Generic Programming and Proving for Programming Language Metatheory *

Adam Chlipala

University of California, Berkeley

adamc@cs.berkeley.edu

Abstract

We present a system for both the generic programming of operations that work over classes of tree-structured data types and the automatic generation of formal type-theoretical proofs about such operations. The system is implemented in the Coq proof assistant, using dependent types to validate code and proof generation statically, quantified over all possible input data types. We focus on generic programming of variable-manipulating operations, such as substitution and free variable set calculation, over abstract syntax tree types implemented as GADTs that combine syntax and typing rules. By accompanying these operations with generic lemmas about their interactions, we significantly ease the burden of formalizing programming language metatheory. Our implementation strategy, based on *proof by reflection*, requires users to trust none of its associated code to be able to trust in the validity of theorems derived with it.

1. Introduction

Idiomatic functional programs use a variety of different types of tree-structured data. For instance, we commonly encounter the type of lists, where each list is either the empty list [] or a new element concatenated onto the beginning of a list with the binary :: operator. Tree-structured data is easy to analyze with recursively-defined functions, as in this definition of a function sizeL to calculate the number of "operations" required to construct a list:

$$\begin{aligned} \mathsf{sizeL}\ [] &= 1 \\ \mathsf{sizeL}\ (\_ :: t) &= 1 + \mathsf{sizeL}\ t \end{aligned}$$

We can define a similar function for binary trees, built using the two *constructors* Leaf and Node.

$$\begin{aligned} \mathsf{sizeT}\ (\mathsf{Leaf}\ \_) &= 1 \\ \mathsf{sizeT}\ (\mathsf{Node}\ t_1\ t_2) &= 1 + \mathsf{sizeT}\ t_1 + \mathsf{sizeT}\ t_2 \end{aligned}$$

The pattern extends to the abstract syntax trees of a small language of arithmetic expressions:

$$\begin{aligned} \mathsf{sizeE}\ (\mathsf{Const}\ \_) &= 1 \\ \mathsf{sizeE}\ (\mathsf{Neg}\ e) &= 1 + \mathsf{sizeE}\ e \\ \mathsf{sizeE}\ (\mathsf{Plus}\ e_1\ e_2) &= 1 + \mathsf{sizeE}\ e_1 + \mathsf{sizeE}\ e_2 \end{aligned}$$

While the definitions of these functions are necessarily particular to the data types on which they operate, we see a common pattern among these examples. The types we are working with are all *algebraic datatypes* as supported in the popular functional programming languages Haskell and ML. There is a generic recipe for reading off one of these function definitions from the description of any algebraic datatype: for each constructor of the datatype, we include a case that adds one to the sum of the function's recursive values on the immediate arguments of that constructor.

Now imagine that we want to extend our last code example to deal with the abstract syntax of a full, industrial-strength programming language. We find ourselves stuck with the mindless work of applying the size recipe manually for every construct of the language we are implementing. Even worse, there are at least several other such *generic functions* that often appear in compilers and other language-manipulating tools.

Even outside of functional programming and programming language implementation, we find similar problems in the manipulation of structured XML data. Must we really accept the software engineering nightmare that comes from continual manual effort keeping generic functions in sync with the data types that they manipulate? Luckily, the answer is no, as the field of *generic programming* suggests many usable approaches to writing code that is polymorphic in the structure of data types.

One of the most popular of these approaches today is the *scrap your boilerplate (SYB)* family of techniques [LP03, LP04], which has the advantage of being usable in popular Haskell compilers with no new language extensions. SYB achieves genericity through a combination of ad-hoc code generation at compile time and "unsafe" type coercions at run time. The result integrates spectacularly well with existing practical programming environments. The necessity for type coercions remains as a skeleton in the family closet whose face we never expect to see in practice.

Unfortunately, there is one sort of activity guaranteed to shine the light of day upon the dark secrets inherent in any programming technique, and that is formal verification. In mechanized programming language metatheory of the kind highlighted recently by the POPLmark Challenge [ABF+05], we set out to write proofs about programs and programming languages that are rigorous enough to convince a mechanical proof checker. What happens when we want to prove the correctness of the compiler we implement with one of today's popular generic programming approaches? Unchecked type coercions and formal mathematical modeling tend not to mix very well.

We also find ourselves looking for a new category of functionality: *generic proofs* about our generic programs. A study of the solutions submitted for the POPLmark Challenge's first benchmark problem [ACPW08] provides some statistics on the composition of formal developments. Examining the solutions to just the first step of the challenge problem, the authors find that none of the 7 submissions that use the Coq proof assistant achieves the task without proving at least 22 auxiliary lemmas. In contrast, the reference "pencil and paper" solution included with the problem description introduces only 2 such lemmas. The remaining facts are taken to be so obvious that all practicing programming language researchers

* This research was supported in part by a National Defense Science and Engineering Graduate Fellowship, as well as National Science Foundation grants CCF-0524784 and CCR-0326577.

will accept them without proof or even explicit statement as theorems.

While the benefits of computer formalization of proofs about programming languages are widely accepted, most people writing proofs find that today's computer theorem proving tools make it more trouble than it is worth to take this extra step. One manifestation of the difficulties is the need to prove the generic and "obvious" lemmas that we alluded to above. In this paper, we present a tool for building some of these proofs automatically, freeing the human prover to focus on "the interesting parts."

We introduced this system in a previous paper [Chl07] about a compiler from lambda calculus to assembly language, implemented in Coq with a proof of total correctness. There, we devoted less than a page to it, summarizing how a user of the tool sees it. In this paper, we present the implementation of this tool, which we call the AutoSyntax component of the Lambda Tamer system for language formalization. We also generalize some aspects of its design to the broader problem of generic programming.

1.1 Outline

In the next section, we present related work in generic programming and situate our new contribution within it. Following that, we introduce the relevant aspects of Coq's underlying programming language. Next, we present our technique for the case of simply-typed algebraic datatypes. With the foundation laid, we summarize our approach to formalizing programming languages using dependent types, demonstrate the difficulties with constructing support code and proofs manually, and show how to adapt our generic programming and proving approach to lifting that burden.

The final implementation sketched in this paper is available in full on the Web at:

`http://ltamer.sourceforge.net/`

We have changed some details in this paper for clarity, but the basic structure is the same.

2. Related Work

In dynamically-typed languages like those in the Lisp family, simple reflective capabilities make it clear how to write generic functions, but the lack of static validation can make it difficult to get their implementations right. Approaches exist [LV02] for statically-typed languages that convert to untyped form and back around generic function invocations, but they retain most of the same deficiency.

Several language designs exist that embody *polytypic programming* [JJ97, Hin00], where functions can be defined over the structure of types. Derivable type classes [HP00] is a polytypic programming extension to Haskell, designed to tackle the practical issues of extending a production language. These systems are not designed to support theorem proving. The Generic and Indexed Programming project [GWdSO07] is engaged in bridging some of that gap.

Recent tools designed to work with out-of-the-box Haskell (with GHC extensions) include Scrap Your Boilerplate [LP03, LP04] and RepLib [Wei06]. RepLib in particular uses reflected type representations very similar to ours. Both approaches benefit from easy integration with a standard programming language, but, because of Haskell's relatively weak type system, they must resort to unsafe type casts in their implementations. Both support mixing generic function implementations that are read off directly from type structure with custom implementations for particular types. Again, neither handles generic proving.

There is a significant body of work on implementing generic programming and proving in type theory using *universe types*. Here, the idea is to define reflected representations of type structure that can be manipulated programmatically. The popular dependent type theories are expressive enough to allow us to use type checking to validate operations over the values of well-chosen universe types. A nice consequence of this is that we do not need to trust in the correct implementation of our generic programs and proofs. Some out-of-band mechanism is used to translate "real" types into their reflected representations, but the result of this process is only a *hint*, so it need not be trusted, either.

Work in this vein includes that by Altenkirch and McBride [AM03], which considers an approach reminiscent of what we present in Section 4 (minus the generic proving) for the OLEG proof assistant. Pfeifer and Rueß [PR99] and Benke et al. [BDJ03] go further and add (in different ways) rudimentary generic proving to the same general program for LEGO and Agda, respectively. Recent work on inductive-recursive definitions [DS06] facilitates more natural definitions of universe types, using an encoding in the spirit of higher-order abstract syntax. Morris et al. [MAG07] provide a very general universe type for the strictly positive inductive families and use it to implement generic programming support in Epigram.

This body of work deals with classes of datatypes in general, limited only by the demands of encoding them in type theory. We build on this work and specialize it to the domain of programming language metatheory, where we have interesting generic operations that do not make sense over datatypes in general. Our system AutoSyntax is specialized to our language representation approach, which fixes the de Bruijn convention for variables. It could be adapted to alternate first-order choices, including those based on nominal logic [UT05]. Higher-order abstract syntax (HOAS) [PE88] is a completely different variable representation technique that uses the variables of the meta language to stand for the variables of the object language. Thus, notions like substitution and weakening are inherited "for free" from the meta language. HOAS is incompatible with CIC, but some generalization of our approach may still be possible to facilitate different kinds of generic programming and proving. We note that it may be worth the pain of explicit variable management to retain Coq's support for statically-validated manipulation of first-class meta-proofs.

We introduced AutoSyntax in one page of previous work [Chl07]. In the current paper, we present the implementation details for the first time, including type theoretic encoding techniques and generic proof sketches.

$$
\begin{array}{rlll}
\text{Variables} & x & & \\
\text{Naturals} & n & \in & \mathbb{N} \\
\text{Terms} & E & ::= & x \mid E\ E \mid \lambda x : E.\ E \mid \forall x : E.\ E \mid \mathsf{Type}_n
\end{array}
$$

Figure 1. Syntax of CoC

3. Background: The Calculus of Inductive Constructions

The Coq proof assistant [BC04] is a type-theoretical theorem proving and certified programming tool based on an underlying logic/programming language, the *Calculus of Inductive Constructions (CIC)*. CIC is a *logic* because it can be used to state approximately all theorems and express approximately all proofs that show up in mathematics. CIC is a *programming language* because it is a typed lambda calculus that can be used to write *certified programs*, using the language's mathematical capabilities to express strong specifications through typing. As Coq is a mature tool for theorem proving and certified program construction, we chose to use it to implement AutoSyntax.

In this section, we give an overview of CIC, starting with its simpler subset, the Calculus of Constructions (CoC). Figure 1 presents the abstract syntax of CoC. We can recognize the elements

$$\frac{\Gamma(x) = E}{\Gamma \vdash x : E} \qquad \frac{\Gamma \vdash E_1 : (\forall x : E_{\mathsf{dom}}.\ E_{\mathsf{ran}}) \quad \Gamma \vdash E_2 : E_{\mathsf{dom}}}{\Gamma \vdash E_1\ E_2 : E_{\mathsf{ran}}[x \mapsto E_2]}$$

$$\frac{\Gamma \vdash E_{\mathsf{dom}} : \mathsf{Type}_n \quad \Gamma, x : E_{\mathsf{dom}} \vdash E : E_{\mathsf{ran}}}{\Gamma \vdash (\lambda x : E_{\mathsf{dom}}.\ E) : (\forall x : E_{\mathsf{dom}}.\ E_{\mathsf{ran}})}$$

$$\frac{\Gamma \vdash E_{\mathsf{dom}} : \mathsf{Type}_n \quad \Gamma, x : E_{\mathsf{dom}} \vdash E_{\mathsf{ran}} : \mathsf{Type}_n}{\Gamma \vdash (\forall x : E_{\mathsf{dom}}.\ E_{\mathsf{ran}}) : \mathsf{Type}_n}$$

$$\frac{}{\Gamma \vdash \mathsf{Type}_n : \mathsf{Type}_{n+1}} \qquad \frac{\Gamma \vdash E : \mathsf{Type}_n}{\Gamma \vdash E : \mathsf{Type}_{n+1}}$$

$$\frac{\Gamma \vdash E_1 : E_2' \quad E_2' \equiv E_2}{\Gamma \vdash E_1 : E_2}$$

Figure 2. Typing rules for CoC

$$\frac{}{(\lambda x : E_{\mathsf{dom}}.\ E_1)\ E_2 \equiv E_1[x \mapsto E_2]}$$

$$\frac{E_1 \equiv E_1' \quad E_2 \equiv E_2'}{E_1\ E_2 \equiv E_1'\ E_2'} \qquad \frac{E_1 \equiv E_1' \quad E_2 \equiv E_2'}{\lambda x : E_1.\ E_2 \equiv \lambda x : E_1'.\ E_2'}$$

$$\frac{E_1 \equiv E_1' \quad E_2 \equiv E_2'}{\forall x : E_1.\ E_2 \equiv \forall x : E_1'.\ E_2'}$$

$$\frac{}{E \equiv E} \qquad \frac{E' \equiv E}{E \equiv E'} \qquad \frac{E \equiv E' \quad E' \equiv E''}{E \equiv E''}$$

Figure 3. Definitional equality rules for CoC

of the simply-typed lambda calculus and System F, but CoC has an important difference from these systems. "Terms" and "types" are collapsed into a single syntactic class. Thus, alongside variables and function abstraction and application, we find the function type form $\forall x : E_1.\ E_2$, denoting a function with domain E_1 and range E_2. This is a *dependent function type* because the variable x is bound in E_2 to the *value* of the function argument. We write $E_1 \rightarrow E_2$ as a shorthand for $\forall x : E_1.\ E_2$ when x is not free in E_2.

Figure 2 gives the complete set of typing rules for CoC. The most interesting part of Figure 2 is the last rule, which says that if two terms are *definitionally equal*, then, when considered as types, they describe the same set of terms. Figure 3 presents the definitional equality relation $\equiv$, an equivalence relation that models usual computational reduction rules. A crucial property of this definition of $\equiv$ is that it is decidable when restricted to well-typed terms. Here we see a standard property of dependently-typed languages like CoC: each of the static and dynamic semantics depends on the other in a key way.

We also have terms Type_n denoting different levels in an infinite type hierarchy. Figure 2 shows that any Type_n has type Type_m for any $m > n$. That is, types themselves are first-class entities that are classified by other types. Because of this, we do not need separate constructs for normal functions and parametric polymorphism, as in System F. Rather, we can write System F-style terms using CoC restricted to using only type level 0 in our definitions. For instance, $\lambda \tau : \mathsf{Type}_0.\ \lambda x : \tau.\ x$ is the polymorphic identity function and has type $\forall \tau : \mathsf{Type}_0.\ \tau \rightarrow \tau$.

Readers accustomed to System F may be growing irritated at the necessity to calculate the right Type indices to include in various places. System F supports *impredicative* type quantification, where quantified types may be instantiated with themselves. Type indices in CoC as presented here forbid such circularities. There is an impredicative variant of CIC that may be turned on in Coq by way of a command-line flag, and, in fact, even the standard language supports a special impredicative sort for logical propositions. We stick to the completely *predicative* version here for simplicity of exposition.

3.1 Inductive Types

If we are just concerned with "programming" of the traditional kind, then CoC is quite sufficient for encoding a wide variety of features, following the standard trick of "encoding types with their polymorphic elimination forms." However, we run into trouble when we attempt to use the same system to encode formal proofs about these programs, let alone proof-manipulating, dependently-typed programs of the kinds that will appear later. Basic properties of data structures turn out unprovable, and we encounter algorithmic inefficiencies with encoding schemes that try to fit so many features into so simple a base language. Additionally, it sometimes turns out that the natural encodings of data types are inhabited by more terms than we expected them to be. For these reasons, Coq was enriched with special support for what are called *inductive types* [PM93], a very general type definition mechanism that can be used to define all of the standard data types and logical connectives. The enhanced logic is called the Calculus of Inductive Constructions (CIC).

Rather than provide an exhaustive formalization, we will demonstrate inductive types by example, relating the examples to the formalization of CoC.

Inductive types subsume the algebraic datatypes found in subsets of Haskell and ML that disallow infinite data structures and non-terminating programs, as demonstrated by this definition of the natural numbers:

$$\begin{array}{l} \mathsf{Inductive\ nat} : \mathsf{Type}_0 := \\ \quad \mid \mathsf{O} : \mathsf{nat} \\ \quad \mid \mathsf{S} : \mathsf{nat} \rightarrow \mathsf{nat} \end{array}$$

We define a type named nat at type level 0 by describing how its values may be built. Its two *constructors* are O, the natural number 0; and S, the function from a natural number to its successor. Thus, the natural numbers are denoted O, S O, S (S O), etc..

We can perform pattern matching on nats:

$$\begin{array}{rcl} \mathsf{pred} & = & \lambda x : \mathsf{nat}.\ \mathsf{match}\ x\ \mathsf{with} \\ & & \quad \mid \mathsf{O} \Rightarrow \mathsf{O} \\ & & \quad \mid \mathsf{S}\ n \Rightarrow n \end{array}$$

...and we can write recursive functions over them:

$$\begin{array}{rcl} \mathsf{plus} & = & \mathsf{fix}\ f\ (n : \mathsf{nat}) : \mathsf{nat} \rightarrow \mathsf{nat}.\ \mathsf{match}\ n\ \mathsf{with} \\ & & \quad \mid \mathsf{O} \Rightarrow \lambda n' : \mathsf{nat}.\ n' \\ & & \quad \mid \mathsf{S}\ n \Rightarrow \lambda n' : \mathsf{nat}.\ \mathsf{S}\ (f\ n\ n') \end{array}$$

Coq enforces that every recursive call in one of these fix expressions passes a recursive argument that is a syntactic subterm of the current argument. This retains the important property of CoC that every program "terminates" in a suitable sense, allowing us to keep the relation $\equiv$ decidable and preclude such tricks as presenting an infinite loop as a "proof" of any proposition.

We can also define parameterized inductive types, such as lists:

$$\begin{array}{l}\textsf{Inductive list } (T : \textsf{Type}_0) : \textsf{Type}_0 := \\ \quad \mid \textsf{nil} : \textsf{list } T \\ \quad \mid \textsf{cons} : T \rightarrow \textsf{list } T \rightarrow \textsf{list } T\end{array}$$

...and write recursive functions over them:

$$\begin{array}{rcl}\textsf{length} & = & \textsf{fix } f\ (T : \textsf{Type}_0)\ (\ell : \textsf{list } T) : \textsf{nat. match } \ell \textsf{ with} \\ & & \quad \mid \textsf{nil} \Rightarrow O \\ & & \quad \mid \textsf{cons } \_\ \ell' \Rightarrow \textsf{S}\ (f\ T\ \ell')\end{array}$$

Inductive types also generalize the *generalized algebraic datatypes (GADTs)* [She04] that have recently crept into reasonably wide use through a popular extension to Haskell. For example, consider this definition of a list-like type family indexed by natural numbers providing strict bounds on list lengths.

$$\begin{array}{l}\textsf{Inductive listN } (T : \textsf{Type}_0) : \textsf{nat} \rightarrow \textsf{Type}_0 := \\ \quad \mid \textsf{nilN} : \textsf{listN } T \textsf{ O} \\ \quad \mid \textsf{consN} : \forall n : \textsf{nat}.\ T \rightarrow \textsf{listN } T\ n \rightarrow \textsf{listN } T\ (\textsf{S } n)\end{array}$$

The only value of $\textsf{listN}\ T\ \textsf{O}$ is the nil list, every value of $\textsf{listN}\ T\ (\textsf{S O})$ is a one-element list, etc.. In contrast to the case of GADTs, where all parameters of the type being defined must be types, inductive types allow arbitrary "term-level" data to appear in type indices. This lets us use the types of functions to be much more specific about their behavior than in traditional programming:

$$\begin{array}{rcl}\textsf{countdown} & = & \textsf{fix } f\ (n : \textsf{nat}) : \textsf{listN nat } n.\ \textsf{match } n \textsf{ with} \\ & & \quad \mid \textsf{O} \Rightarrow \textsf{nilN nat} \\ & & \quad \mid \textsf{S } n' \Rightarrow \textsf{consN nat } n'\ n\ (f\ n')\end{array}$$

Actual Coq code must be a bit more verbose about this, with extra type annotations to make type inference tractable. However, we will omit those details in this paper in the interest of clarity.

We can also take advantage of the possibility to create data structures that contain types. Here is an example of defining the type of lists of types, followed by the type family of tuples whose component types are read off from particular type lists.

$$\begin{array}{l}\textsf{Inductive listT} : \textsf{Type}_1 := \\ \quad \mid \textsf{nilT} : \textsf{listT} \\ \quad \mid \textsf{consT} : \textsf{Type}_0 \rightarrow \textsf{listT} \rightarrow \textsf{listT}\end{array}$$

$$\begin{array}{l}\textsf{Inductive tuple} : \textsf{listT} \rightarrow \textsf{Type}_2 := \\ \quad \mid \textsf{Nil} : \textsf{tuple nilT} \\ \quad \mid \textsf{Cons} : \forall T : \textsf{Type}_0.\ \forall \ell : \textsf{listT}. \\ \qquad T \rightarrow \textsf{tuple } \ell \rightarrow \textsf{tuple } (\textsf{consT } T\ \ell)\end{array}$$

As an example, one value with type $\textsf{tuple}\ (\textsf{consT nat}\ (\textsf{consT}\ (\textsf{nat} \rightarrow \textsf{nat})\ \textsf{nilT}))$ is $\textsf{Cons nat}\ (\textsf{consT}\ (\textsf{nat} \rightarrow \textsf{nat})\ \textsf{nilT})\ O\ (\textsf{Cons}\ (\textsf{nat} \rightarrow \textsf{nat})\ \textsf{nilT}\ (\lambda x : \textsf{nat}.\ x)\ \textsf{Nil})$. Coq supports an *implicit argument* facility that lets us write this term as $\textsf{Cons}\ O\ (\textsf{Cons}\ (\lambda x : \textsf{nat}.\ x)\ \textsf{Nil})$. Though we will not explain the details of implicit arguments here, we will trust in the reader's intuition as we freely omit inferable arguments in the following discussion.

Note that both inductive type families that we define here exist at type levels above 0. The rule in effect here is that an inductive type must live at at least type level $n + 1$ if one of its constructors' types itself has type $\textsf{Type}_{n+1}$ but not type $\textsf{Type}_n$. The intuition is that inductive definitions end up at higher type levels as they employ more sophisticated uses of "types as data." In actual Coq code, Type indices are inferred automatically, and the user only needs to worry about them when the Coq system determines that no feasible solution exists to an induced set of constraints on index variables. Following that discipline, we will use Type unadorned in the remainder of this paper.

3.2 Proof by Reflection

The definitional equality $\equiv$ has no counterpart in the type-checking of mainstream programming languages. With the addition of inductive types in CIC, $\equiv$ grows to include rules for simplifying pattern matching and recursive function application. The CIC type-checker will now interpret quite complex programs in the course of validating larger programs that use them as types. Why is it worth designing a language with this variety of "busy types"?

One compelling answer comes from the technique of proof by reflection [Bou97], which is related to the idea of universes introduced by Martin-Löf [ML84]. It is a subtle approach to efficient encoding of proofs, best introduced by example. Let us consider families of proofs for this (tautological) class of formulas:

$$F \quad ::= \quad \textsf{True} \mid F \wedge F$$

The Coq proof terms for this class are natural deduction-style proof trees that mirror the structure of the formulas themselves. Since any general sort of type inference is undecidable for CIC, the forms in which these proof trees are stored contain many "redundant" type annotations, leading to representation sizes superlinear in the sizes of the original formulas. This seems a harsh price when we could just write down an *algorithm* for generating a formula's proof from its structure. When asked to prove a formula in F, why not just say "run this algorithm and see for yourself"? In fact, proof by reflection allows us to do more or less that.

We want to build a trivial "decision procedure" for F. In this case, it should always answer "the formula is true," but the interesting catch is that the procedure must be *proof-producing*. To make the proof by reflection idiom work, the procedure must be *implemented in CIC*. When we use the right dependent typing, this amounts to proving the correctness of the procedure.

Following this path, we run into a block early on. In Coq, general logical formulas are represented in Type. Type is *open*, because new inductive definitions (modeling new logical connectives, etc.) may add new ways of building Types. Thus, CIC provides no way of pattern matching on Types. We cannot write a function that deconstructs formulas programmatically, so how can our implementation possibly know which proof to build?

The trick that we use instead is the source of the description "reflection." It is related to "reflection" in popular managed object-oriented languages like Java and C#, where it is possible to obtain a runtime "data level" view of type information. In our case, we define a data structure for describing precisely the subset of Types that interests us.

$$\begin{array}{l}\textsf{Inductive F} : \textsf{Type} := \\ \quad \mid \textsf{F\_True} : \textsf{F} \\ \quad \mid \textsf{F\_And} : \textsf{F} \rightarrow \textsf{F} \rightarrow \textsf{F}\end{array}$$

Along with an injection into the original domain of interest:

$$\begin{array}{rcl}\textsf{interp} & = & \textsf{fix } f\ (x : \textsf{F}) : \textsf{Type. match } x \textsf{ with} \\ & & \quad \mid \textsf{F\_True} \Rightarrow \textsf{True} \\ & & \quad \mid \textsf{F\_And } x_1\ x_2 \Rightarrow f(x_1) \wedge f(x_2)\end{array}$$

Now we can write a prover for this class of formulas:

$$\textsf{prove} \quad : \quad \forall x : \textsf{F}.\ \textsf{interp } x$$

Now, for example, with

$$x = \textsf{F\_And F\_True (F\_And F\_True F\_True)}$$

we have $\vdash \textsf{prove } x : \textsf{True} \wedge (\textsf{True} \wedge \textsf{True})$.

In type-checking this expression, Coq uses the definitional equality $\equiv$ to simplify $\textsf{interp}\ x$ to $\textsf{True} \wedge (\textsf{True} \wedge \textsf{True})$. We could never use the same technique to prove, for instance, falsehood, because that formula is not in the range of interp. Using the definitional equality further to simplify prove x, we arrive at exactly the large proof that we would have constructed manually. The point is that the proof checker need not perform this simplification. It can content itself with verifying that this term has the proper type, not venturing into its innards.

4. Generic Programming and Proving for Simply-Typed Data Structures

Recall the family of examples from the introduction. We listed a number of instantiations of a hypothetical generic "size" function. For instance, for a type of trees of natural numbers, we have:

$$\begin{array}{rcl} \mathsf{sizeT}\ (\mathsf{Leaf}\ \_) & = & 1 \\ \mathsf{sizeT}\ (\mathsf{Node}\ t_1\ t_2) & = & 1 + \mathsf{sizeT}\ t_1 + \mathsf{sizeT}\ t_2 \end{array}$$

With the additional language constructs introduced since then, we can give an inductive definition of this data type.

$$\begin{array}{l} \mathsf{Inductive\ tree} : \mathsf{Type} := \\ \quad \mid \mathsf{Leaf} : \mathsf{nat} \to \mathsf{tree} \\ \quad \mid \mathsf{Node} : \mathsf{tree} \to \mathsf{tree} \to \mathsf{tree} \end{array}$$

We want to write a truly generic size function that can be used with any simply-typed algebraic datatype. We imagine that the signature of the generic function should look something like:

$$\mathsf{size} \quad : \quad \forall T : \mathsf{inductiveType}.\ T \to \mathsf{nat}$$

Unfortunately, Coq provides no special type that categorizes inductively-defined types, and, even if it did, we would be left with the challenges of programmatic manipulation of arbitrary datatype definitions. Not surprisingly, the solution that we propose is based on the last section's subject, reflection.

4.1 Reflecting Constructors

What is the essence of an inductive type definition? What reflective representation should we craft for these definitions? Looking at the textual form of an inductive definition, it seems a reasonable start to say that an inductive type is a list of constructors. We can define a reflected representation of constructors like this:

$$\mathsf{con} \quad = \quad \mathsf{nat} \times \mathsf{Type}$$

The idea is that each constructor's list of arguments contains some number that are recursive, referring to the type that's being defined. The nat component of con tells how many recursive arguments the constructor has, and the Type component describes the remaining arguments. It will be a tuple type combining them into one package, or unit (the inductive type whose only value is tt) if all arguments are recursive. For the tree example, Leaf is reflected as $(0, \mathsf{nat})$, and Node as $(2, \mathsf{unit})$.

We need to make the interpretation of constructor descriptions explicit, like we did for our formula type F in the last section. The definition of the interpretation function interpCon relies on an auxiliary recursive function repeat that builds a tuple type by repeating a base type a specified number of times.

$$\begin{array}{rcl} \mathsf{repeat} & = & \lambda T : \mathsf{Type}.\ \mathsf{fix}\ f\ (n : \mathsf{nat}) : \mathsf{Type}.\ \mathsf{match}\ n\ \mathsf{with} \\ & & \mid \mathsf{O} \Rightarrow \mathsf{unit} \\ & & \mid \mathsf{S}\ n' \Rightarrow T \times f\ n' \end{array}$$

$$\begin{array}{rcl} \mathsf{interpCon} & = & \lambda T : \mathsf{Type}.\ \lambda c : \mathsf{con}. \\ & & \quad \mathsf{repeat}\ T\ (\pi_1\ c) \to \pi_2\ c \to T \end{array}$$

Now we are almost ready to state our initial reflective representation of entire inductive definitions. We first need to define two common inductive types. The first type, sig, is the standard "sigma type," or "existential package," of type theory, taken from the Coq standard library. The second is a generalization to existential packages whose first components are lists and whose second components are heterogeneous lists, where the type of each component is determined by applying a fixed function to the element in the corresponding position of the first list.

$$\begin{array}{l} \mathsf{Inductive\ sig}\ (T_1 : \mathsf{Type})\ (T_2 : T_1 \to \mathsf{Type}) : \mathsf{Type} := \\ \quad \mid \mathsf{ex} : \forall x : T_1.\ T_2\ x \to \mathsf{sig}\ T_1\ T_2 \end{array}$$

$$\begin{array}{l} \mathsf{Inductive\ tupleF}\ (T_1 : \mathsf{Type})\ (T_2 : T_1 \to \mathsf{Type}) \\ \quad\quad : \mathsf{list}\ T_1 \to \mathsf{Type} := \\ \quad \mid \mathsf{NilF} : \mathsf{tupleF}\ T_1\ T_2\ \mathsf{nil} \\ \quad \mid \mathsf{ConsF} : \forall x : T_1.\ \forall \ell : \mathsf{list}\ T_1. \\ \quad\quad T_2\ x \to \mathsf{tupleF}\ T_1\ T_2\ \ell \to \mathsf{tupleF}\ T_1\ T_2\ (\mathsf{cons}\ x\ \ell) \end{array}$$

We abbreviate sig T_1 $(\lambda x : T_1.\ T_2)$ as $\Sigma x : T_1.\ T_2$, or $\Sigma x.\ T_2$ in contexts where T_1 is clear. We write $\langle x, y \rangle$ as an abbreviation for ex T_1 T_2 x y when T_1 and T_2 are clear from context. We also treat T_1 as an implicit argument of tupleF, abbreviating tupleF T_1 T_2 ℓ as tupleF T_2 ℓ.

We now define ind : Type $\to$ Type, such that ind T is the type of a reflected representation of inductively-defined T.

$$\mathsf{ind} \quad = \quad \lambda T : \mathsf{Type}.\ \Sigma \ell : \mathsf{list\ con}.\ \mathsf{tupleF}\ (\mathsf{interpCon}\ T)\ \ell$$

Here is the reflected representation rtree of tree.

$$\begin{array}{rcl} \mathsf{clist} & = & [(0, \mathsf{nat}), (2, \mathsf{unit})] \\ \mathsf{cLeaf} & = & \lambda\_ : \mathsf{unit}.\ \lambda n : \mathsf{nat}.\ \mathsf{Leaf}\ n \\ \mathsf{cNode} & = & \lambda r : \mathsf{tree} \times \mathsf{tree} \times \mathsf{unit}.\ \lambda\_ : \mathsf{unit}.\ \mathsf{Node}\ (\pi_1\ r)\ (\pi_2\ r) \\ \mathsf{rtree} & = & \langle \mathsf{clist}, \mathsf{ConsF\ cLeaf\ (ConsF\ cNode\ NilF)} \rangle \end{array}$$

4.2 Reflecting Recursion Principles

With the definition of ind, we can express the desired type of size formally.

$$\mathsf{size} \quad : \quad \forall T : \mathsf{Type}.\ \mathsf{ind}\ T \to T \to \mathsf{nat}$$

We are off to a good start, having replaced the informal quantification over an inductively defined type with a quantification over any type, plus a piece of "evidence" that it really behaves like an inductive type. Unfortunately, the evidence contained in an ind is not yet sufficient to let us write size. The type we have listed for size is good, but we will have to expand the definition of ind.

With an ind T in hand, we know how to *construct* values of T, but we do not yet know how to *deconstruct* values of T; that is, we are not able to write recursive, pattern-matching functions over T. We must expand the definition of ind to require the inclusion of a *recursion principle*.

First, we define a representation for each arm of the pattern matching in a recursive definition. conIH T c gives the type of arms for constructor c of type T, where each is polymorphic in the range R of the function being defined.

$$\begin{array}{rcl} \mathsf{conIH} & = & \lambda T : \mathsf{Type}.\ \lambda c : \mathsf{con}.\ \forall R : \mathsf{Type}. \\ & & \quad \mathsf{repeat}\ (T \times R)\ (\pi_1\ c) \to \pi_2\ c \to R \end{array}$$

Say we are defining some recursive function f of type $T \to R$. The arm of type conIH T c applied to R is given two arguments: first, the recursive arguments to this particular use of constructor c, where each argument comes packaged with the result of calling f on it recursively; and second, the remaining, non-recursive arguments. Such an arm returns something in f's range.

Now we can define the representation of recursion principles, whose job it is to define a recursive function given an arm definition for each constructor of an inductive type:

$$\begin{array}{rcl} \mathsf{rec} & = & \lambda T : \mathsf{Type}.\ \lambda \ell : \mathsf{list\ con}. \\ & & \quad \forall R : \mathsf{Type}.\ \mathsf{tupleF}\ (\lambda c.\ \mathsf{conIH}\ T\ c\ R)\ \ell \to (T \to R) \end{array}$$

$$
\begin{aligned}
\mathsf{size\ tree\ rtree} \quad &\equiv \quad (\mathsf{recOf\ rtree})\ \mathsf{nat}\ (\mathsf{mapF} \\
&\qquad (\lambda c : \mathsf{con}.\ \lambda r.\ \lambda\_.\ \mathsf{foldRepeat} \\
&\qquad\quad (\lambda v : \mathsf{tree} \times \mathsf{nat}.\ \lambda n : \mathsf{nat}.\ \pi_2\ v + n) \\
&\qquad\quad 1\ r) \\
&\qquad (\mathsf{consOf\ rtree}) \\
&\equiv \quad (\mathsf{recOf\ rtree})\ \mathsf{nat} \\
&\qquad (\mathsf{ConsF}\ (\lambda\_.\ \lambda\_.\ 1) \\
&\qquad\quad (\mathsf{ConsF}\ (\lambda r.\ \lambda\_.\ \pi_2\ (\pi_1\ r) \\
&\qquad\qquad + \pi_2\ (\pi_2\ r) + 1)\ \mathsf{NilF})) \\
&\equiv \quad \mathsf{fix}\ f\ (x : \mathsf{tree}) : \mathsf{nat}.\ \mathsf{match}\ x\ \mathsf{with} \\
&\qquad \mid \mathsf{Leaf}\ n \Rightarrow 1 \\
&\qquad \mid \mathsf{Node}\ t_1\ t_2 \Rightarrow f\ t_1 + f\ t_2 + 1
\end{aligned}
$$

Figure 4. Simplifying one application of the generic size function

We can define the recursion principle for tree, overloading the π_i notation to denote the ith projections of tupleFs, not just normal tuples.

$$
\begin{aligned}
\mathsf{rTree} \quad = \quad & \lambda R : \mathsf{Type}.\ \lambda A : \mathsf{tupleF}\ (\lambda c.\ \mathsf{conIH\ tree}\ c\ R)\ \mathsf{clist}. \\
& \mathsf{fix}\ f\ (x : \mathsf{tree}) : R.\ \mathsf{match}\ x\ \mathsf{with} \\
& \quad \mid \mathsf{Leaf}\ n \Rightarrow (\pi_1\ A)\ R\ \mathsf{tt}\ n \\
& \quad \mid \mathsf{Node}\ t_1\ t_2 \Rightarrow (\pi_2\ A)\ R \\
& \qquad ((t_1, f\ t_1), (t_2, f\ t_2), \mathsf{tt})\ \mathsf{tt}
\end{aligned}
$$

Now we are ready to expand the definition of ind by adding a rec component.

$$
\begin{aligned}
\mathsf{ind} \quad = \quad & \lambda T : \mathsf{Type}.\ \Sigma \ell : \mathsf{list\ con}. \\
& \mathsf{tupleF}\ (\mathsf{interpCon}\ T)\ \ell \times \mathsf{rec}\ T\ \ell
\end{aligned}
$$

Define consOf, buildersOf, and recOf as shortcut functions for extracting the three different pieces of an ind.

We can finally define size. In this and later complicated definitions, we will play a little fast and loose with typing, leaving out annotations that Coq really is not able to infer. For instance, we use a function mapF for constructing a tupleF by providing a function to be applied to every element of a list; it will usually be necessary to specify the relevant dependent typing relationship (parameter T_2 of tupleF) explicitly in real code. We also rely on a function foldRepeat to duplicate the behavior of the traditional list right fold operator over tuples built with repeat, which can be thought of as lists of known length. It will also, in practice, require more explicit arguments than we will give here.

$$
\begin{aligned}
\mathsf{size} \quad = \quad & \lambda T : \mathsf{Type}.\ \lambda \iota : \mathsf{ind}\ T. \\
& (\mathsf{recOf}\ \iota)\ \mathsf{nat}\ (\mathsf{mapF} \\
& \quad (\lambda c : \mathsf{con}.\ \lambda r.\ \lambda\_.\ \mathsf{foldRepeat} \\
& \qquad (\lambda v : \mathsf{T} \times \mathsf{nat}.\ \lambda n : \mathsf{nat}.\ \pi_2\ v + n) \\
& \qquad 1\ r) \\
& \quad (\mathsf{consOf}\ \iota))
\end{aligned}
$$

Now we can examine in Figure 4 the simplification behavior of size applied to a revised version of the reflected representation rtree that contains the recursion principle rTree. Modulo some commutativity of addition, we arrive at exactly the definition that we started out with! The definitional equality $\equiv$ is sufficient to simplify instantiations of the generic size function into their natural forms. Thus, we can use the generic and specific versions interchangeably for type-checking (and proof-checking) purposes.

4.3 Generic Proofs

We could stop at this point were we only interested in solving the traditional problems of generic programming. However, the express motivation of this work is to allow the use of generic programs in concert with formal verification. It is high time that we take a look at how we may construct proofs about generic functions.

Obviously we can prove facts about *specific instantiations* of generic functions in the usual way, as the definitional equality allows us to reduce these instantiations to standard forms. However, the real promise of generic techniques in a formal theorem-proving setting is in automating *families* of proofs. We want to be able to prove that certain theorems hold for *any* instantiations of generic functions.

Unfortunately, our definition of the ind evidence packages is not yet sufficient to allow us to prove interesting theorems. We now know how to *construct* values, and we know how to *deconstruct* them via recursive function definitions, but we do not know anything about the behavior of functions that we build in this way. The missing ingredient is a standard *fixed-point equation* decomposing applications of rec-built functions recursively.

$$
\begin{aligned}
\mathsf{eqn} \quad = \quad & \lambda T : \mathsf{Type}.\ \lambda \ell : \mathsf{list\ con}. \\
& \lambda \mathit{constructors} : \mathsf{tupleF}\ (\mathsf{interpCon}\ T)\ \ell. \\
& \lambda f : \mathsf{rec}\ T\ \ell. \\
& \forall R : \mathsf{Type}.\ \forall \mathit{arms} : \mathsf{tupleF}\ (\lambda c.\ \mathsf{conIH}\ T\ c\ R)\ \ell. \\
& \forall c : \mathsf{con}.\ \forall n : \mathsf{nat}.\ \pi_n\ \ell = c \ \rightarrow \\
& \forall r : \mathsf{repeat}\ T\ (\pi_1\ c).\ \forall v : \pi_2\ c. \\
& f\ R\ \mathit{arms}\ ((\pi_n\ \mathit{constructors})\ r\ v) \\
& = (\pi_n\ \mathit{arms})\ R \\
& \quad (\mathsf{mapRepeat}\ (\lambda x : T.\ (x, f\ R\ \mathit{arms}\ x))\ r)\ v
\end{aligned}
$$

eqn is quite a mouthful. It is defined in terms of T, the type we are manipulating; ℓ, its list of reflected constructors; *constructors*, its actual constructors; and f, its recursion principle. We assert an equation for any range type R and any recursive function defined from T to R with pattern-matching arms *arms*. Give the name F to the recursive function $f\ R\ \mathit{arms}$. For any constructor c appearing in position n of ℓ, and any appropriate arguments r and v to the "real" version of c, we have (informally) that $F(c\ r\ v)$ is equal to the expected expansion based on the proper arm from *arms*, which must be described in terms of recursive calls to F with the elements of r.

Now we have all we need to provide each case of an inductive proof. However, we lack the final ingredient to tie the cases together. For the sake of a concrete example, consider the silly modification of size to size2 by changing every occurrence of 1 to 2, so that instances of size2 are equivalent to doubling the results of size instances.

Like for size, for a piece of ind evidence ι, we define size2 by applying recOf ι with $R =$ nat. Remembering the "propositions as types" and "proofs as programs" principle, we try to construct an inductive proof that, for all x, size2 $\iota\ x$ is even. Inductive proofs are isomorphic to recursive functions, so we expect to be able to use recOf ι to build the proof.

Which value of R will let us accomplish this? Informally, we want something like $R =$ isEven (size2 $\iota\ x$). Of course, this is invalid, since it contains a free variable x. What we *really* want is $R = \lambda x : T.$ isEven (size2 $\iota\ x$). That is, R is now a *predicate* over T's, not simply a proposition; alternatively, R is an indexed family of dependent types. We must modify our definition of rec to support such families. The original formulation with non-dependent types will be derivable from this richer version.

We start by redefining conIH, the type of a single arm in a recursive definition. We add a new parameter, b, the function for applying the constructor in question. R's type is changed as described above; the type of the argument that includes recursive call results is changed to use a Σ type to reflect dependence on actual values; and the final result type uses b to build the actual value that we are talking about, so that we can apply R to it.

$$\begin{array}{lcl} \mathsf{conIH} & = & \lambda T : \mathsf{Type}.\ \lambda c : \mathsf{con}.\ \lambda b : \mathsf{interpCon}\ T\ c. \\ & & \quad \forall R : T \to \mathsf{Type}. \\ & & \quad \forall r : \mathsf{repeat}\ (\Sigma x : T.\ R\ x)\ (\pi_1\ c). \\ & & \quad \forall v : \pi_2\ c.\ R\ (b\ (\mathsf{mapRepeat} \\ & & \qquad (\lambda y : (\Sigma x : T.\ R\ x).\ \pi_1\ y)\ r)\ v) \end{array}$$

Now a simple modification to rec finishes the job. We use a type family tupleFF which is like tupleF, but is parameterized on an existing tupleF whose elements may influence the types of the corresponding elements of the new tuple.

$$\begin{array}{lcl} \mathsf{rec} & = & \lambda T : \mathsf{Type}.\ \lambda \ell : \mathsf{list\ con}.\ \lambda b : \mathsf{tupleF}\ (\mathsf{interpCon}\ T)\ \ell. \\ & & \quad \forall R : T \to \mathsf{Type}.\ \mathsf{tupleFF} \\ & & \qquad (\lambda c.\ \lambda b : \mathsf{interpCon}\ T\ c.\ \mathsf{conIH}\ T\ c\ b\ R)\ b \\ & & \qquad \to (\forall x : T.\ R\ x) \end{array}$$

We also change eqn's type to take into account rec's new type.

Though the details are tedious, it is now possible to push through a generic proof of $\forall x, \mathsf{isEven}\ (\mathsf{size2}\ \iota\ x)$. We build the proof using recOf ι with $R = \lambda x : T.\ \mathsf{isEven}\ (\mathsf{size2}\ \iota\ x)$. In each arm of the proof, corresponding to some constructor c, we start out by using eqn to rewrite the goal, revealing it (after some simplification with $\equiv$) to be a fold over the number of recursive arguments that c has. An inner induction on that number of arguments allows us to establish that the sum is even, relying crucially on the inductive hypotheses coming to us by way of the parameter r in conIH.

Though the subject of this paper is formal theorem proving, the above barely deserves to be called a "proof sketch." However, we will stick to that level of detail for the next few portions of this paper. Section 6 discusses the real engineering issues of constructing this kind of proof in Coq.

4.4 A Word on Trusted Code Bases

Writing programs that build programs is a tricky business. It is very easy to introduce bugs that remain hidden even after very careful testing. By implementing generic programs in Coq with rich dependent types, we get the type checker to validate their basic sanity properties. Modulo bugs in Coq, there is no chance for an unexpected input to cause a generic program to produce an ill-typed output. This is already very useful on the examples we have considered, and it will get even more useful when we move to generic programs with fancier dependent types that capture more domain-specific invariants.

Yet the reader may have noticed an obstacle in the way of achieving completely formal generic programming with our techniques. We presented no formal procedure for constructing ind evidence packages. Indeed, as for reflective proofs in general, these packages are constructed in an ad-hoc way; in our case, by OCaml code in a plug-in for Coq. This mirrors the situation for SYB and similar generic programming techniques, where some basic combinators must be constructed for each type outside of any nice static type system. We are slightly better off in that regard, because, once a piece of evidence is created, the code that uses it is free of the unsafe type casts that appear in the corresponding parts of SYB.

It is worth mentioning another consequence of this split for formal theorem-proving. It often happens that one wants a certified program whose specification can be given quite simply without any nasty boilerplate, but whose implementation calls out for judicious use of generic programming. Our certified compiler [Chl07] is one such example. Though the ad-hoc generation of evidence has no proofs associated with it, *it is still not trusted* in this kind of scenario. Any evidence that pleases the type checker leads to correct code generation, and the type checker will catch faulty evidence before it can precipitate any invalid proof.

5. A Lightning-Speed Tour of Language Metatheory using Dependent Types

The basic approach that we present in this paper has applications to a wide variety of interesting domains that use types to enforce detailed specifications. As we argued in the introduction, formalization of programming languages and transformations over them leads to particularly many opportunities to put generic programming and proving to good use. Considering the range of program-manipulating tools, we see a vast array of tree-structured data types that can be fit into just a few patterns.

One of these patterns is based on the variable usage conventions found in the descendants of lambda calculus. There are a wide variety of generic operations associated with such programs, including variable substitution and calculating free variable sets. In many cases, a *nameless* representation of variable usage, otherwise known as the de Bruijn convention [dB72], makes development easier. By representing variable uses as natural numbers counting how many binders outward in lexical scoping order to search to find their matching binders, we do away with the problem of alpha equivalence. That is, our default notion of equality matches up with our notion of syntactic equivalence of programs. However, many bookkeeping tasks become trickier, as transplanting a de Bruijn term into a new context often requires running over it adjusting numeric indices. The many clean-up operations of this kind form another class of useful applications of generic programming.

In the remainder of this paper, we will focus on a particular framework for doing formal language metatheory in this vein. The following subsections give an overview of our system Lambda Tamer [Chl07] and its conventions by example, to provide a reference point for the presentation to follow. Its distinguishing characteristics are use of *dependently-typed abstract syntax* (to ensure that only well-typed terms are representable) and *denotational semantics*. It draws on ideas developed in past work on syntax representation in type theory, including that of Altenkirch and Reus [AR99].

5.1 Syntax and Typing Rules

We use the example of simply-typed lambda calculus with unit as the base type. The starting point for a Lambda Tamer formalization is a definition of the type system. In this case, we want a standard algebraic datatype describing the nullary type constructor Unit and the binary Arrow.

```
Inductive ty : Type :=
  | Unit : ty
  | Arrow : ty → ty → ty
```

To move on to the syntax of terms, we first need to fix a representation of variables. We will be using de Bruijn terms, so we could simply use natural numbers as variables. However, we want to use dependent typing to guarantee lack of dangling variable references. In fact, we will go even further and have variables track, in their meta-level types, the object-level types of the terms they represent. Here is a generic definition of variables, taken from the

Lambda Tamer Coq library:

$$\begin{array}{l}\text{Inductive var } (ty : \text{Type}) : \text{list } ty \rightarrow ty \rightarrow \text{Type} := \\ \quad | \text{ First} : \forall \tau : ty.\ \forall \Gamma : \text{list } ty.\ \text{var } (\tau :: \Gamma)\ \tau \\ \quad | \text{ Next} : \forall \tau, \tau' : ty.\ \forall \Gamma : \text{list } ty.\ \text{var } \Gamma\ \tau \rightarrow \text{var } (\tau' :: \Gamma)\ \tau\end{array}$$

The family var is polymorphic in the type language ty. Additionally, it takes two indices corresponding to the first and last positions of a three-place variable-typing judgment $\Gamma \vdash x : \tau$. We can think of vars as derivations of such judgments, built up from two rules. Alternatively, we can think of vars as the natural numbers we would have used anyway, just annotated with extra typing information to combine selection of a type from a list with the construction of indices. With either interpretation, we are able to use variables easily without maintaining ad-hoc bookkeeping information about them.

Now we can give a natural definition of the syntax of terms. As with variables, we use (meta-level) dependent types to combine typing derivations with abstract syntax. The typing rules so embodied are the standard rules for simply-typed lambda calculus.

$$\begin{array}{l}\text{Inductive term} : \text{list ty} \rightarrow \text{ty} \rightarrow \text{Type} := \\ \quad | \text{ Var} : \forall \Gamma : \text{list ty}.\ \forall \tau : \text{ty}.\ \text{var } \Gamma\ \tau \rightarrow \text{term } \Gamma\ \tau \\ \quad | \text{ UnitIntro} : \forall \Gamma : \text{list ty}.\ \text{term } \Gamma \text{ Unit} \\ \quad | \text{ App} : \forall \Gamma : \text{list ty}.\ \forall \tau_1, \tau_2 : \text{ty}.\ \text{term } \Gamma\ (\text{Arrow } \tau_1\ \tau_2) \\ \qquad \rightarrow \text{term } \Gamma\ \tau_1 \rightarrow \text{term } \Gamma\ \tau_2 \\ \quad | \text{ Lam} : \forall \Gamma : \text{list ty}.\ \forall \tau_1, \tau_2 : \text{ty}.\ \text{term } (\tau_1 :: \Gamma)\ \tau_2 \\ \qquad \rightarrow \text{term } \Gamma\ (\text{Arrow } \tau_1\ \tau_2)\end{array}$$

Part of the attraction of this representation technique is that, when we write compilers and other code transformations, the Coq type system ensures that our transformations produce well-typed terms when passed well-typed terms. For a simple example, consider this function, which implements η-expansion.

$$\begin{array}{lcl}\text{eta} & = & \lambda \Gamma, \tau_1, \tau_2.\ \lambda e : \text{term } \Gamma\ (\text{Arrow } \tau_1\ \tau_2). \\ & & \quad \text{Lam } (\text{App } e\ (\text{Var First}))\end{array}$$

There is a problem here. The definition of eta does not type-check! Where the Coq variable e is used, a term of type term $(\tau_1 :: \Gamma)$ (Arrow τ_1 τ_2) is expected, while e has type term Γ (Arrow τ_1 τ_2). We know informally that it is of course acceptable to bring an extra, unused variable (in this case, the variable bound by the new lambda) into scope, and it is exactly this fact that would convince the type-checker to accept this definition.

What we need is an auxiliary function typed like this:

$$\text{lift} \quad : \quad \forall \Gamma : \text{list ty}.\ \forall \tau, \tau' : \text{ty}.\ \text{term } \Gamma\ \tau \rightarrow \text{term } (\tau' :: \Gamma)\ \tau$$

This is the lifting operation of de Bruijn indices. As generally happens with this kind of dependently-typed representation, lift can also be thought of as a lemma about typing derivations. In this case, it is a standard *weakening* lemma: "If $\Gamma \vdash e : \tau$, then, for x not free in e, $\Gamma, x : \tau' \vdash e : \tau$."

Now we can revise the definition of eta, such that it type-checks without incident.

$$\begin{array}{lcl}\text{eta} & = & \lambda \Gamma, \tau_1, \tau_2.\ \lambda e : \text{term } \Gamma\ (\text{Arrow } \tau_1\ \tau_2). \\ & & \quad \text{Lam } (\text{App } (\text{lift } \tau_1\ e)\ (\text{Var First}))\end{array}$$

5.2 Dynamic Semantics

When an object language is purely functional and permits no recursion beyond primitive recursion, it is often most convenient to give its dynamic semantics by *compilation into CIC*. CIC is rich enough that such compilations can be written in the same calculus, taking the dependently-typed abstract syntax trees of the last subsection as inputs. This strategy is "denotational" rather than "operational," and it brings with it some serious benefits. As demonstrated in the earlier discussion of proof by reflection, the embedding of term simplification in the Coq typing judgment often allows us to get the type checker/proof checker to "do our work for us." By giving programs meanings by compilation into CIC, we essentially end up with a logic that has "axioms" about program behavior built into it and applied automatically for us by the Coq theorem proving tools.

The first step for our example is to define a simple recursive function mapping object language types into meta language types:

$$\begin{array}{lcl}\text{tyDenote} & = & \text{fix } d\ (\tau : \text{ty}) : \text{Type} := \text{match } \tau \text{ with} \\ & & \quad | \text{ Unit} \Rightarrow \text{unit} \\ & & \quad | \text{ Arrow } \tau_1\ \tau_2 \Rightarrow \text{d } \tau_1 \rightarrow \text{d } \tau_2\end{array}$$

Now we are almost ready to give a denotational semantics to terms. We will map each term to a function from a *substitution* for its free variables to its denotation as a "native" CIC term. To represent substitutions, we introduce subst as an illustrative synonym for tupleF. Now, using a library function varDenote that interprets variables following the same scheme, we have:

$$\begin{array}{lcl}\text{termDenote} & = & \text{fix } d\ (\Gamma : \text{list ty})\ (\tau : \text{ty})\ (e : \text{term } \Gamma\ \tau) \\ & & \qquad : \text{subst tyDenote } \Gamma \rightarrow \text{tyDenote } \tau := \\ & & \quad | \text{ Var } v \Rightarrow \lambda \sigma.\ \text{varDenote } v\ \sigma \\ & & \quad | \text{ UnitIntro} \Rightarrow \lambda\_.\ \text{tt} \\ & & \quad | \text{ App } e_1\ e_2 \Rightarrow \lambda \sigma.\ (d\ e_1\ \sigma)\ (d\ e_2\ \sigma) \\ & & \quad | \text{ Lam } e' \Rightarrow \lambda \sigma.\ (\lambda x.\ d\ e'\ (\text{SCons } x\ \sigma))\end{array}$$

We can use termDenote to state the correctness property of eta-expansion:

$$\begin{array}{l}\forall \Gamma, \tau_1, \tau_2.\ \forall e : \text{term } \Gamma\ (\text{Arrow } \tau_1\ \tau_2).\ \forall \sigma : \text{subst tyDenote } \Gamma. \\ \quad \text{termDenote } (\text{eta } e)\ \sigma = \text{termDenote } e\ \sigma\end{array}$$

The dynamic semantics of CIC matches closely enough our intended dynamic semantics of the object language that we can compare term denotations with equality, rather than with some more customized congruence. However, what seems certain to be a trivial proof has a wrinkle in it. We need to reason about the behavior of lift to prove this theorem, which requires an induction over the structure of terms and some painful reasoning about dependent types. What we really want is a lemma like this:

$$\begin{array}{lcl}\text{liftSound} & : & \forall \Gamma, \tau, \tau'.\ \forall e : \text{term } \Gamma\ \tau. \\ & & \quad \forall \sigma : \text{subst tyDenote } \Gamma.\ \forall v : \text{tyDenote } \tau'. \\ & & \quad \text{termDenote } (\text{lift } \tau'\ e)\ (\text{SCons } v\ \sigma) \\ & & \qquad = \text{termDenote } e\ \sigma\end{array}$$

With a tool to prove this lemma for us, the correctness of eta is established easily, with no explicit induction.

6. A Taste of Manual Implementation

A natural first reaction to discovering the utility of lift and liftSound is to implement them manually, specialized to this language. How hard could it be, after all? In our experience, while implementing a helper function like lift without rich types in a language like ML or Haskell can be painful, implementing it *and* its correctness lemma in Coq can be tantamount to an existential crisis. It certainly does not help us get to the interesting parts of a development very quickly, and such manual derivations will in fact tend to monopolize a total body of code.

To portray the difficulties therein, we will present in this section snippets of manual derivation of lift and liftSound. As a side benefit, our examples will draw in aspects of programming with general inductive types and first-class equality proofs that have rarely been included in formal publications.

6.1 Implementing lift

First, it quickly becomes apparent that we will need to define an auxiliary function lift′. When lifting a term, recursing inside a lambda binder adds a new type to the context, and so our workhorse function must support some additional prefix of types before the position in a context where we will insert a new type. More formally, we want the following, where ⊕ is list concatenation and both ⊕ and :: are right associative:

$$\begin{array}{lll} \mathsf{lift}' & : & \forall \Gamma_1, \Gamma_2 : \mathsf{list\ ty}.\ \forall \tau, \tau' : \mathsf{ty}.\ \mathsf{term}\ (\Gamma_1 \oplus \Gamma_2)\ \tau \\ & & \quad \to \mathsf{term}\ (\Gamma_1 \oplus \tau' :: \Gamma_2)\ \tau \end{array}$$

Armed with our knowledge of fix expressions, we make a first attempt at coding lift′. We give ourselves a break by assuming we have an appropriately-typed library function liftVar′ that makes the corresponding adjustment to variables. We also take good advantage of opportunities for Coq to infer some omitted arguments from the values of others.

$$\begin{array}{lll} \mathsf{lift}' & = & \mathsf{fix}\ f\ \Gamma_1\ \Gamma_2\ \tau\ (e : \mathsf{term}\ (\Gamma_1 \oplus \Gamma_2)\ \tau) \\ & & \quad : \forall \tau'.\ \mathsf{term}\ (\Gamma_1 \oplus \tau' :: \Gamma_2)\ \tau := \\ & & \mathsf{match}\ e\ \mathsf{with} \\ & & \quad \mid \mathsf{Var}\ v \Rightarrow \lambda \tau'.\ \mathsf{Var}\ (\mathsf{liftVar}'\ v\ \tau') \\ & & \quad \mid \mathsf{UnitIntro} \Rightarrow \lambda \tau'.\ \mathsf{UnitIntro} \\ & & \quad \mid \mathsf{App}\ e_1\ e_2 \Rightarrow \lambda \tau'.\ \mathsf{App}\ (\mathsf{lift}'\ e_1\ \tau')\ (\mathsf{lift}'\ e_2\ \tau') \\ & & \quad \mid \mathsf{Lam}\ e' \Rightarrow \lambda \tau'.\ \mathsf{Lam}\ (\mathsf{lift}'\ e'\ \tau') \end{array}$$

Here we introduce one aspect of Coq programming that we have been eliding so far. To enable decidable type-checking, Coq requires certain annotations on match expressions that deal with dependent types. In this case, we should rewrite the first line of the function body as:

$$\begin{array}{l} \mathsf{match}\ e\ \mathsf{in}\ (\mathsf{term}\ (T_1 \oplus T_2)\ T_3) \\ \quad \mathsf{return}\ (\forall \tau'.\ \mathsf{term}\ (T_1 \oplus \tau' :: T_2)\ T_3)\ \mathsf{with} \end{array}$$

An in clause describes the type of the expression being analyzed, providing a place to bind variables like the T_i above. A return clause describes the type of the match expression in terms of the variables bound by the in clause. In each branch arm, the T_i variables are replaced by the forms implied by the constructor being matched. For instance, in the UnitIntro branch, we have the instantiation $T_3 = \mathsf{Unit}$. This general regime should be familiar to readers accustomed to working with GADTs. In Coq, the programmer writes explicitly some of what standard GADT inference algorithms infer. In return, the Coq programmer is able to type-check more complex programs by providing explicit coercions.

Unfortunately, there is still a problem with our amended definition! Coq only allows in clauses that are inductive type names applied to lists of variables. Our code has $T_1 \oplus T_2$ where it must have a single variable. Why does Coq place this restriction? The answer is that, without the restriction, the type checker would have to solve higher-order unification problems, making type checking undecidable [Hue73]. In other words, there can exist no algorithm that matches up variables between arbitrary in clauses and arbitrary parameter choices in the range types of constructors. This does not mean that there can be no useful restrictions that lead to decidablity, or that we cannot optimistically apply heuristics until some timeout is reached, but the designers of Coq chose to follow a simpler path.

A standard trick gets us out of this mess. We convert to *equality-passing style*, taking advantage of CIC's expressiveness to work with *first-class equality proofs* and their associated coercions. In the Coq standard library, equality is an inductive type like any other.

$$\begin{array}{l} \mathsf{Inductive\ eq}\ (T : \mathsf{Type})\ (x : T) : T \to \mathsf{Type} := \\ \quad \mid \mathsf{refl\_equal} : \mathsf{eq}\ T\ x\ x \end{array}$$

$$\begin{array}{lll} \mathsf{lift}'' & = & \mathsf{fix}\ f\ \Gamma\ \tau\ (e : \mathsf{term}\ \Gamma\ \tau) \\ & & \quad : \forall \Gamma_1, \Gamma_2.\ \Gamma = \Gamma_1 \oplus \Gamma_2 \\ & & \qquad \to \forall \tau'.\ \mathsf{term}\ (\Gamma_1 \oplus \tau' :: \Gamma_2)\ \tau := \\ & & \mathsf{match}\ e\ \mathsf{in}\ (\mathsf{term}\ T_1\ T_2) \\ & & \quad \mathsf{return}\ (\forall \Gamma_1, \Gamma_2.\ T_1 = \Gamma_1 \oplus \Gamma_2 \\ & & \qquad \to \forall \tau'.\ \mathsf{term}\ (\Gamma_1 \oplus \tau' :: \Gamma_2)\ T_2)\ \mathsf{with} \\ & & \quad \mid \mathsf{Var}\ v \Rightarrow \lambda \Gamma_1, \Gamma_2, pf, \tau'.\ \mathsf{Var}\ (\mathsf{liftVar}''pf\ v\ \tau') \\ & & \quad \mid \mathsf{UnitIntro} \Rightarrow \lambda \Gamma_1, \Gamma_2, pf, \tau'.\ \mathsf{UnitIntro} \\ & & \quad \mid \mathsf{App}\ e_1\ e_2 \Rightarrow \lambda \Gamma_1, \Gamma_2, pf, \tau'. \\ & & \qquad \mathsf{App}\ (\mathsf{lift}''\ e_1\ pf\ \tau')\ (\mathsf{lift}''\ e_2\ pf\ \tau') \\ & & \quad \mid \mathsf{Lam}\ e' \Rightarrow \lambda \Gamma_1, \Gamma_2, pf, \tau'. \\ & & \qquad \mathsf{Lam}\ (\mathsf{lift}''\ e'\ (\mathsf{liftPf}\ pf\ \_)\ \tau') \end{array}$$

Figure 5. Correct definition of lift″

We define equality (usually abbreviated with the binary = operator) as "the least reflexive relation," bootstrapping off of the definitional equality ≡ built into CIC. Whereas ≡ is applied implicitly, we can name equality "facts" that may or may not hold, by reifying ≡ into this "propositional equality." Based on CIC's rules governing inductive types, the following operation is derivable:

$$\begin{array}{lll} \mathsf{cast} & : & \forall T : \mathsf{Type}.\ \forall f : (T \to \mathsf{Type}).\ \forall x, y : T. \\ & & \quad x = y \to f\ x \to f\ y \end{array}$$

It says that, for any "type with a hole in it" (represented by a function f), if we present a proof that $x = y$, and if x and y are of the right type to fill the hole, then we may cast type $f\ x$ to type $f\ y$. This is a computational interpretation of what it means for equality to be a congruence.

We can use equality to write the main helper function lift″ as shown in Figure 5. We give the function a new (but equivalent) type to the type we gave lift′: it takes as input a term e that can be associated with *any* context Γ. *However*, as additional arguments, we are required to provide contexts Γ_1 and Γ_2 and a proof that $\Gamma = \Gamma_1 \oplus \Gamma_2$. Though we do not show it here, cast is used in the definition of liftVar″. We leave underscores as "jokers" in some places where their values are inferable.

To build the proof for the recursive call in the Lam case, we rely on some implementation of this type, corresponding to an "obvious" theorem about lists and equality:

$$\mathsf{liftPf} \quad : \quad \forall \Gamma_1, \Gamma_2.\ \Gamma_1 = \Gamma_2 \to \forall \tau.\ \tau :: \Gamma_1 = \tau :: \Gamma_2$$

Now lift′ is definable as

$$\lambda \Gamma_1, \Gamma_2, \tau, \tau', e.\ \mathsf{lift}''\ e\ (\mathsf{refl\_equal}\ (\Gamma_1 \oplus \Gamma_2))\ \tau'$$

6.2 Proving liftSound

The heart of the proof of liftSound is, probably unsurprisingly, in a lemma about lift″. We overload :: and ⊕ to denote one- and multi-element concatenation of substitutions.

$$\begin{array}{lll} \mathsf{liftSound}'' & : & \forall \Gamma, \tau.\ \forall e : \mathsf{term}\ \Gamma\ \tau.\ \forall \Gamma_1, \Gamma_2. \\ & & \quad \forall pf : (\Gamma = \Gamma_1 \oplus \Gamma_2).\ \forall \tau', \sigma_1, v, \sigma_2. \\ & & \qquad \mathsf{termDenote}\ (\mathsf{lift}''\ e\ pf\ \tau')\ (\sigma_1 \oplus v :: \sigma_2) \\ & & \qquad = \mathsf{termDenote}\ (\mathsf{cast}\ (\lambda X.\ \mathsf{term}\ X\ \_) \\ & & \qquad\quad pf\ e)\ (\sigma_1 \oplus \sigma_2) \end{array}$$

The most interesting part is the use of cast. Without it, we would have an argument type incompatibility for the second use of termDenote, since e is in context Γ and $\sigma_1 \oplus \sigma_2$ is in context

$$\begin{array}{lcl} \ldots & & \\ pf & : & \Gamma = \Gamma_1 \oplus \Gamma_2 \\ \ldots & & \end{array}$$

$$\begin{array}{l} \mathsf{termDenote}\ (\mathsf{lift}''\ \mathsf{UnitIntro}\ pf\ \tau')\ (\sigma_1 \oplus v :: \sigma_2) \\ \quad = \mathsf{termDenote}\ (\mathsf{cast}\ (\lambda X.\ \mathsf{term}\ X\ \_) \\ \qquad pf\ \mathsf{UnitIntro})\ (\sigma_1 \oplus \sigma_2) \end{array}$$

Figure 6. Partial initial proof state for UnitIntro case of liftSound″

$\Gamma_1 \oplus \Gamma_2$. By presenting an explicit equality proof between those two types, we bridge the gap.

The skeleton of the proof is an induction on the structure of e. Following the "proofs as programs" principle, this can be written as a fix expression over e. In this case, it is more convenient to use Coq's tactic-based theorem proving mode to construct the proof term interactively, with help from decision procedures. We will show only the simplest case of the proof, that for UnitIntro, as it already demonstrates how slippery this kind of proof is.

The initial proof state appears in Figure 6. We have a set of hypotheses/free variables appearing above the horizontal line with their types. Below the line appears the proposition we are trying to prove (the "conclusion"); alternatively, what we see there is a type, and we are trying to code a program with that type. We can simplify the lefthand side of the conclusion to tt using only the definitional equality $\equiv$. The tricky part is simplifying the righthand side to the same value.

We might think we ought to be able to replace the cast term with UnitIntro. However, here UnitIntro has an elided parameter specifying which typing context it lives in, Γ. So, erasing the cast, we are left with the argument type incompatibility that we added the cast to avoid. A somewhat counterintuitive trick helps us make the next step. We use pf to replace Γ with $\Gamma_1 \oplus \Gamma_2$ everywhere that Γ appears, *even in pf's own type*! Now pf has type $\Gamma_1 \oplus \Gamma_2 = \Gamma_1 \oplus \Gamma_2$, and we have made some good progress. Now we would be left with a *well-typed* righthand side if we made the simplification we want to make; what remains is to figure out how to justify the rewrite.

Surprisingly enough, working from just the simple inductive definition of eq that we gave earlier, we cannot make further progress. CIC just is not a strong enough type theory to model the computational behavior of casting without the addition of further rules. To fix this, it is common to add some *axiom* characterizing this behavior. Axioms are propositions asserted without proof. They are convenient ways to make global changes to the logic one is working in, though it is always critical to check that your set of axioms introduces no inconsistency. The axiom that we chose to depend on [Str93] is included in the Coq standard library:

$$\begin{array}{lcl} \mathsf{castEq} & : & \forall T : \mathsf{Type}.\ \forall f : (T \rightarrow \mathsf{Type}).\ \forall x : T. \\ & & \forall pf : x = x.\ \forall v : f\ x.\ \mathsf{cast}\ f\ pf\ v = v \end{array}$$

One rewrite with castEq, followed by simplification using $\equiv$ alone, reduces the conclusion to tt = tt, which is proved directly by refl_equal.

6.3 The Prognosis

It is possible, with enough patience, to craft these programs and proofs manually, but it remains an intellectually challenging problem in each iteration. Researchers who do not specialize in type theory but want to formalize their programming languages would be justified in recoiling at the thought of needing to learn the type theory arcana we have employed.

We would rather have all of this done automatically. The more of code and proof generation we can have validated statically, the better. Though this will require even more wizardry in the implementation of the generic programming system, it will be worth the effort, because users will be able to employ the system without understanding those internals, while retaining the benefits of their rigorous verification.

7. The Lambda Tamer AutoSyntax System

The AutoSyntax piece of the Lambda Tamer system uses the techniques we have presented to provide generic implementations of functions like lift and lemmas like liftSound. To do so, it uses reflected representations of inductive definitions of term languages, much as we did for simply-typed datatypes in Section 4. However, our situation is now more complicated, as inductive definitions like term's from the last section use universal quantification and manipulate typing contexts.

Luckily, a very regular structure suffices for the types of AST constructors. Each type begins with a $\forall$ quantification over a typing context Γ. We then have some quantifications over types and other data (such as an integer as an argument to an integer constant constructor). In the scope of these quantifiers are some variable and term arguments. Each variable has context Γ, and each has a type expressed as a function of the quantified variables. Recursive term instances are a bit more complicated. Each has a type expressed in the same way, but we also need to allow for new types to be "pushed onto the front of" Γ. Thus, we allow the context argument of a subterm to be any number of types consed onto the beginning of Γ. Finally, the result type of any constructor is the term type itself at context Γ and some type (written, like before, as a function of the quantified variables).

With this pattern in mind, we can redo the definition of con from Section 4:

$$\begin{array}{lcl} \mathsf{con} & = & \lambda ty : \mathsf{Type}.\ \Sigma T : \mathsf{Type}.\ \mathsf{list}\ (T \rightarrow ty) \\ & & \times\ \mathsf{list}\ (T \rightarrow \mathsf{list}\ ty \times ty) \\ & & \times\ (T \rightarrow ty) \end{array}$$

con is parameterized by the type ty of object language types. Each con contains a type T that combines the domains of all the "real" constructor's quantified variables, using tupling if necessary; a list of the types of variables; a list of the types of subterms; and the result type. For later convenience, we define projection functions for the four components of cons: quantsOf, varsOf, termsOf, and resultOf.

A few examples should serve to explain the pattern. A good con ty for the constructor UnitIntro of our running example is:

$$\langle \mathsf{unit}, (\mathsf{nil}, \mathsf{nil}, \lambda\_.\ \mathsf{Unit}) \rangle$$

The case of Var illustrates variable typing:

$$\langle \mathsf{ty}, ([\lambda\tau.\ \tau], \mathsf{nil}, \lambda\tau.\ \tau) \rangle$$

Finally, the case of Lam illustrates binder typing:

$$\langle \mathsf{ty} \times \mathsf{ty}, (\mathsf{nil}, [\lambda t.\ ([\pi_1\ t], \pi_2\ t)], \lambda t.\ \mathsf{Arrow}\ (\pi_1\ t)\ (\pi_2\ t)) \rangle$$

From this starting point, we can redefine interpCon, conIH, rec, eqn, and ind in the (more or less) obvious ways. Since the new definitions are messier but unsurprising, we will not include them here. (The interested reader can, of course, obtain them in our source distribution.) The AutoSyntax plug-in for Coq generates all of the corresponding pieces automatically by inspecting type definitions.

7.1 Restricting Denotations

AutoSyntax also contains generic proofs of lemmas like $\mathsf{liftSound}$. Looking back at that theorem's statement, we see that it depends in no way on the details of the language under analysis. Its statement follows a pattern that recurs in our setting, where we give programs "denotational semantics" via definitional compilers. We want to know that a particular "commutative diagram" holds, in a sense specific to the generic function in question. We want to know that we get the same result by "compiling" expression e directly with substitution σ as we get by applying the generic function to e, applying some compensating transformation to σ, and *then* compiling.

How can we write a proof of this theorem that works for *any* definitional compiler? Actually, it is clear that we cannot make it work for just *any* dynamic semantics, since it is easy to come up with perverse semantics that perform intensional analysis on terms. For instance, we could write a denotation function that checks if its input equals a particular term and then returns some arbitrary result; otherwise, it uses a sane recursive definition. The $\mathsf{liftSound}$ specification might not hold on the term that's been singled out. Now the question is what conditions we can impose on denotations to facilitate use of a generic proof technique without compromising flexibility in language definition.

We find our criterion in the standard sanity check of denotational semantics, *compositionality*. Roughly speaking, we want the meaning of a term to be a function of nothing but the constructor used to build it, the values given to its quantified variables, and *the denotations of* its variables and subterms. In particular, it should not be valid to examine the *syntax* of a subterm in a denotation function.

For our specific example language, this means that we want there to exist functions

$$\begin{array}{rcl} f_{\mathsf{Var}} & : & \forall \tau.\ \mathsf{tyDenote}\ \tau \to \mathsf{tyDenote}\ \tau \\ f_{\mathsf{UnitIntro}} & : & \mathsf{tyDenote}\ \mathsf{Unit} \\ f_{\mathsf{App}} & : & \forall \tau_1, \tau_2.\ \mathsf{tyDenote}\ (\mathsf{Arrow}\ \tau_1\ \tau_2) \\ & & \quad \to \mathsf{tyDenote}\ \tau_1 \to \mathsf{tyDenote}\ \tau_2 \\ f_{\mathsf{Lam}} & : & \forall \tau_1, \tau_2.\ (\mathsf{tyDenote}\ \tau_1 \to \mathsf{tyDenote}\ \tau_2) \\ & & \quad \to \mathsf{tyDenote}\ (\mathsf{Arrow}\ \tau_1\ \tau_2) \end{array}$$

Notice that no detail of the implementation of $\mathsf{tyDenote}$ comes into play; we hope that it is initially plausible that function signatures like these can be read off from the definition of a term type. It is just a "coincidence" of the closeness of our object language to CIC that f_{Var}, f_{App}, and f_{Lam} ending up having the types of particular identity functions.

A denotational semantics for our object language is *compositional* when there exist functions of these types that satisfy the following equations. We overload juxtaposition as notation for looking up a variable in a substitution.

$$\begin{array}{rcl} \mathsf{termDenote}\ (\mathsf{Var}\ v)\ \sigma & = & f_{\mathsf{Var}}\ (\sigma v) \\ \mathsf{termDenote}\ \mathsf{UnitIntro}\ \sigma & = & f_{\mathsf{UnitIntro}} \\ \mathsf{termDenote}\ (\mathsf{App}\ e_1\ e_2)\ \sigma & = & f_{\mathsf{App}}\ (\mathsf{termDenote}\ e_1\ \sigma) \\ & & \quad (\mathsf{termDenote}\ e_2\ \sigma) \\ \mathsf{termDenote}\ (\mathsf{Lam}\ e')\ \sigma & = & f_{\mathsf{Lam}}\ (\lambda x.\ \mathsf{termDenote}\ e' \\ & & \quad (\mathsf{SCons}\ x\ \sigma)) \end{array}$$

7.2 Reflecting Denotations

We need to formalize the requirements of compositionality, so that we can write generic proofs about arbitrary compositional denotation functions. To start with, here is a function that calculates the proper type for one of the $f_{\_}$'s above, given a constructor and a type denotation function:

$$\begin{array}{rcl} \mathsf{schema} & = & \lambda ty : \mathsf{Type}.\ \lambda c : \mathsf{con}\ ty.\ \lambda d : (ty \to \mathsf{Type}). \\ & & \quad \forall q : \mathsf{quantsOf}\ c.\ \mathsf{tupleF}\ (\lambda v.\ d\ (v\ q))\ (\mathsf{varsOf}\ c) \\ & & \quad\quad \to \mathsf{tupleF}\ (\lambda t.\ \mathsf{tupleF}\ d\ (\pi_1\ (t\ q)) \\ & & \quad\quad\quad \to d\ (\pi_2\ (t\ q)))\ (\mathsf{termsOf}\ c) \\ & & \quad\quad \to d\ ((\mathsf{resultOf}\ c)\ q) \end{array}$$

Now we can define what it means for a denotation function to be compositional with respect to a single constructor. We appeal to the reader's intuition for the new types associated with functions like $\mathsf{interpCon}$.

$$\begin{array}{rcl} \mathsf{isComp} & = & \lambda ty : \mathsf{Type}.\ \lambda T : \mathsf{list}\ ty \to ty \to \mathsf{Type}. \\ & & \quad \lambda c : \mathsf{con}\ ty.\ \lambda b : \mathsf{interpCon}\ T\ c. \\ & & \quad \lambda dt : (ty \to \mathsf{Type}). \\ & & \quad \lambda dT : (\forall \Gamma, \tau.\ T\ \Gamma\ \tau \to \mathsf{subst}\ dt\ \Gamma \to dt\ \tau). \\ & & \quad \Sigma f : \mathsf{schema}\ ty\ c\ dt.\ \forall qs, vs, es, \sigma. \\ & & \quad\quad dT\ (b\ qs\ vs\ es)\ \sigma \\ & & \quad\quad = f\ qs\ (\mathsf{mapF}\ (\lambda v.\ \sigma v)\ vs) \\ & & \quad\quad\quad (\mathsf{mapF}\ (\lambda e.\ \lambda vs.\ dT\ (\pi_2\ e)\ (vs \oplus \sigma))\ es) \end{array}$$

With this definition, it is easy to define full compositionality by asserting that isComp must hold for each constructor of a type.

7.3 Sketch of a Generic $\mathsf{liftSound}''$ Proof

We wrap up our discussion of implementation techniques with a quick sketch of the proof of $\mathsf{liftSound}''$ for an AST type T reflected into evidence package ι, with respect to denotation function dT that has been shown compositional. We have not space to provide a detailed definition of the generic lift'' itself, but the basic idea should be straightforward: recurse through all term structure using $\mathsf{recOf}\ \iota$, applying $\mathsf{liftVar}''$ to all variables encountered.

1. The proof is by induction on the structure of the term e. Concretely, we apply the "induction principle" found in $\mathsf{recOf}\ \iota$. Perform the following steps for each inductive case, where each corresponds to some constructor c that was used to build e.
2. Just as in Section 6.2, use the proof $pf : \Gamma = \Gamma_1 \oplus \Gamma_2$ to rewrite Γ to $\Gamma_1 \oplus \Gamma_2$ everywhere, and then use castEq to remove the use of cast.
3. Simplify each side of the equality by rewriting with $\mathsf{eqnOf}\ \iota$. That is, we unfold the definition of lift'' one level, as we know the top level structure of its arguments.
4. Use the compositionality of dT to rewrite the $dT\ (c\ \cdots)\ \_$ on each side of the equation with f_c. Now each side is f_c applied to the same quantified variable values, followed by different lists of denotations of object-language variables and subterms.
5. Rewrite each lefthand side variable denotation to match its righthand side counterpart using $\mathsf{liftVar}''\_\mathsf{sound}$, a lemma from the Lambda Tamer library. (This is an inner induction on the number of variables.)
6. Rewrite each lefthand side subterm denotation to match its righthand side counterpart using the inductive hypothesis supplied by $\mathsf{recOf}\ \iota$. (This is an inner induction on the number of subterms.)
7. The conclusion is reduced to a trivial equality, which we prove by reflexivity.

7.4 Mutually-Inductive AST Types

Mutually-recursive type definitions are common in programming languages. It turns out that Coq is expressive enough to let us build support for mutually-inductive types on top of what we have already described. We illustrate this by the simple example of two mutually inductive term types `term1` and `term2` associated with a type language `ty`. First, we define a type-carrying enumeration of the different term sorts:

$$\begin{array}{l}\textsf{Inductive which : Type :=}\\ \quad\mid \textsf{Term1 : ty} \rightarrow \textsf{which}\\ \quad\mid \textsf{Term2 : ty} \rightarrow \textsf{which}\end{array}$$

The actual AutoSyntax implementation allows the choice of distinct type languages for typing contexts versus term types. We can take advantage of this by using which as our new term type language. Now we can define a new, combined term type:

$$\begin{array}{rcl}\textsf{term} & = & \lambda\Gamma : \textsf{list ty}.\ \lambda x : \textsf{which}.\ \textsf{match}\ x\ \textsf{with}\\ & & \mid \textsf{Term1}\ \tau \Rightarrow \textsf{term1}\ \Gamma\ \tau\\ & & \mid \textsf{Term2}\ \tau \Rightarrow \textsf{term2}\ \Gamma\ \tau\end{array}$$

It is a straightforward exercise in type-level computation to build an ind package that treats `term1` and `term2` like two parts of a single inductive type `term`. The AutoSyntax Coq plug-in does this work for arbitrary mutual definitions, and it translates the results back to the mutual setting.

7.5 Evaluation

Our most extensive case study for AutoSyntax to date is on our certified type-preserving compiler from simply-typed lambda calculus to an idealized assembly language [Chl07]. There, AutoSyntax is applied to the statically-typed target languages of CPS conversion, closure conversion, and conversion to closed first-order programs with explicit allocation. The portion of the source code responsible for these pieces weighs in at under 3000 lines. This includes the definitions of the syntax, static semantics, and dynamic semantics of all languages; the definitions of the transformations, which are dependently-typed in a way that brings type preservation theorems "for free"; and the semantics preservation proofs, which use Coq's tactic-based proof search facility. Among the generic functions provided by AutoSyntax are lifting, permutation, free variable set calculation, and strengthening (removing unused variables from a typing context, critical to efficient closure construction), along with derived versions of these operations, such as lifting multiple unused variables into a context at once. Following standard idioms for writing the generic pieces manually, the amount of code would have more than doubled.

We also put together a stand-alone case study of certified CPS conversion for simply-typed lambda calculus, found in `examples/CPS.v` in the source distribution. It contains about 250 lines of code. Notably, its proofs achieve almost the level of conciseness of their usual pencil-and-paper equivalents. Hypotheses in specific proof contexts are never mentioned by name. Rather, the lemmas generated by AutoSyntax are sufficient to guide relatively simple proof automation, augmented by a few "hints" about domain-specific approaches to getting out of tricky situations. We think this makes our implementation unique among all that we are aware of, with all competitors relying on considerable manual proving effort.

8. Comparison with Other Approaches

There have been past investigations into formalizing lambda calculi in type theory. For instance, McKinna and Pollack [MP99] worked with a formalization that avoids dependently-typed syntax. Their proofs are free of the obscurities from the manual proof sketched in Section 6.2. However, they must thread assumptions of term validity through proofs. This is the standard sort of trade-off that arises in choosing how expressive a formalization's types should be, and it is also illustrated in the many recent solutions to the POPLmark Challenge [ABF+05].

Another partial cure for the complexity of the underlying proofs comes from designing more convenient surface languages for type theory. The Agda and Epigram projects have made progress along these lines, providing more convenient notions of dependent pattern matching and equality. Often these extensions can be mapped into simpler type theories by mechanical translations [GMM06]. We consider it quite possible that much of the complexity of AutoSyntax is a result of shortcomings of Coq that these newer systems address. Generally speaking, it seems that Coq today supports complex theorem proving better than Agda and Epigram, while Agda and Epigram have better support for dependently-typed programming. While the need for management of complex proofs motivated us to choose Coq for this project, we can hope for a future system that combines the advantages of all three. However, our main idea of generic programming with a domain-specific universe for programming language syntax can also be applied usefully to any of these systems today.

We only demonstrate the technique on a very simple language in this article. One common misconception is that our use of denotational semantics prevents us from handling languages with general recursive functions. In fact, formalizations of several such languages were involved in our certified compiler project [Chl07]. CIC is expressive enough to let us represent denotations as possibly-infinite computation streams or trees, using co-inductive types [Gim95]. Many other common programming languages features that we do not treat here are also very natural to encode in CIC, such as universal and existential types, lists, and inductive types in general, so we do not see any fundamental difficulties in extending Lambda Tamer and AutoSyntax to handle them.

At the same time, our main point in this paper is that generic programming with universes is useful for automating proofs about programming languages. We believe that this general idea will translate well to any popular approach to semantics, including operational semantics on non-dependently-typed terms, with fairly straightforward modifications to the universe types. We have no evidence to present for this now, but we see no essential barriers, so the question remains for future work to decide. It seems unlikely that a technique like this can prove type soundness theorems, but most proofs of properties of variable-shuffling operations seem like obvious extensions.

AutoSyntax lacks many useful features that have appeared in generic programming systems for conventional programming languages, where the lack of formality both allows more design freedom and makes new functionality easier to implement. For instance, among the leaves of ASTs, AutoSyntax gives special treatment only to variables; there is no way to distinguish between, e.g., integers and booleans. More generally, many operations on our universe types, such as equality testing, are unimplementable in CIC, because we allow injection of arbitrary "real" terms as AST leaves. Many less formal approaches to reflection represent types in ways that admit these operations easily.

Haskell supports general recursive types, which are a broader class of types than CIC's inductive types. Not all recursive types have reasonable inductive principles, but many recursive types that are not inductive types still support useful generic programming, and several toolkits for Haskell facilitate this.

Several generic programming systems for Haskell allow generic programs defined by a combination of recursion on type structure and special cases for particular named types. AutoSyntax supports nothing like this.

These features are absent from AutoSyntax because we found no need for them in our case studies. It is unclear how one could hope to achieve this level of automation for proofs about language semantics in general, and proofs about variable reshufflings just do not seem to reap any benefit from these extra features. However, we see no fundamental obstacle to adding them.

9. Conclusion

We have presented a new approach to statically-validated generic programming and proving for classes of inductive types. The approach is compatible with small-trusted-code-base formal theorem proving, and our particular AutoSyntax system eases significantly the development of formal programming language metatheory and certified code transformations. As Section 7 demonstrates, the technique must be instantiated manually to suit different domain-specific uses of dependent typing. We believe it likely that a broad range of domains stand to benefit from the construction of such instantiations.

Acknowledgments

Thanks to the anonymous POPL'08 referees, Jesper Louis Andersen, Robin Green, Vesa Karvonen, Edward Kmett, Yitzhak Mandelbaum, Matthieu Sozeau, and Eelis van der Weegen for helpful feedback on earlier versions of this article.

References

[ABF+05] 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 *Proc. TPHOLs*, pages 50–65, 2005.

[ACPW08] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, and Stephanie Weirich. Engineering formal metatheory. In *Proc. POPL*, January 2008.

[AM03] Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In *Proc. Working Conference on Generic Programming*, 2003.

[AR99] Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic, 13th International Workshop, CSL '99*, pages 453–468, 1999.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[BDJ03] Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10(4):265–289, 2003.

[Bou97] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In *Proc. STACS*, pages 515–529, 1997.

[Chl07] Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *Proc. PLDI*, June 2007.

[dB72] Nicolas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formal manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[DS06] Peter Dybjer and Anton Setzer. Indexed induction-recursion. *Journal of Logic and Algebraic Programming*, 66(1):1–49, January 2006.

[Gim95] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In *Proc. TYPES*, pages 39–59. Springer-Verlag, 1995.

[GMM06] Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In *Algebra, Meaning and Computation*, pages 521–540. 2006.

[GWdSO07] Jeremy Gibbons, Meng Wang, and Bruno C. d. S. Oliveira. Generic and indexed programming. In Marco Morazan, editor, *Trends in Functional Programming*, 2007.

[Hin00] Ralf Hinze. A new approach to generic functional programming. In *Proc. POPL*, pages 119–132, 2000.

[HP00] Ralf Hinze and Simon Peyton Jones. Derivable type classes. In *Proc. Haskell workshop*, 2000.

[Hue73] G. Huet. The undecidability of unification in third order logic. *Information and Control*, 22:257–267, 1973.

[JJ97] Patrik Jansson and Johan Jeuring. PolyP – a polytypic programming language extension. In *Proc. POPL*, pages 470–482, 1997.

[LP03] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. In *Proc. TLDI*, January 2003.

[LP04] Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proc. ICFP*, September 2004.

[LV02] Ralf Lämmel and Joost Visser. Typed combinators for generic traversal. In *Proc. PADL*, pages 137–154, 2002.

[MAG07] Peter Morris, Thorsten Altenkirch, and Neil Ghani. Constructing strictly positive families. In *The Australasian Theory Symposium (CATS2007)*, January 2007.

[ML84] P. Martin-Löf. Intuitionistic type theory, 1984. Bibliopolis-Napoli.

[MP99] James McKinna and Robert Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3-4):373–409, 1999.

[PE88] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proc. PLDI*, pages 199–208, 1988.

[PM93] Christine Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In *Proc. TLCA*, pages 328–345. 1993.

[PR99] Holger Pfeifer and Harald Rueß. Polytypic proof construction. In *Proc. TPHOLs*, 1999.

[She04] Tim Sheard. Languages of the future. In *Proc. OOPSLA*, pages 116–119, 2004.

[Str93] T. Streicher. Semantical investigations into intensional type theory. Habilitationsschrift, LMU München, 1993.

[UT05] C. Urban and C. Tasson. Nominal Techniques in Isabelle/HOL. In *Proc. CADE*, pages 38–53, 2005.

[Wei06] Stephanie Weirich. RepLib: a library for derivable type classes. In *Proc. Haskell workshop*, pages 1–12, 2006.