

7.3 Discussion

The language formalizations use only a fraction of the lemmas generated by INFRAGEN. For instance, for the simply-typed lambda calculus INFRAGEN generates about 250 lines of code for commutation lemmas which are not used in the formalization. However the proof of weakening for λLF typing rules makes use of all commutation lemmas except for the ones featuring two substitutions. Moreover, we expect that the lemmas of typing preservation under substitutions will use all generated lemmas.

8. Related and Future Work

Because there is a large body of work related to variable binding, we have to limit our discussion to work on specification languages for variable binding, and to systems and tools for reasoning about syntax with binders.

8.1 Specification Languages

The Ott tool [?] allows the definition of concrete syntax of programming languages and inductive relations on terms. Its binding specifications have inspired those of INFRA. While Ott generates datatype and function definitions for abstract syntax in multiple proof assistants, support for lemmas is absent.

The Cαml tool [?] defines a specification language for abstract syntax with binding specifications from which it generates OCaml definitions and substitutions. Types can be annotated with atoms and atoms occurring in terms are considered to be binders. *inner* and *outer* annotations let the users specify if subterms are inside or outside of an enclosing abstraction. For instance, the pattern bindings of $F_\times$ are specified by

$$\begin{array}{l}\textbf{type}\ term = \\ \quad \mid ELet\ \textbf{of} < pat * outer\ term * inner\ term > \\ \textbf{type}\ pat = \\ \quad \mid pvar\ \textbf{of}\ atom \\ \quad \mid pprod\ \textbf{of}\ pat * pat\end{array}$$

The angle brackets <> introduce an abstraction ranging over a pattern, a term outside of the abstraction and a term inside of the abstraction. The occurrences of atoms in the pattern are considered to be bound by the abstraction. It is unclear to us how the expressiveness of Cαml binding specifications relates to that of INFRA binding specifications. However, Cαml does not allow abstractions to be nested, which disallows the telescopic lambdas of Figure 6.

8.2 Generated Boilerplate

LNGEN Aydemir and Weirich [?] created LNGEN, a tool that generates locally-nameless Coq definitions from an Ott specifica-

		Syntax	Semantics		Theorems				Total Boilerplate	
			Useful	Boilerplate	Useful	Boilerplate				
						Terms	Contexts	Relations		
λ		10	74	43	140	0	18	61	122	
	IG	10	74	0	140	0	0	41	41	34%
$F_\times$		23	177	132	311	145	90	318	685	
	IG	23	177	0	311	0	36	279	315	45%
$F_{<:}$		13	113	113	413	135	66	229	543	
	IG	13	113	0	413	0	0	153	153	28%
$F_{<:,\times}$		24	187	134	577	146	102	338	720	
	IG	24	187	0	577	0	36	295	331	46%

Table 2. Size statistics of the meta-theory mechanizations.

tion. It takes care of boilerplate syntax operations, local closure predicates and lemmas. It supports multiple namespaces but restricts itself to single-variable binders.

DBGEN Polonowksi [?] developed the DBGEN tool that generates de Bruijn representations and boilerlate code. It supports multiple namespaces and has some support for binding multiple variables: one can specify that n variables are to be bound in a field where n is either a natural number literal or a natural number field of the constructor. The DBGEN definition of the destructuring pattern bindings of $F_\times$ looks like this:

$$\begin{array}{l} Let\ (n : nat)\ (p : pat)\ (t_1 : term) \\ \quad ((*bind\ n : term\ \mathbf{in}*)\ t_2 : term) \end{array}$$

where the annotation $(*bind\ n : term\ \mathbf{in}*)$ specifies that n variables are to be bound in the subterm t_2. The constraint that n equals the number of variables bound by p is established externally by a well-formedness relation.

Why3 Clochard et al. [?] support variable binding in the Why3 framework [?] using nested datatypes to represent binders [? ?]. They generate datatype definitions from a simple specification language that supports multiple namespaces and single-variable binders. Proof obligations are dispatched to automatic provers or proof-assistants and proof hints for variable binding are provided.

Unlike the above generative approaches, INFRAGEN's specification language INFRA has been mechanically formalized. formalized its specification language INFRA. Furthermore, our tool has the added benefit that the variable case is implemented generically and only the easy congruence cases are generated where the works presented above generate the complete boilerplate directly.

8.3 Generic Boilerplate

GMETA GMETA [?] is a framework for first-order representations of variable binding developed by Lee et al.. It is implemented as a library in Coq that makes use of datatype-generic programming concepts to implement syntactic operations and well-formedness predicates generically. The system supports multiple namespaces by comparing the generic representation of the associated term types but is restricted to the single-variable case. Moreover, GMETA contains a library for contexts of one or two sorts. In the case of two sorts, e.g. term and type variables, only the binding of term variables can depend on types and only the binding of type variables can be telescopic.

GMETA captures the structure of terms generically, but not the structure of contexts and the accompanying library implements two instances, but admittedly the ones that are used the most.

Unbound The UNBOUND library [?] is a Haskell library for programming with abstract syntax. It's specification language consists of a set of reusable type combinators that specify variables, abstractions, recursive and sequential scoping. The library internally uses a locally nameless approach to implement the binding boilerplate which is hidden from the user. The library also has a combinator called *Shift* which allows to skip enclosing abstractions. This form of non-linear scoping is not supported by INFRA.

8.4 Language Support

Several languages have direct support for variable binding. Nominal Isabelle [?] is an extension of the Isabelle/HOL framework with support for nominal terms which provides α-equivalence for free. At the moment the system is limted to single variable binding but support for richer binding structure is planned [?].

Logical frameworks suchs as Abella [?], Hybrid [?], Twelf [?] and Beluga [?] are specifically designed to reason abouts logics and programming languages. Their specialized meta-logic encourage the use of higher-order abstract syntax (HOAS) which represents object-level variable binding using the binding of the meta-language. The advantage is that facts about substitution and α-equivalence are inherited from the meta-language. HOAS approaches can represent a list of variable bindings directly or by transforming the abstract syntax and using an auxiliary datatype for binding lists as for example in

$$\begin{array}{l} base : term \to bterm. \\ bind : (term \to bterm) \to bterm. \end{array}$$

Using this we can represent unstructured binders. For structured binders such as patterns part of the binding specification needs to be addressed by an external relations such as a typing relation.

Keuchel and Jeuring [?] show how to generically restructure UNBOUND specifications without *Shift* but including recursive and sequential scoping into a HOAS representation and enforce the binding specification using dependent-types.

8.5 Comparison

Table 3 summarizes the features supported by the above languages and systems. The upper half of the table shows the support for the specification of rich binding forms. In this comparison we require that the abstract syntax can be specified in a natural way, without restructuring it to accommodate restrictions of the system. In terms of expressiveness Ott and INFRA coincide and with two exceptions subsume the support of all other systems. In particular INFRAGEN is more expressive than the other meta-theoretical frameworks

	Specification Languages			HOAS	First-Order Approaches			
	Ott	Cαml	UNBOUND		LNGEN	GMETA	DBGEN	INFRAGEN
Distinct namespaces	●	●	●	●	●	●	●	●
Unstructured binders	●	●	●	●	○	○	●	●
Heterogenous binders	◐	●	●	●	○	○	◐	◐
Structured binders	●	●	●	◐	○	○	◐	●
Recursive scoping	●	◐	●	○	○	○	◐	●
Sequential scoping	●	◐	●	○	○	○	◐	●
Non-linear scoping	○	?	●	○	○	○	○	○
Inductive relations	●	○	○	●	○	○	○	●
Renaming / shifting	○	○	○	●	●	●	●	●
Substitutions	○	○	○	●	●	●	●	●
Contexts	○	○	○	●	○	◐	○	●
Telescopic contexts	○	○	○	●	○	◐	○	●
Lookups	○	○	○	●	○	◐	○	●
Lookup weakening	○	○	○	●	○	◐	○	●

Table 3. Support of binding constructs in various systems

using first-order approaches. As discussed in Section 6.3, the restricted support for heterogeneous binders is a deliberate choice to simplify reasoning. Conceptually, adding supporting for heterogeneous binders to INFRA is easy. The other notable exception is the unusual non-linear scoping that is only supported by UNBOUND. In all other systems including INFRA every subterm lies in an extension of its parent's scope. To add support for non-linear scoping to INFRA, we have to either break this assumption or allow multiple scopes for terms. However, in contrast to UNBOUND our work involves reasoning about variable binding and both directions seem to impose difficulties.

The lower half of the table lists how good the systems are at removing variable binding boilerplate including properties of the operations. Notably the frameworks based on HOAS get rid of all the boilerplate by inheriting the properties from the meta-language. As shown by our case study INFRAGEN gets rid of all the boilerplate at the level of terms and most of the boilerplate for contexts. The remaining boilerplate for contexts and that for inductive relations is not covered by any of the other first-order systems either; we leave it as an important open challenge.

9. Conclusion

This paper has presented INFRAGEN, a generative approach to variable binding boilerplate backed by a generic formalization. INFRAGEN distinguishes itself from earlier work on two accounts. Firstly, it covers a wider range of binding constructs featuring rich binding forms and advanced scoping rules. Secondly, it covers a larger extent of the boilerplate functions and lemmas needed for the mechanization of programming languages.