January 2009

Abstracting Syntax

Brian Aydemir
University of Pennsylvania

Stephan A. Zdancewic
University of Pennsylvania, stevez@cis.upenn.edu

Stephanie Weirich
University of Pennsylvania, sweirich@cis.upenn.edu

Recommended Citation

Brian Aydemir, Stephan A. Zdancewic, and Stephanie Weirich, "Abstracting Syntax", . January 2009.

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-09-06

Abstracting Syntax

Abstract

Binding is a fundamental part of language specification, yet it is both difficult and tedious to get right. In previous work, we argued that an approach based on locally nameless representation and a particular style for defining inductive relations can provide a portable, transparent, lightweight methodology to define the semantics of binding. Although the binding infrastructure required by this approach is straightforward to develop, it leads to duplicated effort and code as the number of binding forms in a language increases.

In this paper, we critically compare a spectrum of approaches that attempt to ameliorate this tedium by unifying the treatment of variables and binding. In particular, we compare our original methodology with two alternative ideas: First, we define variable binding in the object language via variable binding in a reusable library. Second, we present a novel approach that collapses the syntactic categories of the object language together, permitting variables to be shared between them.

Our main contribution is a careful characterization of the benefits and drawbacks of each approach. In particular, we use multiple solutions to the POPLMARK challenge in the Coq proof assistant to point out specic consequences with respect to the size of the binding infrastructure, transparency of the definitions, impact to the metatheory of the object language, and adequacy of the object language encoding.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-09-06

Abstracting Syntax

Brian Aydemir Stephanie Weirich Steve Zdancewic
University of Pennsylvania
{baydemir,sweirich,stevez}@cis.upenn.edu

Technical Report MS-CIS-09-06
March 2009

Abstract

Binding is a fundamental part of language specification, yet it is both difficult and tedious to get right. In previous work, we argued that an approach based on locally nameless representation and a particular style for defining inductive relations can provide a portable, transparent, lightweight methodology to define the semantics of binding. Although the binding infrastructure required by this approach is straightforward to develop, it leads to duplicated effort and code as the number of binding forms in a language increases.

In this paper, we critically compare a spectrum of approaches that attempt to ameliorate this tedium by unifying the treatment of variables and binding. In particular, we compare our original methodology with two alternative ideas: First, we define variable binding in the object language via variable binding in a reusable library. Second, we present a novel approach that collapses the syntactic categories of the object language together, permitting variables to be shared between them.

Our main contribution is a careful characterization of the benefits and drawbacks of each approach. In particular, we use multiple solutions to the PoplMark challenge in the Coq proof assistant to point out specific consequences with respect to the size of the binding infrastructure, transparency of the definitions, impact to the metatheory of the object language, and adequacy of the object language encoding.

1 Introduction

Binding is a fundamental part of programming language specification, yet it is both difficult and tedious to get it right. When you begin to specify a programming language formally in your favorite proof assistant, the last thing you want to do is prove properties about substitution, alpha equivalence, etc. For this reason, there are proof assistants carefully tailored to help with binding (e.g., Nominal Isabelle [16] and Twelf [11]) and tools to help you get started (e.g., Ott [14] and Hybrid [9]). In previous work, we argued that a combination of a locally nameless representation (described in Section 2.1) with a particular style for defining inductive relations provides a portable, transparent, lightweight methodology to define the semantics of binding [1]. These proof assistants, tools, and approaches each address binding throughout the entire scope of a metatheory development—from defining syntax and relations to proving properties by induction on those definitions.

In this paper, we focus on a central aspect of the definition of binding: capture-avoiding substitution for bound and free variables, and proofs of its essential properties. In general, we need to define substitution and prove its properties for each kind of bound term appearing in each syntactic class. For example, in the PoplMark Challenge [2], we need three substitution operations: Types in System $F_{<:}$ bind types, and expressions bind both types and expressions. This means that part of our development is duplicated three times, once for each substitution operation. For larger languages, with increasingly many kinds of binding forms, the overhead associated with substitution becomes increasingly unpleasant.

Our main contribution is a detailed comparison of the costs and benefits of several variations of syntax encodings, each intended to reduce the burden associated with defining and reasoning about substitution. These variations live on a spectrum of representation strategies: At one end is the locally nameless approach proposed by our previous work. At the other end is an approach reminiscent of higher-order abstract syntax (HOAS) [10], in which a reusable library provides a meta-language (the lambda calculus) for representing the binding structure of an object language. In between are forms of "collapsed" syntax, in which binding structures (and hence syntactic categories) of the object language are identified; we believe that these collapsed syntax representations are a second contribution of this work.

Operationally, we carry out this analysis by using five[1] different encodings for syntax, each described in Section 2, in solutions to the PoplMark Challenge using the Coq proof assistant.[2] With the exception of one variant that requires indexed and dependent types, these encodings should be implementable in any proof assistant that supports inductive datatypes. In Section 3, we compare the encodings along two lines: by the degree to which they reduce the infrastructure associated with substitution, and by the extent and nature to which they change a metatheory development. To facilitate our comparisons, we keep the developments as similar as possible to each other, restricting changes to those aspects that arise from the syntax representation used. For reasons of practicality, we cannot include here direct comparisons to *all* encodings of variable binding. We do, however, discuss in Section 4 other closely related encodings, e.g., HOAS and nominal approaches, along with other related work. We summarize our conclusions and discuss future work in Section 5.

2 Approaches to abstraction

In this section, we describe in detail several approaches for encoding syntax, working from locally nameless to various forms of "collapsed" syntax. We use System $F_{<:}$, the language of the PoplMark Challenge, as a running example of an object language that we wish to encode. This language has two syntactic categories, types and expressions, both of which include binding constructs. The syntax is not mutually recursive: while types may appear in expressions, expressions may *not* appear in types. For reference, we show below the syntax of types and expressions in System $F_{<:}$.

$$\begin{array}{rcll} \text{types} & T & ::= & \top \mid X \mid T_1 \to T_2 \mid \forall X{<:}T_1.\, T_2 \\ \text{expressions} & e & ::= & x \mid \lambda x{:}T.\, e \mid e_1\; e_2 \mid \Lambda X{<:}T.\, e \mid e\;[T] \end{array}$$

We have implemented System $F_{<:}$ in Coq and proved type safety for the language using each of the approaches we describe below. Our implementations also include sum types (with case analysis) and let expressions, constructs that we omit from our running example for brevity. Where possible, we use standard mathematical notations for inductive definitions in the descriptions that follow, using Coq's concrete syntax only in cases requiring indexed datatypes or dependent types.

2.1 Locally nameless

Locally nameless representation is a first-order approach that encodes bound variables using de Bruijn indices and free variables using names [12]. Thus, alpha-equivalent terms are syntactically equal, and there is no need to place an arbitrary ordering on free variables, as with a pure de Bruijn representation. In this section, we give a brief overview of this approach.

We present below the syntax of System $F_{<:}$ in a locally nameless representation. By each constructor, we put in parentheses the System $F_{<:}$ term it corresponds to.

[1] In the course of our research, we actually implemented several additional variants whose properties were too similar to those we selected to present here.

[2] Our Coq code may be found at `http://www.cis.upenn.edu/~baydemir/`.

Operation or judgement	Description
$\{n \mapsto T_1\}T_2$	index substitution of T_1 for index n in T_2
$\{n \mapsto T\}e$	index substitution of T for index n in e
$\{n \mapsto e_1\}e_2$	index substitution of e_1 for index n in e_2
$[x \mapsto T_1]T_2$	name substitution of T_1 for x in T_2
$[x \mapsto T]e$	name substitution of T for x in e
$[x \mapsto e_1]e_2$	name substitution of e_1 for x in e_2
$\vdash T$	judgement that T is locally closed
$\vdash e$	judgement that e is locally closed
$\Gamma \vdash_{\mathsf{typ}} T$	judgement that T is well-formed in Γ
$\Gamma \vdash_{\mathsf{exp}} e$	judgement that e is well-formed in Γ

Figure 1: Infrastructure for the locally nameless representation. Here, Γ is a mapping from variable names to their sorts—either "type variable" or "expression variable."

$$\begin{array}{rcll@{\qquad}rcll}
T & ::= & \mathsf{typ\_bvar}\ i & \text{(bound var.)} & e & ::= & \mathsf{exp\_bvar}\ i & \text{(bound var.)} \\
 & | & \mathsf{typ\_fvar}\ x & \text{(free var.)} & & | & \mathsf{exp\_fvar}\ x & \text{(free var.)} \\
 & | & \mathsf{typ\_top} & (\top) & & | & \mathsf{exp\_abs}\ T\ e & (\lambda x{:}T.\ e) \\
 & | & \mathsf{typ\_arrow}\ T_1\ T_2 & (T_1 \to T_2) & & | & \mathsf{exp\_app}\ e_1\ e_2 & (e_1\ e_2) \\
 & | & \mathsf{typ\_all}\ T_1\ T_2 & (\forall X{<:}T_1.\ T_2) & & | & \mathsf{exp\_tabs}\ T\ e & (\Lambda X{<:}T.\ e) \\
 & & & & & | & \mathsf{exp\_tapp}\ e\ T & (e\ [T])
\end{array}$$

The constructors exp_tabs, exp_abs, and typ_all do not name the variables they bind since bound variables are represented with de Bruijn indices. We use only one sort for variables names (denoted by the metavariable x), as opposed to separate sorts for type and expression variables (as in a traditional paper presentation). Conflating these sorts of variables does not change the correctness of our representation and, more importantly, simplifies comparisons with the representations discussed below.

As an example term in this representation, the polymorphic identity function, $\Lambda X{<:}\top.\ \lambda y{:}X.\ y$, is written here as

$$\mathsf{exp\_tabs}\ \mathsf{typ\_top}\ (\mathsf{exp\_abs}\ (\mathsf{typ\_bvar}\ 0)\ (\mathsf{exp\_bvar}\ 0))\,.$$

The two occurrences of "0" in the identity function point to different binding occurrences. The 0 in (typ_bvar 0) refers to the variable bound by exp_tabs; the 0 in (exp_bvar 0) refers to the variable bound by exp_abs.

In the remainder of this section, we describe the infrastructure for substitution and induction associated with this locally nameless representation. While our focus is on substitution, the treatment of induction varies between the representations we consider below. It is, therefore, an interesting point for comparison. The infrastructure is summarized in Figure 1.

Substitution Because there are two kinds of variables (bound and free) in this representation, there are two kinds of substitution operations. Index substitution replaces a bound variable with another term, and name substitution replaces a free variable with another term. We need to define three versions of each operation due to the syntax of System $\mathrm{F}_{<:}$: types may be bound in types, types may be bound in expressions, and expressions may be bound in expressions. This results in a total of six functions, all of which are straightforward to define by induction on syntax. Lemmas concerning the properties of substitution, such as the fact that substituting for a name fresh for a term leaves the term unchanged, are straightforward to prove. Such properties are proved for each version of substitution.

The key points here are not the particular definitions of substitution or the properties we need, but that we need *three* versions of each substitution operation, each with its own lemmas. The three versions arise solely from the definition of System $\mathrm{F}_{<:}$ (specifically, its binding forms), not our use of a locally nameless representation.

Induction The inductive definitions for types and expressions, as stated above, include terms, such as (typ_bvar 3), that contain dangling indices. Thus, the induction principles for those datatypes do not correspond to the usual induction principles for syntax: they require one to consider cases for bound variables (indices), which have no correspondence to informal practice. One solution is to define a predicate that holds when a term is *locally closed*—that is, when all its indices point to binders within the term.[3] We need two judgements for local closure, one each for types and expressions. As an example, the definition of locally closed types is shown below.

$$\frac{}{\vdash \mathsf{typ\_fvar}\ x} \qquad \frac{}{\vdash \mathsf{typ\_top}} \qquad \frac{\vdash T_1 \qquad \vdash T_2}{\vdash \mathsf{typ\_arrow}\ T_1\ T_2} \qquad \frac{\vdash T_1 \qquad \vdash \{0 \mapsto \mathsf{typ\_fvar}\ x\}T_2}{\vdash \mathsf{typ\_all}\ T_1\ T_2}$$

Induction on a derivation of $\vdash T$ corresponds to informal structural induction on the structure of T.[4] There is no case for bound variables, since they are never locally closed.

However, in order to prove that the representation here defines the same language as the representations below, we need also to rule out terms where the same name is used as both a type variable and an expression variable. For example, in the term (exp_abs (typ_fvar b) (exp_fvar b)), the name b is reused in this way. Our particular formalization of System $\mathrm{F}_{<:}$ also does not treat such terms as being meaningful, e.g., they are never typeable. We rule out these terms by defining a predicate, called *well-formedness*, that ensures both that a term is locally closed and that its free variables are used in a well-sorted way with respect to an environment declaring their sorts. As with local closure, we define one well-formedness judgement each for types and expressions. We choose to define these judgements in such a way that they also provide structural induction principles for syntax. This simplifies comparisons to the approaches below because for those representations, well-formedness will sometimes be the only means of reasoning by structural induction on syntax. Here, in our locally nameless representation, when we need to reason by structural induction on syntax, we have a choice and prefer to use local closure.

Strictly speaking, for our locally nameless representation, we do not need to define both local closure and well-formedness, since well-formedness is sufficient to define System $\mathrm{F}_{<:}$ and prove type safety. However, local closure by itself is enough to prove lemmas about index and name substitution. Statements of these lemmas are slightly simpler than with well-formedness, since they do not need to mention an arbitrary environment.

2.2 Collapsed syntax

As stated in the introduction, we need a definition of substitution and proofs of its properties for each kind of bound term appearing in each syntactic class. Thus, the number of definitions and proofs required scales with the size of the language. We can mitigate this scaling issue by collapsing multiple syntactic categories into one single category, called simply *syntax*. This is similar to the collapsed syntactic categories of Pure Type Systems [3]. We give below the definition of System $\mathrm{F}_{<:}$ in this style, highlighting the changes from our original definition.

$$\begin{array}{llclcl}
\text{syntax} & s, T, e & ::= & \mathsf{bvar}\ i & \mid & \mathsf{fvar}\ x \\
 & & \mid & \mathsf{typ\_top} & \mid & \mathsf{exp\_abs}\ T\ e \\
 & & \mid & \mathsf{typ\_arrow}\ T_1\ T_2 & \mid & \mathsf{exp\_app}\ e_1\ e_2 \\
 & & \mid & \mathsf{typ\_all}\ T_1\ T_2 & \mid & \mathsf{exp\_tabs}\ T\ e \\
 & & & & \mid & \mathsf{exp\_tapp}\ e\ T
\end{array}$$

This definition makes no distinction between types and expressions, and we retain only one constructor each for bound and free variables.[5] Similarly, we combine the definitions of the local closure predicates for

[3] McKinna and Pollack first used this technique in their formalization of Pure Type Systems [8].

[4] As the definition is stated here, the resulting induction principle is actually comparatively weak. Aydemir et al. [1] discuss how to take this idea and strengthen it, which is what we do in the Coq code.

[5] We could further collapse the binding forms by adding to the collapsed syntax an additional constructor (abs e). In this case, abs would be the only constructor to bind a variable. For example, we would write here (exp_abs T (abs e)) instead of (exp_abs T e). Collapsing binding forms in this way does not change developments in any other significant way. Thus, we say nothing more about this approach.

Operation or judgement	Description
$\{n \mapsto s_1\}s_2$	index substitution of s_1 for index n in s_2
$[x \mapsto s_1]s_2$	name substitution of s_1 for x in s_2
$\vdash s$	judgement that s is locally closed
$\Gamma \vdash_{\mathsf{typ}} s$	judgement that s is a well-formed type in Γ
$\Gamma \vdash_{\mathsf{exp}} s$	judgement that s is a well-formed expression in Γ

Figure 2: Infrastructure for the collapsed representation.

types and expressions into a single predicate on syntax. As an example term, in this representation, the polymorphic identity function is written

$$\mathsf{exp\_tabs}\ \mathsf{typ\_top}\ (\mathsf{exp\_abs}\ (\mathsf{bvar}\ 0)\ (\mathsf{bvar}\ 0))\ .$$

Compared to our locally nameless encoding for this term, the primary difference concerns the bound variables: they are both constructed using the same constructor.

The advantage of this collapsed representation over our locally nameless one is in the reduction of infrastructure, which is summarized in Figure 2. Instead of three versions of substitution, as in the locally nameless representation, we need only one version: substitution of syntax in syntax. This results in a significant reduction in the number of lemmas that we must prove about substitution and is the main source of the reduction in infrastructure.

However, this collapsed datatype encodes the syntax of System $F_{<:}$ in an essentially untyped manner: there are no restrictions on where types and expressions may appear in terms. For example, the datatype includes terms of the form $(\mathsf{exp\_abs}\ \mathsf{typ\_top}\ (\mathsf{typ\_all}\ \ldots))$, where a type appears in a location where an expression should be. Local closure cannot rule out such terms, since it is defined for the entire syntax datatype. Consequently, local closure also does not provide an effective induction principle over types and expressions as it includes too many cases for each. The distinctions and induction principles we need are provided by the definitions of well-formed types and expressions. Thus, we still need to define well-formedness predicates as we did for the locally nameless encoding.

Tagged syntax In a proof assistant that supports indexed and dependent types, we can define syntax as an *indexed* datatype that distinguishes types from expressions.[6] Here, the index is used to capture the recursion between multiple datatypes. Specifically, the datatype for syntax is indexed by a tag that indicates the original syntactic category that each constructor came from. Because indexed and dependent types play a crucial role in this formulation, we give the definitions below using a pretty-printed form of Coq's concrete syntax. (Coq uses "∀" for writing dependent types.)

For System $F_{<:}$, we define a datatype `tag` with two zero-argument constructors, `Typ` and `Exp`.

```
Inductive tag : Set :=  Typ : tag  |  Exp : tag.
```

With tags defined, we define the indexed datatype for syntax.

[6] This idea is due to Randy Pollack (personal communication).

```
Inductive syntax : tag → Set :=
  | bvar      : ∀ T:tag, nat → syntax T
  | fvar      : ∀ T:tag, atom → syntax T
  | typ_top   : syntax Typ
  | typ_arrow : syntax Typ → syntax Typ → syntax Typ
  | typ_all   : syntax Typ → syntax Typ → syntax Typ
  | exp_abs   : syntax Typ → syntax Exp → syntax Exp
  | exp_app   : syntax Exp → syntax Exp → syntax Exp
  | exp_tabs  : syntax Typ → syntax Exp → syntax Exp
  | exp_tapp  : syntax Exp → syntax Typ → syntax Exp.
```

Above, `syntax` is a type constructor that takes one argument: a `tag`. We use `syntax Typ` as the type of System $F_{<:}$ types and `syntax Exp` as the type of System $F_{<:}$ expressions. Under this reading, all the constructors except for `bvar` and `fvar` construct types and expressions exactly as they did in the original locally nameless representation. The constructors for bound variables and free variables are polymorphic, since they can be used to construct both types and expressions. Except for the `tag` arguments to `bvar` and `fvar`, terms are written in this encoding exactly as they were in the original collapsed encoding. The types of the constructors rule out terms, such as (exp_abs typ_top (typ_all ...)), that do not respect where types and expressions may appear. This is the key advantage of using this tagged representation over the original collapsed representation.

In this variation, we still need to define only one version each of index substitution and name substitution (the same as in the original collapsed encoding). Thus, we retain the reduction in infrastructure of the collapsed encoding. Local closure for syntax must still be defined, but it is now a dependently typed judgement: it takes a tag `t` and a term of type `syntax t`. Therefore, we can distinguish between locally closed types and expressions by supplying a tag to the local closure judgement. Local closure for the untagged collapsed representation could *not* make such a distinction. The ability to make this distinction gives us structural induction principles for System $F_{<:}$ types and expressions. As before, we still need to define well-formed types and expressions in order to rule out terms with reused variable names.

2.3 Reusable lambda calculus library

We now turn to an approach inspired by HOAS and Twelf [11]. In Twelf, the Edinburgh Logical Framework (LF [7]) is used as a meta-language for representing deductive systems, where the systems are typically encoded using HOAS. Twelf adds to LF a separate meta-logic to reason about the metatheoretic properties of such encodings. In our work, we implement a library that encodes a simple language to represent binding (taking the place of LF), proving the properties of that language once and for all, and separately from the semantics of the object language. Below, we consider two languages for representation: an untyped lambda calculus and a simply-typed lambda calculus. Unlike with LF, we use these languages only for representing syntax and do not represent the typing rules or operational semantics of System $F_{<:}$ in this way.

2.3.1 Untyped lambda calculus

In our first version of the library, we implement the untyped lambda-calculus extended with a constructor for constants using a locally nameless encoding.

$$\text{lambda terms} \qquad t \quad ::= \quad \mathsf{const}\ C \mid \mathsf{bvar}\ i \mid \mathsf{fvar}\ x \mid \mathsf{abs}\ t \mid \mathsf{app}\ t_1\ t_2$$

This definition is parameterized over a syntactic category C of constant symbols. When using this language as a meta-language, constant symbols represent the names of constructors in an object language. Because constant symbols do not bind anything nor contain any variables (free or bound), we can define, once and for all, index substitution and name substitution over this datatype and prove their properties. We can also define, once and for all, local closure for this datatype.

Operation or judgement	Description
$\Gamma \vdash_{\mathsf{typ}} s$	judgement that s is a well-formed type in Γ
$\Gamma \vdash_{\mathsf{exp}} s$	judgement that s is a well-formed expression in Γ

Figure 3: Infrastructure (specific to System $F_{<:}$) for the representations based on a lambda calculus library.

In order to encode System $F_{<:}$ using this untyped lambda calculus, we first define a datatype of constant symbols corresponding to the constructors of types and expressions in System $F_{<:}$.

$$\begin{array}{rcl} C & ::= & \mathsf{typ\_top\_c} \mid \mathsf{typ\_arrow\_c} \mid \mathsf{typ\_all\_c} \\ & \mid & \mathsf{exp\_abs\_c} \mid \mathsf{exp\_app\_c} \mid \mathsf{exp\_tabs\_c} \mid \mathsf{exp\_tapp\_c} \end{array}$$

Second, we instantiate the lambda-calculus definition with this datatype. We can write System $F_{<:}$ terms via explicit applications (apps) of constant symbols to their arguments, using abs to encode binding. To illustrate how the encoding works, the following table illustrates how the constructors of our originally locally nameless representation of System $F_{<:}$ types are encoded here.

Locally nameless	Corresponding lambda term
typ_bvar i	bvar i
typ_fvar x	fvar x
typ_top	const typ_top_c
typ_arrow T_1 T_2	app (app (const typ_arrow_c) T_1) T_2
typ_all T_1 T_2	app (app (const typ_all_c) T_1) (abs T_2)

The encoding for expressions is similar. In this representation, the polymorphic identity function is written as

$$\begin{array}{l} \mathsf{app\ (app\ (const\ exp\_tabs)\ (const\ typ\_top))} \\ \quad \mathsf{(abs\ (app\ (app\ (const\ exp\_abs)\ (bvar}\ 0))\ \mathsf{(abs\ (bvar}\ 0)))) \, . \end{array}$$

The infrastructure needed specifically for System $F_{<:}$ using this representation is summarized in Figure 3. We get the definitions of index and name substitution, and local closure on System $F_{<:}$ terms "for free" since the terms are encoded into the library's lambda calculus implementation, where they are already defined. This is a significant savings in the infrastructure needed specifically to define the language. However, because the local closure predicate in the library is defined over the structure of lambda terms in the library, it does not track the structure of System $F_{<:}$ terms. Thus, it does *not* provide a structural induction principle over System $F_{<:}$ types and expressions. We obtain suitable induction principles by defining well-formedness predicates. (We also need to define well-formedness in order to rule out terms with reused variable names.)

2.3.2 Typed lambda calculus

In our second version of the library, we implement a typed lambda calculus. The language of terms is the same as in the library above. We add to that library a definition for sorts parameterized over a set of base sorts A. We use these sorts as the types of the lambda calculus terms.

$$\text{sorts} \qquad S \ ::= \ \mathsf{lt\_base}\ A \mid \mathsf{lt\_arrow}\ S_1\ S_2$$

We then define a typing judgement $\Gamma \vdash_{\Sigma} t : S$ on these lambda-calculus terms in the usual way for a locally nameless encoding. (The environment Γ maps variable names to sorts.) The relation is parameterized by a signature Σ which assigns sorts to constant symbols.

To encode System $F_{<:}$ using this typed lambda calculus, we define a datatype of constant symbols corresponding to the constructors of types and expressions in System $F_{<:}$, as we did with the untyped lambda calculus library. We also need to define a set of base sorts and a signature. For the base sorts, we take

$$A \ ::= \ \mathsf{Typ} \mid \mathsf{Exp} \, .$$

We define the signature such that the sort assigned to each base constant is reminiscent of the type it would have under a HOAS encoding. For example, the sort we assign to typ_all_c is

$$\mathsf{lt\_arrow}\ (\mathsf{lt\_base}\ \mathsf{Typ})\ (\mathsf{lt\_arrow}\ (\mathsf{lt\_arrow}\ (\mathsf{lt\_base}\ \mathsf{Typ})\ (\mathsf{lt\_base}\ \mathsf{Typ}))\ (\mathsf{lt\_base}\ \mathsf{Typ}))\,.$$

The infrastructure needed specifically for System $\mathrm{F}_{<:}$ using this representation is the same as with the untyped library (c.f. Figure 3). We get the definitions of index and name substitution, and local closure "for free" from the library. Local closure from the library still does not provide a useful induction principles over the syntax of System $\mathrm{F}_{<:}$ terms. As before, we define well-formedness to obtain such induction principles (and to rule out terms with reused variable names).

3 Comparison

Having described several approaches to representing syntax, we turn now to more detailed comparisons. We use our Coq implementations of System $\mathrm{F}_{<:}$ in each approach to collect both quantitative data (reduction in substitution infrastructure, changes to metatheory proofs, adequacy) as well as qualitative data (i.e., anecdotal evidence).

The locally nameless implementation served as the starting point for the other implementations. To implement System $\mathrm{F}_{<:}$ in the other approaches, we copied the locally nameless implementation and then modified the copy. We were careful to ensure that differences between our developments directly reflect the differences in the representations used. Since we did not rework the soundness proofs from scratch, there is a small possibility that we missed opportunities to simplify those proofs. However, our main focus is on the definitions and infrastructure code, e.g., proofs about substitution.

3.1 Substitution infrastructure

The easiest, and perhaps simplest, comparison is to examine the sizes of each of our implementations. The data are shown in Figure 4. Each implementation is split into three parts.

- Definitions: Contains the definitions for System $\mathrm{F}_{<:}$, including its syntax, relations, operations over syntax (e.g., index and name substitution), and induction principles for syntax (e.g., well-formedness).
- Infrastructure: Contains the proofs of properties of substitution.
- Soundness: Contains the proof of type-safety for System $\mathrm{F}_{<:}$.

Our line counts include definitions, statements of theorems, and tactic scripts used to generate proofs of theorems. We exclude definitions of tactics and notations, except for the library-based variants, where we include notations that are required to write terms in a familiar way.

The differences in the sizes of the definition and infrastructure parts of each development follow directly from the differences in the representations used. For example, the collapsed and tagged representations need only one version of substitution, compared with three for the locally nameless representation. The library-based representations obtain these definitions directly from their respective libraries. The representation based on the typed lambda calculus library requires more definition lines than the untyped variant since it defines a signature for the base constants.

Overall, the library-based implementations are shortest, with a 18–23% reduction in code size compared to the locally nameless development. The collapsed and tagged developments, while longer than the library-based implementations, still result in a noticeable 14% reduction in code size.

3.2 Changes to metatheory proofs

Figure 4 also indicates that as we move from the locally nameless development to the library-based ones, the soundness proofs become longer. The presence or absence of one particular lemma—which states that

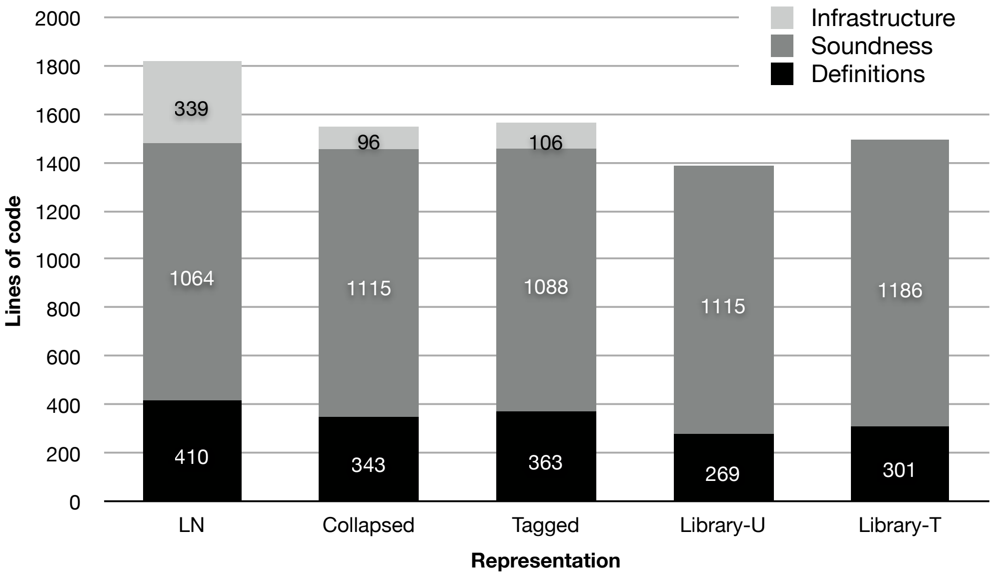

Figure 4: Lines of code in each of our Coq developments. LN is the locally nameless implementation. Library-U and Library-T are the implementations based on the untyped and typed lambda calculus libraries.

substituting for an expression variable inside a type leaves the type unchanged—accounts for most of this increase. In the locally nameless development, this fact is immediately evident; the lemma need not even be stated. With the tagged representation, this lemma requires a short and trivial proof. With the other representations, the proof is noticeably longer but still straightforward. The soundness proofs in the library-based implementations are also slightly longer due to additional steps that make it easier for Coq's automated tactics to find proofs.

3.3 Adequacy

All the representations of System $F_{<:}$ described above (in Section 2) define the same language, and we prove this formally within Coq. Were this not the case, our comparisons would be significantly less meaningful. We would have to account for the fact that some differences between the formalizations may reflect differences in the languages and not the representations being used.

More precisely, we prove that the collapsed and library-based representations are *adequate* with respect to the locally nameless representation. An encoding of syntax (e.g., for types and expressions) is adequate when there is a bijection between the terms of the object language and its encoding such that substitution commutes with the bijection [7]. Formally, we give a bijection between the well-formed terms of the locally nameless representation and each of the other representations. (We cannot show that substitution commutes with the bijection if we consider terms that are merely locally closed.) We prove that a locally nameless term type checks (according the typing relation of System $F_{<:}$) if and only if its corresponding term in the collapsed representation type checks. We also prove similar facts about reduction and subtyping, and with respect to the other representations.

3.4 Effects on metatheory development

Turning away from quantitative aspects of our developments, we report on several qualitative aspects of our developments: How easy are terms to read? Where are errors in definitions caught? Are proofs easy to carry out? We consider each of these questions in turn.

How easy are terms to read? We find locally nameless representations to be relatively easy and straightforward to read for two reasons: free variables are named, and there is little of the noise associated with shifting of indices in a pure de Bruijn representation. With respect to the definitions of System $\mathrm{F}_{<:}$'s subtyping and typing relations and the soundness proofs, the collapsed and tagged representations look almost exactly like a locally nameless representation. This is not surprising: they are based on the same general idea! The only difference is whether there are multiple constructors for variables or only a single one. If we use implicit coercions that automatically insert applications of constructors for variables where needed, there is no visible difference between these approaches and locally nameless.

The library-based approaches, however, are noticeably less easy to read in their raw form. Recall the example of the polymorphic identity function in Section 2.3. The term is unreadable due to all the nested instances of app. In practice, we found that the only way to completely address this problem was to define notations that enabled us to read and write terms as we would in the locally nameless representation; the notations effectively look and act like the constructors of a datatype. One still has to check that these notations are defined correctly. This involves either visually inspecting the definitions or verifying that standard properties of an inductive datatype hold (e.g., injectivity and distinctness of constructors).

Where are errors in definitions caught? There are several kinds of errors that may happen when defining a new language. At the level of syntax, we consider two kinds of typographical mistakes: arity and sort errors. We may apply a constructor to an incorrect number of arguments (an arity error) or to arguments of the wrong type (a sort error).

With a locally nameless representation or tagged representation, both arity and sort errors are caught at definition time (i.e., when definitions are processed by the proof assistant), because of the typed nature of definitions. With a collapsed representation, since multiple syntactic categories are implemented in the same inductive datatype, only arity errors can be caught at definition time. For example, the term (exp_abs typ_top) would be caught at definition time, but not (exp_abs typ_top (typ_all ...)). Sort errors can be caught only by verifying that the objects in which terms appear satisfy certain properties, e.g., that they contain only well-formed terms. For the library-based approaches, no sort errors and only some arity errors will be caught at definition time. This is true even for the typed lambda calculus library: the sort checking of terms is at the level of the lambda calculus encoding, not the proof assistant's type system. For example, both (const exp_tapp_c) and (app (const exp_tapp_c) (const typ_top)) will be allowed at definition time, even though neither corresponds to a System $\mathrm{F}_{<:}$ type application.

Errors other than arity and sort errors tend to be of a more semantic nature, e.g., swapping two terms of same sort or defining an unsound typing rule. No representation for syntax can catch such errors.

Are proofs easy to carry out? For System $\mathrm{F}_{<:}$, we were able to port proofs from one implementation to another with only minimal changes to the already written parts of the soundness proof. Because we copied proofs in this manner, we did not explicitly test the ability to generate proofs from scratch. However, once the infrastructure for a language is in place, proof development should be equally easy across all the representations. We note that while a collapsed representation does not check as much at definition time as a locally nameless representation, our experience with using a collapsed representation for other developments suggests that errors in encoding syntax are easily caught and corrected during proof development. We expect that the same would be true for the library-based representations.

It is worth noting that some of Coq's tactics do not handle indexed datatypes, such as the one used by the tagged representation, in a completely natural manner. In order to prove some properties, e.g., injectivity of index substitution, we proved by-hand the desired elimination principles.

4 Related work

Below, we discuss related approaches to those presented here. We do not attempt to survey all representation techniques for binding. See Aydemir et al. [1] for a comparison.

Our collapsed representation is novel to this paper, but inspired by Pure Type Systems (PTS) [3]. PTS elegantly combine all syntactic forms for a dependently-typed language into a single syntax, and collapse the typing relation into a single judgment on that semantics. Our collapsed version does the former, but not the latter. By collapsing the semantics as well as the syntax, PTS simplifies the metatheory of substitution for that semantics. Only one substitution lemma need be proved. However, the collapsed static semantics can make the language definition less transparent. Furthermore, it is not clear that combining judgements is beneficial to languages such as System $F_{<:}$, where there is little structural similarity between different judgements. For example, we may need to prove substitution only once, but we may have to consider as many cases as with several proofs about several different judgements.

Others have also proposed using a library or tool to provide generic support for representing and reasoning about binding. The Nominal Isabelle package [16] provides substantial support for defining datatypes, functions, and relations involving binding, and for reasoning about them. The collapsed representations and reusable library studied here are also applicable to Nominal Isabelle, since it does not automatically define substitution functions and their properties. The benefits would be less than in our locally nameless setting, since a number (but not all) of the properties we need to prove here are specific to a locally nameless encoding, e.g., interactions between index and name substitution. However, our results are useful for those who wish to work in a system without a nominal package (e.g., Coq) for whatever reason (e.q., necessity of dependent types).

The Lambda Tamer package for Coq [4] also provides substantial support for defining datatypes with binding and their associated infrastructure. It assumes a pure de Bruijn encoding of object languages and appears to be designed for use with denotational semantics, not the structural operational semantics we considered in our System $F_{<:}$ type-safety proofs.

Gordon and Melham used an axiomatized presentation of named syntax to encode object languages [6]. This is completely analogous to our use of our lambda calculus libraries (c.f. Section 2.3) to encode object languages. The main difference between their work and ours is whether encodings of object languages look named (Gordon and Melham) or locally nameless (ours).

Stump [15] and Ridge [13] have both proposed libraries for two-level representation based on raw, named representations of syntax. Stump's library also generically proves a number of lemmas about terms with respect to an environment of free variables, in addition to providing basic support for syntax. Ridge's work is similar, though he chooses to represent treat variables bound by an environment similarly to variables bound by a term. Both libraries provide more functionality than our current lambda calculus libraries.

The Hybrid tool of Momigliano et al. [9] also takes a similar approach to provide an implementation of HOAS. Similar to our work, they build off of a locally nameless implementation of the lambda calculus. They provide a view of this datatype that uses the function space of the proof assistant (either Isabelle/HOL or Coq). Considerable effort is required to restrict the proof assistant's function space so that adequate encodings are possible. We avoid that effort by viewing the locally nameless datatypes in our lambda calculus libraries as they are. Moreover, because we do not attempt to make use of Coq's function space to implement HOAS, we obviate the need for two-level reasoning [5] that is needed when working with Hybrid. Rather than encode our judgements in a specification logic, which itself is encoded in Coq, we can write our definitions as standard Coq inductive datatypes and directly use Coq's mechanisms for manipulating such datatypes.

As mentioned in Section 2.3, our lambda calculus libraries are inspired by HOAS, LF and Twelf. Our lambda calculus libraries can be viewed as a simple logical framework, capable of only representing syntax. Canonical forms in LF are essential for adequate encodings. Our lambda calculus libraries do not define reduction on lambda terms. Instead, well-formedness (defined at the level of the object language) is used to ensure that one reasons only about terms in canonical form.

5 Conclusions and future work

Stepping back, our comparisons show that our collapsed and library-based approaches are successful at reducing binding infrastructure in a development, though the library-based representations are slightly less transparent than the others. Given the current state of our libraries, we prefer the collapsed and tagged representations, since they are very close to a locally nameless representation but with significantly less infrastructure.

There are several promising directions for future work. First, we can modify our typed lambda calculus library such that only well-sorted terms are representable.[7] This would allow both sort and arity errors when writing syntax to be caught at definition time (c.f. Section 3.4). Second, we conjecture that with a more sophisticated typed lambda calculus library, we could automatically generate induction principles and proofs of lemmas about well-formed terms. In essence, we would be implementing in Coq some of the features provided by Twelf's metalogic. We could alternatively use a tool, perhaps implemented as an extension of Ott, to automatically generate the infrastructure for any of the approaches discussed above.

Acknowledgments Many thanks to the members of the Penn PLClub, Randy Pollack, and anonymous reviewers for extensive feedback and discussion. This work was supported by NSF grant 545886, *CRI: Machine Assistance for Programming Language Research.*

References

[1] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *POPL '08: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 3–15. ACM, 2008.

[2] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The PoplMark challenge. In Joe Hurd and Tom Melham, editors, *Theorem Proving in Higher Order Logics: 18th International Conference, TPHOLs 2005*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005.

[3] Henk Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume II*, pages 117–309. Oxford University Press, 1992.

[4] Adam Chlipala. Generic programming and proving for programming language metatheory. Technical Report UCB/EECS-2007-147, University of California, Berkeley, 2007.

[5] Amy P. Felty. Two-level meta-reasoning in Coq. In Victor A. Carreño, Cézar A. Muñoz, and Sophiène Tahar, editors, *Theorem Proving in Higher Order Logics: 15th International Conference, TPHOLs 2002*, volume 2410 of *Lecture Notes in Computer Science*, pages 198–213. Springer, 2002.

[6] Andrew D. Gordon and Tom Melham. Five axioms of alpha-conversion. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs '96*, volume 1125 of *Lecture Notes in Computer Science*, pages 173–190. Springer, 1996.

[7] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[8] James McKinna and Robert Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3–4):373–409, 1999.

[9] Alberto Momigliano, Alan J. Martin, and Amy P. Felty. Two-level Hybrid: A system for reasoning using higher-order abstract syntax. In Brigitte Pientka and Carsten Schürmann, editors, *Proceedings of the Second International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2007)*, volume 196 of *Electronic Notes in Theoretical Computer Science*, pages 85–93. Elsevier, 2008.

[7] Dan Licata made this suggestion to us (personal communication).

[10] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, 1988.

[11] Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In Harald Ganzinger, editor, *Automated Deduction, CADE 16: 16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 202–206. Springer, 1999.

[12] Robert Pollack. Closure under alpha-conversion. In H. Barendregt and T. Nipkow, editors, *TYPES'93: Workshop on Types for Proofs and Programs, Nijmegen, May 1993, Selected Papers*, volume 806 of *Lecture Notes in Computer Science*, pages 313–332. Springer, 1994.

[13] Tom Ridge. To arrive where we started: experience of mechanizing binding. In *2nd Informal ACM SIGPLAN Workshop on Mechanizing Metatheory (WMM '07)*, 2007.

[14] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. In *ICFP '07: Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming*, pages 1–12. ACM, 2007.

[15] Aaron Stump. Towards a Coq library for programming languages metatheory with concrete names. In *1st Informal ACM SIGPLAN Workshop on Mechanizing Metatheory (WMM '06)*, 2006.

[16] Christian Urban, Julien Narboux, and Stefan Berghofer. Nominal datatype package for Isabelle/HOL. Available from `http://isabelle.in.tum.de/nominal/`, 2008.