J Autom Reasoning manuscript No.
(will be inserted by the editor)

Strongly Typed Term Representations in Coq

Nick Benton · Chung-Kil Hur ·
Andrew Kennedy · Conor McBride

Received: date / Accepted: date

Abstract There are two approaches to formalizing the syntax of typed object languages in a proof assistant or programming language. The *extrinsic* approach is to first define a type that encodes untyped object expressions and then make a separate definition of typing judgements over the untyped terms. The *intrinsic* approach is to make a single definition that captures well-typed object expressions, so ill-typed expressions cannot even be expressed. Intrinsic encodings are attractive and naturally enforce the requirement that metalanguage operations on object expressions, such as substitution, respect object types. The price is that the metalanguage types of intrinsic encodings and operations involve non-trivial dependency, adding significant complexity.

This paper describes intrinsic-style formalizations of both simply-typed and polymorphic languages, and basic syntactic operations thereon, in the Coq proof assistant. The Coq types encoding object-level variables (de Bruijn indices) and terms are indexed by both type and typing environment. One key construction is the boot-strapping of definitions and lemmas about the action of substitutions in terms of similar ones for a simpler notion of renamings. In the simply-typed case, this yields definitions that are free of any use of type equality coercions. In the polymorphic case, some substitution operations do still require type coercions, which we at least partially tame by uniform use of heterogeneous equality.

Keywords The Coq proof assistant · de Bruijn indices · Typed object languages

Hur was supported by the Digiteo/Ile-de-France project COLLODI (2009-28HD)

N. Benton (✉) · A. J. Kennedy
Microsoft Research, 7 J J Thomson Avenue, Cambridge, CB3 0FB, United Kingdom
E-mail: nick@microsoft.com

A. J. Kennedy
E-mail: akenn@microsoft.com

C. McBride
Department of Computer and Information Sciences, University of Strathclyde, 26 Richmond Street, Glasgow, G1 1XH, Scotland
E-mail: conor@strictlypositive.org

C.-K. Hur
Max Planck Institute for Software Systems, Campus E1 4, 66123 Saarbrücken, Germany
E-mail: gil@mpi-sws.org

1 Introduction

When encoding a typed object language in a proof assistant or programming language it is common to first define a datatype representing the abstract syntax of object-level expressions, and then make a separate inductive definition of typing judgements, relating expressions to types and type environments. This approach is sometimes referred to as *extrinsic*, and informally equated with Curry's view of types as *a posteriori* specifications concerning the form or behaviour of the raw untyped expressions. If, as is often the case, one is really only interested in statements about well-typed terms, an attractive alternative is the more Church-style[1] *intrinsic* approach, which builds the type rules into the definition of the abstract syntax right from the start, so all terms are well-typed 'by construction'. Intrinsic encodings reflect object-level types in metalanguage types, and can naturally and compactly enforce the requirement that operations over object language terms, such as substitution, should respect object-level types. In the extrinsic approach, definitions and lemmas become hedged with extra preconditions that not only add clutter but, when one works with them, have to be repeatedly and explicitly fulfilled – work that can largely be done by the metalanguage type system in an intrinsic encoding.

The price for working with intrinsic encodings is that they require more sophisticated metalanguage types to express the stronger invariants that one is now enforcing on object language expressions. In the functional programming community, 'strongly typed' encodings are a popular motivating example for generalized algebraic datatypes (GADTs) [22, 24, 10], and similar techniques can also be applied in modern object oriented languages [18]. In the type theory and automated reasoning community, such 'internal' representations have been described by a number of authors [2, 1, 13, 12, 21, 16, 8]. So the idea is well-known, and it is intuitively clear that dependently-typed calculi such as CiC have the power to express intrinsic encodings of, for example, simply typed languages. But in practice, however, the pain of actually working with 'very' dependent types in systems like Coq seems to have led most programming language researchers to use extrinsic encodings in their mechanized formalizations. A note by Sozeau[2] presents an intrinsic treatment of simple types and substitutions, but requires many awkward equality coercions.

This paper is a tutorial account of one way of working with intrinsic encodings in Coq, intended to show programming language metatheorists that this really is a viable option, rather than at telling hard-core type-theorists something new. We make no great claims of originality: most of the basic ideas are drawn from papers by Goguen and McKinna [15], Altenkirch and Reus [2] and Adams [1], and a note on representing simply typed terms and substitutions in Epigram by the fourth author.[3]

However, we give what we believe to be the first intrinsic treatment of simply-typed terms and substitutions in Coq that is entirely free of coercions. We show how some features added in Coq 8.2 by Sozeau, notably the `dependendent destruction` and `Program` tactics, help when working with non-trivial dependencies. We show how the technique extends to richer simply-typed constructs, such as pattern matching, and then apply it to give an intrinsic encoding of a second-order polymorphic language.

[1] The identification of the two encoding styles with the Church/Curry positions is rather imprecise. In particular, it is common to work with extrinsic contextual typing judgements over a Church-style datatype for syntax, in which binders are annotated with types.

[2] Sozeau, M: A dependently-typed formalization of simply-typed lambda-calculus: substitution, denotation, normalization, manuscript (2007).

[3] McBride, C: Type preserving renaming and substitution, manuscript (2005).

The encodings presented here have been successfully used in Coq formalizations of non-trivial results about programming languages. We have formalized the domain-theoretic semantics of the simply-typed language and proved its soundness and adequacy [5], and subsequently used that semantics to formulate and prove results on compiler correctness [3]. The encoding of the polymorphic language has also been used as the basis of a formalization of compiler correctness [4].

The Coq code described here is available from the authors' web pages.

2 A simply-typed language

We start with a small simply-typed language with a base type of natural numbers. For presentational purposes, we omit general recursion as this will let us present a simple set-theoretic denotational semantics later on.

In the first part of the paper, we will present the *complete* Coq code for the simply-typed language, albeit not in a strict lexically-scoped order. The following prologue sets up the options and imports from the Coq library that we will use:

```
Require Import List.
Require Import Program.
Require Import FunctionalExtensionality.
Require Import EqNat.
Set Implicit Arguments.
```

The Coq type `Ty` defines the types of the object language, with base type `NAT` and arrow type constructor `ARR`. Since we are using de Bruijn indices, an object level type environment is encoded as a list of types:

```
Inductive Ty := NAT | ARR (ty1 ty2 : Ty).
Definition Env := list Ty.
```

Figure 1 presents typing rules for our (entirely standard) object language in the conventional way, using named binders.

$$\text{VAR}\ \frac{x : t \in E}{E \vdash x : t} \qquad \text{CONST}\ \frac{}{E \vdash n : \mathsf{nat}} \qquad \text{SUCC}\ \frac{E \vdash e : \mathsf{nat}}{E \vdash \mathsf{succ}\ e : \mathsf{nat}} \qquad \text{PRED}\ \frac{E \vdash e : \mathsf{nat}}{E \vdash \mathsf{pred}\ e : \mathsf{nat}}$$

$$\text{IFZ}\ \frac{E \vdash e : \mathsf{nat} \qquad E \vdash e_1 : t \qquad E \vdash e_2 : t}{E \vdash \mathsf{ifz}\ e\ e_1\ e_2 : t} \qquad \text{LAM}\ \frac{E, x : t_1 \vdash e : t_2}{E \vdash \lambda x.e : t_1 \to t_2}$$

$$\text{APP}\ \frac{E \vdash e_1 : t_1 \to t_2 \qquad E \vdash e_2 : t_1}{E \vdash e_1\ e_2 : t_2}$$

Fig. 1 The simply-typed lambda calculus with naturals

An *extrinsic* approach to encoding this language would be to define an abstract syntax for expressions, using one's favourite method for representing binding, and then separately to give a typing relation over that syntax. Instead, we take an *intrinsic* approach, combining syntax and typing, by *indexing* the types of variables `Var` and expressions `Exp` by the environment `E` and type `t` for which they are well-typed:

```
Inductive Var : Env → Ty → Type :=
| ZVAR : ∀ E t, Var (t::E) t
| SVAR : ∀ E t t', Var E t → Var (t'::E) t.

Inductive Exp E : Ty → Type :=
| VAR  : ∀ t, Var E t → Exp E t
| CONST: nat → Exp E NAT
| SUCC : Exp E NAT → Exp E NAT
| PRED : Exp E NAT → Exp E NAT
| IFZ  : ∀ t, Exp E NAT → Exp E t → Exp E t → Exp E t
| APP  : ∀ t1 t2, Exp E (ARR t1 t2) → Exp E t1 → Exp E t2
| LAM  : ∀ t1 t2, Exp (t1 :: E) t2 → Exp E (ARR t1 t2).
```

An element of `Var E t` is essentially a derivation establishing that the type `t` is at some position in the list `E`. We count from the left, so the constructor `ZVAR` witnesses that `t` is at the zeroth position in an environment of the form `t::E`, whilst `SVAR` takes a proof that `t` is at some position n in the list `E` and produces a proof that `t` is at position $n+1$ in `t'::E`. (Note that the type of `ZVAR` builds in weakening.)

Now the typing rules for expressions are directly encoded in the types of the corresponding constructors of `Exp`. For example, the application constructor, `APP`, takes two expressions as arguments, one of arrow type `ARR t1 t2` and the other of type `t1`, yielding an expression of type `t2`. The function constructor `LAM` takes an expression typed as `t2` under an environment extended with the argument of type `t1` and produces an expression of function type `ARR t1 t2`.

The inductive types `Var` and `Exp` are *indexed* by a (value) `t:Ty` that is allowed to vary in the result types of the different constructors. In the functional programming community, datatypes whose (type) indices vary in this way are known as Generalized Algebraic Datatypes (*GADT*s). Indeed, a typed term representation similar to the above can be found in the standard test suite for the `ghc` compiler. For the `Exp` type, we have made the environment `E` into a *parameter*; note that this is scoped over all the constructors and remains the same in the return type of every constructor, though it does vary in the arguments to constructors, in a way that a functional programmer would call *non-regular* or *nested* [6].

Definitions and statements involving strongly-typed terms are beautifully concise. For example, here is a complete definition of the "call-by-value" evaluation relation for closed expressions of type `t`:

```
Inductive Ev : ∀ t, Exp nil t → Exp nil t → Prop :=
| EvCONST : ∀ n, Ev (CONST _ n) (CONST _ n)
| EvLAM  : ∀ t1 t2 (e:Exp [t1] t2), Ev (LAM e) (LAM e)
| EvSUCC : ∀ e n, Ev e (CONST _ n) → Ev (SUCC e) (CONST _ (n+1))
| EvPRED : ∀ e n, Ev e (CONST _ n) → Ev (PRED e) (CONST _ (n-1))
| EvIFZTHEN : ∀ t1 e (e1 e2:Exp nil t1) v,
    Ev e (CONST _ 0) → Ev e1 v → Ev (IFZ e e1 e2) v
| EvIFZELSE : ∀ t1 e (e1 e2:Exp nil t1) v n,
    Ev e (CONST _ (S n)) → Ev e2 v → Ev (IFZ e e1 e2) v
| EvAPP : ∀ t1 t2 e v w (e1 : Exp nil (ARR t1 t2)) e2,
    Ev e1 (LAM e) → Ev e2 w → Ev (STmExp {| w |} e) v →
    Ev (APP e1 e2) v.
```

Figure 2 presents essentially the same relation in more conventional form using named binders and type-free syntax. Again, the Coq encoding is a fairly direct translation of the

$$\textsc{EvConst}\ \frac{}{n \Downarrow n} \qquad \textsc{EvLam}\ \frac{}{\lambda x.e \Downarrow \lambda x.e} \qquad \textsc{EvSucc}\ \frac{e \Downarrow n}{\mathsf{succ}\ e \Downarrow n+1}$$

$$\textsc{EvPred}\ \frac{e \Downarrow n}{\mathsf{pred}\ e \Downarrow n-1} \qquad \textsc{EvIfzThen}\ \frac{e \Downarrow 0 \qquad e_1 \Downarrow v}{\mathsf{ifz}\ e\ e_1\ e_2 \Downarrow v} \qquad \textsc{EvIfzElse}\ \frac{e \Downarrow n+1 \qquad e_2 \Downarrow v}{\mathsf{ifz}\ e\ e_1\ e_2 \Downarrow v}$$

$$\textsc{EvApp}\ \frac{e_1 \Downarrow \lambda x.e \qquad e_2 \Downarrow v_2 \qquad [x := v_2]e \Downarrow v}{e_1\ e_2 \Downarrow v}$$

Fig. 2 Evaluation relation

conventional rules, but because we work with a strongly-typed representation, the fact that the definition of `Ev` typechecks in Coq gives us a proof that our evaluation relation preserves types almost for free, alongside its definition.[4]

Consider the most complex constructor `EvAPP`: it states that if an expression `e1` evaluates to a lambda expression `LAM e`, and `e2` evaluates to `w`, and `e` with its only free variable replaced by `w` evaluates to `v`, then `APP e1 e2` evalutes to `v`. However, we have presented our definitions out of order: neither the notation `{| w |}`, which is intended to denote a substitution mapping the zero'th variable in the environment to `w`, nor the function `STmExp`, which applies a substitution to an expression, have yet been defined. How to define and work with strongly typed substitutions will be explained in the next couple of sections.

3 Substitutions

We represent *typed substitutions* by functions that map variables typed in an environment `E` to expressions typed in an environment `E'`, written `Sub E E'`:

```
Definition Sub E E' := ∀ t, Var E t → Exp E' t.
```

The list-style notation `{| e0 ,..., en-1 |}` represents the substitution which takes variables numbered 0 to $n-1$ to expressions $\mathtt{e}_0$ to $\mathtt{e}_{n-1}$. The substitution is built up using the identity substitution `idSub` and an operator `consSub` which extends a substitution on the zero'th variable. Operations `hdSub` and `tlSub` can be used to decompose a substitution into the image of the zero'th variable and the remainder of the substitution.

```
Definition idSub {E} : Sub E E := @VAR E.

Program Definition consSub {E E' t} (e:Exp E' t) (s:Sub E E')
  : Sub (t::E) E' :=
  fun t' (v:Var (t::E) t') ⇒
    match v with
    | ZVAR _ _ ⇒ e
    | SVAR _ _ _ v' ⇒ s _ v'
    end.

Notation "{| e ; .. ; f |}" := (consSub e .. (consSub f idSub) ..).
```

[4] Though one might reasonably object that the strongly-typed evaluation relation obscures the fact that evaluation does not depend on typing, of course.

```
Definition tlSub {E E' t} (s:Sub (t::E) E') : Sub E E' :=
  fun t' v ⇒ s t' (SVAR t v).
Definition hdSub {E E' t} (s:Sub (t::E) E') : Exp E' t :=
  s t (ZVAR _ _).
```

Notice here the use of Sozeau's `Program` tactic [23], supporting GADT-style pattern matching in Coq. Looking at the first branch of the `match` in the definition of `consSub`, we see that the declared type of `e` is `Exp E' t` whereas the type of the whole match expression is supposed to be `Exp E' t'`. But if `v : Var (t::E) t'` matches `ZVAR _ _` we must have `t=t'`. The `Program` tactic generates and exploits this equation, producing a slightly more complex CiC term 'behind the scenes'. (McBride [19] explains this transformation in more detail.)

Now let us write the function that applies a substitution to an expression:

```
Fixpoint STmExp E E' t (s:Sub E E') (e:Exp E t) :=
  match e with
  | VAR _ v      ⇒ s _ v
  | CONST n      ⇒ CONST _ n
  | SUCC e       ⇒ SUCC (STmExp s e)
  | PRED e       ⇒ PRED (STmExp s e)
  | IFZ _ e e1 e2 ⇒ IFZ (STmExp s e) (STmExp s e1) (STmExp s e2)
  | APP _ _ e1 e2 ⇒ APP (STmExp s e1) (STmExp s e2)
  | LAM _ _ e    ⇒ LAM (STmExp (STmL s) e)
  end.
```

In the variable case we apply the substitution, and in most of the other cases we just do the obvious homomorphic thing. The interesting case is that for the `LAM` constructor, in which we have to apply the substitution under a binder. We want (indeed, the type system tells us we need) a function `STmL` that will *lift* the substitution to work over expressions in an extended environment. So let us define `STmL`:

```
Program Definition
  STmL {E E'} t (s:Sub E E') : Sub (t::E) (t::E') :=
  fun t' v ⇒
  match v with
  | ZVAR _ _ ⇒ VAR (ZVAR _ _)
  | SVAR _ _ _ v' ⇒ ShTmExp t (s _ v')
  end.
```

So far, so good, but we have now discovered that we have to be able to reinterpret (or 'transport') the expressions returned by our original substitution to work in the larger environment; hence we have postulated a *shift* operation `ShTmExp` of type `Exp E t' → Exp (t::E) t'`. Intuitively, the shift just increments all the (term) variables in the expression. In Haskell with GADTs, we could implement this operation simply by applying a trivial substitution to the expression, using `STmExp (fun t v ⇒ SVAR t' v)`. But this mutual recursion between substitution and shifting is not structurally recursive, and is therefore unacceptable in Coq.

One might now define `ShTmExp` directly, rather than in terms of substitutions. Because shifting itself has to be able to go under binders, one soon realizes that the type has to be generalized, but one can then easily define a shift function with type

```
∀ E E' t' t, Exp (E++E') t → Exp (E++[t']++E') t
```

Unfortunately, working with that definition in Coq quickly becomes difficult. The problem is then properties of shifting have to be proved by induction over an argument expression of *arbitrary* type `Exp E t`. This involves recasting statements of the form

```
∀ E E' t (e:Exp (E++E') t), ...
```

into the form

```
∀ E0 t (e:Exp E0 t) E E', E0=E++E' → ...
```

which requires passing in a proof of the equality. Sozeau's Coq formalization of the simply-typed lambda calculus uses this technique, but then requires the use of `eq_rect` cast operations, and many lemmas that simply push the coercions around in a manner that quickly becomes no fun at all.

4 Renamings

A better way of defining shifting is to observe [1] that a shift is an instance of a special, restricted kind of substitution: a *renaming*; that is, a map from variables to variables:

```
Definition Ren E E' := ∀t, Var E t →Var E' t.
```

It is easy to define lifting for renamings, without running into issues with recursion:

```
Program Definition RTmL {E E' t}
  (r : Ren E E') : Ren (t::E) (t::E') := fun t' v ⇒
  match v with
  | ZVAR _ _ ⇒ZVAR _ _
  | SVAR _ _ _ v' ⇒SVAR _ (r _ v')
  end.
```

Applying a renaming to an expression is straightforward:

```
Fixpoint RTmExp E E' t (r:Ren E E') (e:Exp E t) :=
  match e with
  | VAR _ v      ⇒VAR (r _ v)
  | CONST n      ⇒CONST _ n
  | SUCC e       ⇒SUCC (RTmExp r e)
  | PRED e       ⇒PRED (RTmExp r e)
  | IFZ _ e e1 e2 ⇒IFZ (RTmExp r e) (RTmExp r e1) (RTmExp r e2)
  | APP _ _ e1 e2 ⇒APP (RTmExp r e1) (RTmExp r e2)
  | LAM _ _ e    ⇒LAM (RTmExp (RTmL r) e)
  end.
```

And we can now define our shifting operation `ShTmExp` by applying a trivial renaming:

```
Definition ShTmExp E t t' : Exp E t →Exp (t'::E) t
  := RTmExp (fun _ v ⇒SVAR _ v).
```

In order to make any use of these definitions, we have to prove a standard collection of lemmas about composing substitutions and so on. So as to fit the complete development of the theory of our simple language in the paper, and because it is instructive in itself, we first define a couple of custom tactics for rewriting:

```
Ltac Rewrites E :=
  (intros; simpl; try rewrite E;
   repeat (match goal with | [H:context[_=_] |- _] ⇒ rewrite H end);
   auto).

Ltac ExtVar :=
 match goal with
    [ |- ?X = ?Y ] ⇒
    (apply (@functional_extensionality_dep _ _ X Y) ;
     let t := fresh "t" in intro t;
     apply functional_extensionality;
     let v := fresh "v" in intro v;
     dependent destruction v; auto)
  end.
```

The `Rewrites` tactical applies the rewrite rule passed as argument, and then applies any equations that have been introduced as hypotheses, typically through induction. The `ExtVar` tactic applies extensionality for renamings or substitutions, introducing a type `t` and variable `v : Var _ t` into the context and then doing inversion on `v` by `dependent destruction`, which (as with `Program`) takes care of generalizing the goal by the equalities generated by matching on `v`.

Using these tactics, we prove that lifting the identity gives the identity on the extended context, and hence that the action of the identity is the identity on terms:

```
Lemma LiftIdSub : ∀E t, STmL (@idSub E) = @idSub (t::E).
Proof. intros. ExtVar. Qed.

Lemma ActIdSub : ∀E t (e : Exp E t), STmExp idSub e = e.
Proof. induction e; Rewrites LiftIdSub. Qed.
```

The main downside of defining shift in terms of renaming is that we have defined everything twice: once for renaming, and once for substitution. And we now have *four* notions of composition:

```
Definition RcR {E E' E''} (r : Ren E' E'') (r' : Ren E E') :=
  (fun t v ⇒ r t (r' t v)) : Ren E E''.
Definition ScR {E E' E''} (s : Sub E' E'') (r : Ren E E') :=
  (fun t v ⇒ s t (r t v)) : Sub E E''.
Definition RcS {E E' E''} (r : Ren E' E'') (s : Sub E E') :=
  (fun t v ⇒ RTmExp r (s t v)) : Sub E E''.
Definition ScS {E E' E''} (s : Sub E' E'') (s' : Sub E E') :=
  (fun t v ⇒ STmExp s (s' t v)) : Sub E E''.
```

For each notion of composition we prove that lifting is preserved and that the action of a composition is a composition of actions. These lemmas must be proved *in order*, with each building on the previous:

```
Lemma LiftRcR : ∀E E' E'' t (r:Ren E' E'') (r':Ren E E'),
  RTmL (t:=t) (RcR r r') = RcR (RTmL r) (RTmL r').
Proof. intros. ExtVar. Qed.

Lemma ActRcR : ∀E t (e:Exp E t) E' E'' (r:Ren E' E'') (r':Ren E E'),
```

```
  RTmExp (RcR r r') e = RTmExp r (RTmExp r' e).
Proof. induction e; Rewrites LiftRcR. Qed.

Lemma LiftScR : ∀E E' E'' t (s:Sub E' E'') (r:Ren E E'),
  STmL (t:=t) (ScR s r) = ScR (STmL s) (RTmL r).
Proof. intros. ExtVar. Qed.

Lemma ActScR : ∀E t (e:Exp E t) E' E'' (s:Sub E' E'') (r:Ren E E'),
  STmExp (ScR s r) e = STmExp s (RTmExp r e).
Proof. induction e; Rewrites LiftScR. Qed.

Lemma LiftRcS : ∀E E' E'' t (r:Ren E' E'') (s:Sub E E'),
  STmL (t:=t) (RcS r s) = RcS (RTmL r) (STmL s).
Proof. intros. ExtVar. unfold RcS. simpl.
unfold ShTmExp. rewrite <- 2 ActRcR. auto. Qed.

Lemma ActRcS : ∀E t (e:Exp E t) E' E'' (r:Ren E' E'') (s:Sub E E'),
  STmExp (RcS r s) e = RTmExp r (STmExp s e).
Proof. induction e; Rewrites LiftRcS. Qed.

Lemma LiftScS : ∀E E' E'' t (s:Sub E' E'') (s':Sub E E'),
  STmL (t:=t) (ScS s s') = ScS (STmL s) (STmL s').
Proof. intros. ExtVar. simpl. unfold ScS. simpl.
unfold ShTmExp. rewrite <- ActRcS. rewrite <- ActScR. auto. Qed.

Lemma ActScS : ∀E t (e:Exp E t) E' E'' (s:Sub E' E'') (s':Sub E E'),
  STmExp (ScS s s') e = STmExp s (STmExp s' e).
Proof. induction e; Rewrites LiftScS. Qed.
```

5 Example

Our experience with strong typing of terms is that the pain is worth it. Just as with typeful *programming*, typeful *proving* provides a framework for getting the definitions right and, generally speaking, the proofs then follow smoothly.

Type-indexed terms fit very well with typed-indexed semantics. To illustrate, we now describe how to give a set-theoretic denotational semantics to our simple language. The formalization of a slightly more sophisticated domain-theoretic semantics for a language with recursion [5] follows just the same pattern, but working with a total language allows us here to focus on the type structure, without dragging in extraneous definitions concerning cpos and continuous functions.

We interpret `NAT` as Coq's `nat` type, `ARR` as Coq's $\rightarrow$, and environments as iterated product:

```
Fixpoint SemTy t :=
  match t with
  | NAT ⇒ nat
  | ARR t1 t2 ⇒ SemTy t1 → SemTy t2
  end.
```

```
Fixpoint SemEnv E :=
  match E with
  | nil ⇒ unit
  | t :: E ⇒ prodT (SemTy t) (SemEnv E)
  end.
```

It is then straightforward to give a meaning to variables of type `Var E t` and (open) expressions of type `Exp E t` as functions of type `SemEnv E → SemTy t`:

```
Fixpoint SemVar E t (v:Var E t) : SemEnv E → SemTy t :=
  match v with
  | ZVAR _ _    ⇒ fun se ⇒ fst se
  | SVAR _ _ _ v ⇒ fun se ⇒ SemVar v (snd se)
  end.

Fixpoint SemExp E t (e:Exp E t) : SemEnv E → SemTy t :=
  match e with
  | VAR _ v      ⇒ SemVar v
  | CONST n      ⇒ fun se ⇒ n
  | SUCC e       ⇒ fun se ⇒ SemExp e se + 1
  | PRED e       ⇒ fun se ⇒ SemExp e se - 1
  | IFZ _ e e1 e2 ⇒ fun se ⇒ if beq_nat (SemExp e se) 0
                             then SemExp e1 se else SemExp e2 se
  | APP _ _ e1 e2 ⇒ fun se ⇒ SemExp e1 se (SemExp e2 se)
  | LAM _ _ e    ⇒ fun se ⇒ fun x ⇒ SemExp e (x,se)
  end.
```

Notice how natural these definitions are: the typed de Bruijn representation for variables fits perfectly with the use of pairing to extend environments.

In order to prove that the semantics is sound, we first need to prove a lemma showing that the semantics commutes with substitution. As with the syntactic proofs concerning composition, this lemma must be boot-strapped from an analogous lemma concerning renaming, as follows:

```
Fixpoint SemSub E E' : Sub E' E → SemEnv E → SemEnv E' :=
  match E' with
  | nil   ⇒ fun s se ⇒ tt
  | _ :: _ ⇒ fun s se ⇒ (SemExp (hdSub s) se, SemSub (tlSub s) se)
  end.

Fixpoint SemRen E E' : Ren E' E → SemEnv E → SemEnv E' :=
  match E' with
  | nil   ⇒ fun r se ⇒ tt
  | _ :: _ ⇒ fun r se ⇒ (SemVar (hdRen r) se, SemRen (tlRen r) se)
  end.

Lemma SemRenComm :
   ∀ E t (e : Exp E t) E' (r : Ren E E'),
   ∀ se, SemExp e (SemRen r se) = SemExp (RTmExp r e) se.
```

```
Lemma SemSubComm :
   ∀ E t (e : Exp E t) E' (s : Sub E E'),
   ∀ se, SemExp e (SemSub s se) = SemExp (STmExp s e) se.
```

We can now prove *soundness*: if an expression `e` evaluates to a value `v` then the denotation of `e` is the denotation of `v`. The above lemma is used in the crucial `EvAPP` case in the proof.

```
Theorem Soundness :
  ∀ t (e : Exp nil t) v, Ev e v → SemExp e = SemExp v.
```

The semantics is also *adequate*: if the denotation of a closed expression `e` of base type is `n`, then `e` evaluates to `CONST _ n`. To prove adequacy, we use the standard method of defining a logical relation between syntax and semantics.

```
Definition evAndRel R t (e:Exp nil t) (d:SemTy t) :=
  ∃ v, Ev e v ∧ R t v d.

Fixpoint rel t : Exp nil t → SemTy t → Prop :=
match t with
| NAT          ⇒ fun v n ⇒ v = CONST _ n
| ARR t1 t2    ⇒ fun v f ⇒ ∃ e, v = LAM e
                  ∧ ∀ x w, @rel t1 w x →
                     @evAndRel rel t2 (STmExp {|w|} e) (f x)
end.
```

This is extended to environments:

```
Fixpoint relEnv E : Sub E nil → SemEnv E → Prop :=
  match E with
  | nil    ⇒ fun s se ⇒ True
  | t :: E ⇒ fun s se ⇒
    @rel t (hdSub s) (fst se) ∧ @relEnv E (tlSub s) (snd se)
  end.
```

Then we can prove a fundamental theorem and adequacy is a corollary:

```
Theorem FundamentalTheorem :
  ∀ E t e se s, @relEnv E s se →
                   @evAndRel rel t (STmExp s e) (SemExp e se).

Corollary Adequacy :
  ∀ (e : Exp nil NAT) n, SemExp e tt = n → Ev e (CONST _ n).
```

The whole development, including syntax and semantics, is roughly 200 lines of definition and 120 lines of proof.

6 Extensions

The approach extends easily to more complex uses of variable binding. For example, here is a constructor for recursive functions, which might be written rec $f(x:t_1):t_2 = e$ in more conventional notation.

```
Inductive Exp E : Ty → Type :=
  ...
| REC : ∀ t1 t2, Exp (t1::ARR t1 t2::E) t2 → Exp E (ARR t1 t2).
```

It is also straightforward to model SML-style pattern matching. Assuming type constructors PROD and SUM for products and sums, we can define SML-style pattern expressions of type `Pat E t` where `t` is the type of the whole pattern and `E` lists the types of variables mentioned in the pattern, reading from left to right.

```
Inductive Pat : Env → Ty → Type :=
| PPAIR : ∀ E1 E2 t1 t2, Pat E1 t1 → Pat E2 t2 →
           Pat (E1++E2) (PROD t1 t2)
| PVAR : ∀ t, Pat [t] t
| PWILD : ∀ t, Pat [] t
| PFAIL : ∀ t, Pat [] t
| PAS  : ∀ E t, Pat E t → Pat (t::E) t
| PINL : ∀ E t1 t2, Pat E t1 → Pat E (SUM t1 t2)
| PINR : ∀ E t1 t2, Pat E t2 → Pat E (SUM t1 t2).
```

We can then use pattern expressions in a ‘let’ construct, as follows:

```
Inductive Exp E : Ty → Type :=
  ...
| LETPAT : ∀ E' t1 t2, Exp E t1 → Pat E' t1 → Exp (E'++E) t2 →
            Exp E t2.
```

7 Abstracting maps

For languages larger than Exp, the necessity to do so many things twice, once for renamings, and once for substitutions, becomes somewhat painful. At least some cutting-and-pasting of definitions and proofs can be avoided by observing the commonality between renaming and substitution, abstracting both notions into a single Map type that is parameterized on the type constructor used to construct its target, namely Var (for renamings) or Exp (for substitutions).

```
Section MAPS.
  Variable P : Env → Ty → Type.
  Definition Map E E' := ∀ t, Var E t → P E' t.
  Definition tlMap {E E' t} (m:Map (t::E) E') : Map E E' :=
    fun t' v ⇒ m t' (SVAR t v).
  Definition hdMap {E E' t} (m:Map (t::E) E') : P E' t :=
    m t (ZVAR _ _).

  Program Definition consMap {E E' t} (p:P E' t) (m:Map E E')
  : Map (t::E) E' :=
  fun t' (var:Var (t::E) t') ⇒
    match var with
    | ZVAR _ _ ⇒ p
    | SVAR _ _ _ var' ⇒ m _ var'
    end.
```

We then package up the fundamental operations used in lifting and the action of a map on an expression:

```
Record MapOps := mkOps
{
  vr : ∀E t, Var E t →P E t;
  vl : ∀E t, P E t →Exp E t;
  wk : ∀E t t', P E t →P (t' :: E) t
}.
```

Here `vr` is the embedding of a variable into the target type, namely the identity for renamings, and the `Var` constructor for expressions. The `vl` is an embedding into `Exp`, namely the `Var` constructor for variables, and the identity for expressions. Finally `wk` is the operation that maps into a weaker context, namely `SVAR` for variables, and `ShTmExp` (shift) for expressions. We can then define 'generic' lifting and application functions, given a package `ops` of type `MapOps P`.

```
Variable ops : MapOps.
Definition shiftMap {E E'} t (m:Map E E') : Map E (t::E') :=
  fun vt v ⇒wk ops t (m vt v).
Definition MTmL E E' t (m:Map E E') : Map (t::E) (t::E') :=
  consMap (vr ops (ZVAR E' t)) (shiftMap t m).

Fixpoint MTmExp E E' t (m:Map E E') (e:Exp E t) :=
match e with
| VAR _ v ⇒vl ops (m _ v)
| CONST n ⇒CONST _ n
| SUCC e ⇒SUCC (MTmExp m e)
| PRED e ⇒PRED (MTmExp m e)
| IFZ _ e e1 e2 ⇒IFZ (MTmExp m e) (MTmExp m e1) (MTmExp m e2)
| APP _ _ e1 e2 ⇒APP (MTmExp m e1) (MTmExp m e2)
| LAM _ _ e ⇒LAM (MTmExp (MTmL m) e)
end.
```

Notice how this time we have defined the `MTmL` lifting operation in terms of `consMap`; we could have done something similar earlier for renamings and substitutions.

The instantiation to give operations on renamings and substitutions is straightforward:

```
Definition Ren := Map Var.
Definition RMapOps := mkOps Var (fun _ _ v ⇒v) VAR (@SVAR).
Definition RTmExp := MTmExp RMapOps.
Definition RTmL  := MTmL RMapOps.

Definition Sub := Map Exp.
Definition ShTmExp E t t' : Exp E t →Exp (t'::E) t
 := RTmExp (fun _ v ⇒SVAR _ v).
Definition SMapOps := mkOps Exp VAR (fun _ _ v ⇒v) ShTmExp.
Definition STmExp := MTmExp SMapOps.
Definition STmL  := MTmL SMapOps.
```

The composition lemmas one then proves are essentially the same as before, and these are easily dispatched tactically; the main advantage of the map abstraction here is in avoiding

repeated definitions. One might also usefully parameterize MTmExp by a monad, to support generic effectful traversals.

In applications such as our semantic soundness result, we have a degree of *proof* duplication too: we needed to prove both SemRenComm and SemSubComm. But here, too, it is possible to abstract out the commonality in a generic SemMapComm lemma:

```
Variable P : Env → Ty → Type.
Variable ops : MapOps P.
Variable Sem : ∀E t, P E t → SemEnv E → SemTy t.
Variable SemVl : ∀E t (v:P E t), Sem v = SemExp (vl ops _ _ v).
Variable SemVr : ∀E t se, Sem (vr ops (ZVAR E t)) se = fst se.
Variable SemWk : ∀E t (v:P E t) t' se,
  Sem (wk ops _ _ t' v) se = Sem v (snd se).

Fixpoint SemMap E E' : Map P E' E → SemEnv E → SemEnv E' :=
match E' with
| nil ⇒ fun m se ⇒ tt
| _   ⇒ fun m se ⇒ (Sem (hdMap m) se, SemMap (tlMap m) se)
end.

Lemma SemMapComm :
 ∀ E t (e : Exp E t) E' (m : Map P E E'),
 ∀ se, SemExp e (SemMap m se) = SemExp (MTmExp ops m e) se.
```

We've parameterized on P and ops, as with the definitions of syntax, but also on Sem, which we later instantiate to SemVar and SemExp, and on three simple properties of Sem that describe its interaction with the vl, vr and wk operations from ops. The proof of SemMapComm makes use of these properties.

The proof of SemRenComm is then just an easy special case of SemMapComm, the three proof obligations being discharged by auto. The proof of SemSubComm requires non-trivial reasoning only to discharge the SemWk property, and of course makes use of SemRenComm.

8 Polymorphism

We now turn to applying the same basic ideas to an intrinsic encoding of the second order polymorphic lambda calculus, System F. We will use the renamings and substitutions idea for both types and terms. There is a mild combinatorial explosion in the number of forms of application and composition (e.g. the action of a type substitution on a term renaming) but, fortunately, not *all* combinations show up in establishing the lemmas that clients need.

Types now contain type variables, represented again by de Bruijn indices. A type variable context is represented simply by its length, a natural number u saying how many type variables are available. We then define Coq types for well-formed type variables and types in context:

```
Inductive TyVar : nat → Type :=
| ZTYVAR : ∀u, TyVar (S u)
| STYVAR : ∀u, TyVar u → TyVar (S u).
```

```
Inductive Ty u : Type :=
| TYVAR : TyVar u → Ty u
| ARR : Ty u → Ty u → Ty u
| ALL : Ty (S u) → Ty u.
```

Type renamings and substitutions are defined as follows:

```
Definition RenT u w := TyVar u → TyVar w.
Definition SubT u w := TyVar u → Ty w.
```

Lifting of renamings and the action of a renaming on a type are given by

```
Program Definition RTyL u w (r:RenT u w) : RenT (S u) (S w) :=
fun var ⇒
  match var with
  | ZTYVAR _ ⇒ (ZTYVAR _)
  | STYVAR _ var' ⇒ STYVAR (r var')
  end.

Fixpoint RTyT u w (r:RenT u w) (t:Ty u) : Ty w :=
  match t with
  | TYVAR v ⇒ TYVAR (r v)
  | ARR t1 t2 ⇒ ARR (RTyT r t1) (RTyT r t2)
  | ALL t ⇒ ALL (RTyT (RTyL r) t)
  end.
```

and again, shifting is defined as a special renaming, which is in turn used to define the action of a substitution on a type:

```
Definition ShTyT u : Ty u → Ty (S u) := RTyT (@STYVAR _).

Program Definition STyL u w (s:SubT u w) : SubT (S u) (S w) :=
fun v ⇒
  match v with
  | ZTYVAR _ ⇒ TYVAR (ZTYVAR _)
  | STYVAR _ v' ⇒ ShTyT (s v')
  end.

Fixpoint STyT u w (s:SubT u w) (t:Ty u) : Ty w :=
  match t with
  | TYVAR v ⇒ s v
  | ARR t1 t2 ⇒ ARR (STyT s t1) (STyT s t2)
  | ALL t ⇒ ALL (STyT (STyL s) t)
  end.
```

Following the pattern we used earlier for simply typed terms, we now define notations for particular type substitutions (we use `[| t1,...,tn |]`), composition of type renamings and type substitutions, and prove appropriate lemmas.

We are now ready to introduce strongly polymorphically typed *terms*. A well-formed term variable environment in a type variable context with u free variables is represented, again using de Bruijn indices, as a list of u-types, and the action of type substitutions on term environments is just given by mapping:

```
Definition Env u := list (Ty u).

Fixpoint STyE u w (sub: SubT u w) (env: Env u) : Env w :=
  match env with
    | nil ⇒ nil
    | T::TS ⇒ STyT sub T :: STyE sub TS
  end.
```

Next we define a type for typed variables in a given term and type variable context:

```
Inductive Var u : Env u → Ty u → Type :=
| ZVAR : ∀ env ty, Var (ty :: env) ty
| SVAR : ∀ env ty' ty, Var env ty → Var (ty' :: env) ty.
```

For convenience, we re-express type shifting as a substitution:

```
Definition shSubT u : SubT u (S u) := fun v ⇒ TYVAR (STYVAR v).
Implicit Arguments shSubT [].
```

and the definition of terms is then a pleasingly direct translation of the 'normal' typing rules for the polymorphic lambda calculus:

```
Inductive Exp u (E:Env u) : Ty u → Type :=
| VAR : ∀ t, Var E t → Exp E t
| LAM : ∀ t1 t2, Exp (t1 :: E) t2 → Exp E (ARR t1 t2)
| APP : ∀ t1 t2, Exp E (ARR t1 t2) → Exp E t1 → Exp E t2
| TAPP : ∀ t, Exp E (ALL t) → ∀ t':Ty u, Exp E (STyT [| t' |] t)
| TABS : ∀ t, Exp (u:=S u) (STyE (shSubT _) E) t → Exp E (ALL t).
```

Again, no proofs of equalities are passed as arguments to any of the constructors; everything is built up by inductive definitions. Note how type substitution shows up in the type of the `TAPP` constructor, and how the environment is shifted in the type of the argument to `TABS`.

The broad pattern of the formalization of operations on the polymorphic language follows that of the simply-typed case. We have type renamings and type substitutions, and term renamings and term substitutions. We choose to abstract the two kinds of traversals over terms into a more general notion of mapping, along the lines described in Section 7, with an extra operation component to account for the action of type substitutions.

Working with the polymorphic encoding is qualitatively more tricky than was the case for the simply-typed language, however. Although the definitions of polymorphic types and terms are simple and elegant, we have this time *not* managed to avoid explicit uses of type equalities when we come to define functions working over that syntax. Here, for example, is the definition of the action of type substitutions on terms:

```
Fixpoint STyExp u w (s:SubT u w) (E:Env u) t (e:Exp E t)
 : Exp (STyE s E) (STyT s t) :=
  match e with
  | VAR _ v ⇒ VAR (STyVar s v)
  | APP _ _ e1 e2 ⇒ APP (STyExp s e1) (STyExp s e2)
  | LAM _ _ e ⇒ LAM (STyExp s e)
  | TAPP _ e t' ⇒ cast (STyExp_cast1 _ _ _ _)
                  (TAPP (STyExp s e) (STyT s t'))
  | TABS _ e ⇒ TABS (cast (STyExp_cast2 _ _ _)
                    (STyExp (STyL s) e))
  end.
```

Observe that we have been forced to make explicit applications of `cast`, the obvious operation of type `∀ A B : Type, A = B → A → B`, to make this definition typecheck. The two type equalities passed to `cast` are

```
Lemma STyExp_cast1 : ∀u w (sub: SubT u w) (env: Env u)
                                    (ty : Ty (S u)) (ty' : Ty u),
  @eq Type
  (Exp (STyE sub env) (STyT [| STyT sub ty' |] (STyT (STyL sub) ty)))
  (Exp (STyE sub env) (STyT sub (STyT [| ty' |] ty))).

Lemma STyExp_cast2 : ∀u w (sub:SubT u w) (env:Env u) (ty:Ty (S u)),
  @eq Type
  (Exp (STyE (STyL sub) (STyE (shSubT u) env)) (STyT (STyL sub) ty))
  (Exp (STyE (shSubT _) (STyE sub env)) (STyT (STyL sub) ty)).
```

which establish (with a one-line proof in each case) that the inferred and declared types are actually equal in the `STyExp` clauses for type application and type abstraction, respectively.

Where Coq definitions can easily be rephrased in a way that avoids the necessity to do this kind of casting, they usually should be. When they cannot, however, we have sometimes found ourselves overwhelmed by the complexities of trying to do more-or-less *ad hoc*, on the fly dependent rewrites with various forms of proof irrelevance (be they axiomatic or proved). Our formalization of polymorphic lambda calculus makes more disciplined and stylised use of the weapon of heterogeneous equality [19], which turns out to be more effective than aimless slashing.

We prove lemmas that all our definitions are congruences with respect to `JMeq` in their non-trivially dependent arguments, and Leibniz equality in the others. These lemmas are tedious to state but essentially just boilerplate: some simple custom tactics prove them immediately, and there seems no reason why they should not be generated automatically. Here is an example:

```
Lemma APP_JMcong: ∀u (env env': Env u) ty1 ty2 ty1' ty2'
               (v1 :Exp env (ARR ty1 ty2)) (v2 : Exp env ty1)
               (v1':Exp env' (ARR ty1' ty2')) (v2': Exp env' ty1'),
  JMeq v1 v1' → JMeq v2 v2' → ty1 = ty1' → ty2 = ty2' → env = env'
       → JMeq (APP v1 v2) (APP v1' v2').
Proof. intros. JMsubst. reflexivity. Qed.
```

Now those standard lemmas about renamings and substitutions that would not otherwise typecheck can be expressed using `JMeq`. For example, the lemma that the action on expressions of the composition of two type substitutions is the composition of the actions looks like this:

```
Lemma STyExp_ss: ∀u env ty (exp: Exp env ty) v w
                          (sub2:SubT v w) (sub1:SubT u v),
  JMeq (STyExp sub2 (STyExp sub1 exp))
       (STyExp (sub2 @ss@ sub1) exp).
```

where `@ss@` is notation for the composition of type substitutions. The proofs of these lemmas are straightforward inductions, essentially just as before, except for the application of the appropriate (boilerplate) congruence property for each constructor. In the cases where the type of the constructor involves a cast, however, we can now easily 'absorb' the specific cast appearing in the type into the more uniform and generic `JMeq` judgement that we are trying

to prove using some simple lemmas such as the following, which removes a cast from the left hand side of the goal:

```
Lemma cast_elim_cong : ∀(A B C:Type) (pf: A = C) (a:A) (b:B),
                         JMeq a b → JMeq (cast pf a) b.
```

Some of the lemmas whose proofs make use of heterogeneous equality internally only mention Leibniz equality in their statements, but this is not the case for all the lemmas that one needs in applications. So the uses of `JMeq` do 'leak out' of the syntax module into clients, rather than being encapsulated as one might hope. In our work on compiler correctness for a polymorphic language [4], for example, the definitions of logical relations between high-level and low-level programs do explicitly involve heterogeneous equality. However, working with `JMeq` uniformly, and from the start, does seem to us to work well, especially compared to pushing particular type equalities around. The basic definitions and lemmas concerning types, terms, renamings and substitutions for System F come in at around 910 lines of Coq, which does not seem *completely* unreasonable.

9 Discussion

We have explained how to define and work with strongly-typed term representations of both simply-typed and polymorphic languages in Coq. The key ideas include the bootstrapping of definitions and lemmas about substitutions in terms of their counterparts for the simpler notion of renamings, and the uniform use of heterogeneous equality in the case of quantified types.

We have used intrinsically typed representations like these in formalizing and proving some non-trivial results about the semantics and compilation of typed languages. Our experience has been that the initial complexity over an extrinsic representation really does pay off - one gets all the stuff to do with the static type system out of the way in the beginning and then when it comes to doing interesting things with those terms, the type system becomes a useful form of scaffolding, with the metalanguage type checker helping ensure that definitions and lemmas make sense, rather than a constant nagging extra obligation.

The use of coercions and `JMeq` in the formalization of the polymorphic language is still slightly inconvenient. The second author has recently designed and implemented a Coq library, Heq,[5] for working with a heterogeneous equality, `==`, based on equality of dependent pairs. Heq supports convenient rewriting with heterogeneous equalities and the manipulation of coercions, and provides `==`-aware versions of tactics such as `subst`. Using Heq, the formalization of the basic theory of System F drops to only 610 lines of Coq; a strong normalization proof [14] is formalized in another 470 lines.

It is possible to work with encodings that are 'partially' intrinsic. Some researchers have used syntax definitions that are well-*scoped* (i.e. whose types express an upper bound on their free de Bruijn indices) by construction, but are still actually typed by an extrinsic typing relation [7,1,17]. This seems particularly natural if one is trying to formalize type theory within type theory, as it is common to treat dependency by working with 'pre-terms' in the first instance. At the other end of the complexity spectrum, however, several researchers have recently presented entirely intrinsic formulations of dependently-typed languages [13, 9,20].

[5] C.-K. Hur: Heq: A Coq library for heterogeneous equality. `http://www.mpi-sws.org/~gil/Heq/` (2010).

There are a number of possible variations on the techniques we have used here. One possibility is to re-examine the 'Haskell-style' definition of shifting in terms of substitution. Although this is not structurally recursive, one *could* define it in Coq using well-founded induction, but we have not yet investigated how easy such a definition would be to work with. One can also represent intrinsically typed syntax in Coq using various kinds of higher-order abstract syntax. This style is already common in Twelf, and Chlipala [11], for example, has done similar things in Coq.

References

1. Adams, R.: Formalized metatheory with terms represented by an indexed family of types. In: Types for Proofs and Programs, *Lecture Notes in Computer Science*, vol. 3839, pp. 1–16. Springer (2006)
2. Altenkirch, T., Reus, B.: Monadic presentations of lambda terms using generalized inductive types. In: Computer Science Logic, 13th International Workshop (CSL'99), *Lecture Notes in Computer Science*, vol. 1683, pp. 453–468. Springer (1999)
3. Benton, N., Hur, C.K.: Biorthogonality, step-indexing and compiler correctness. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP'09), pp. 97–108. ACM (2009)
4. Benton, N., Hur, C.K.: Realizability and compositional compiler correctness for a polymorphic language. Tech. Rep. MSR-TR-2010-62, Microsoft Research (2010)
5. Benton, N., Kennedy, A.J., Varming, C.: Some domain theory and denotational semantics in Coq. In: Theorem Proving in Higher Order Logics, 22nd International Conference (TPHOLs'09), *Lecture Notes in Computer Science*, vol. 5674, pp. 115–130. Springer (2009)
6. Bird, R., Meertens, L.: Nested datatypes. In: Proceedings of the 4th International Conference on Mathematics of Program Construction (MPC'98), *Lecture Notes in Computer Science*, vol. 1422, pp. 52–67. Springer (1998)
7. Bird, R., Paterson, R.: de Bruijn notation as a nested datatype. Journal of Functional Programming **9**, 77–91 (1999)
8. Brady, E., Hammond, K.: A verified staged interpreter is a verified compiler. In: Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE'06), pp. 111–120. ACM (2006)
9. Chapman, J.: Type theory should eat itself. In: International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'08), *Electr. Notes Theor. Comput. Sci.*, vol. 228, pp. 21–36. Elsevier (2009)
10. Cheney, J., Hinze, R.: First-class phantom types. Tech. Rep. TR2003-1901, Cornell University (2003)
11. Chlipala, A.: Parametric higher-order abstract syntax for mechanized semantics. In: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08), pp. 143–156. ACM (2008)
12. Coquand, C.: A formalized proof of the soundness and completeness of a simply typed lambda-calculus with explicit substitutions. Higher-Order and Symbolic Computation **15**, 57–90 (2002)
13. Danielsson, N.A.: A formalization of a dependently typed language as an inductive-recursive family. In: Types for Proofs and Programs, *Lecture Notes in Computer Science*, vol. 4502, pp. 93–109. Springer (2007)
14. Girard, J.Y., Lafont, Y., Taylor, P.: Proofs and Types, *Cambridge Tracts in Theoretical Computer Science*, vol. 7. CUP (1989)
15. Goguen, H., McKinna, J.: Candidates for substitution. Tech. Rep. ECS-LFCS-97-358, University of Edinburgh (1997)
16. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the ACM **40**(1), 143–184 (1993)
17. Hirschowitz, A., Maggesi, M.: Nested abstract syntax in Coq. Journal of Automated Reasoning (2010). URL `http://dx.doi.org/10.1007/s10817-010-9207-9`
18. Kennedy, A., Russo, C.: Generalized algebraic datatypes and object oriented programming. In: ACM Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'05), pp. 21–40. ACM (2005)
19. McBride, C.: Elimination with a motive. In: Types for Proofs and Programs, *Lecture Notes in Computer Science*, vol. 2277, pp. 197–216. Springer (2002)
20. McBride, C.: Outrageous but meaningful coincidences (dependently type-safe syntax and evaluation). In: Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming (WGP'10). ACM (2010)

21. McBride, C., McKinna, J.: The view from the left. Journal of Functional Programming **14**(1), 69–111 (2004)
22. Sheard, T.: Languages of the future. In: Companion to the 19th ACM Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'04), pp. 116–119. ACM (2004)
23. Sozeau, M.: PROGRAM-ing finger trees in COQ. In: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP'07), pp. 13–24. ACM (2007)
24. Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. In: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03), pp. 224–235. ACM (2003)