Growing a Proof Assistant

William J. Bowman
wjb@williamjbowman.com
Northeastern University

Abstract

Theoreticians often use sophisticated notation to communicate and reason about key ideas in their theories and models. Notation is often domain-specific or even invented on-the-fly when creating a new theory or model. Proof assistants aid theoreticians by rigorously checking formal models, but have poor support for allowing users to *conveniently* define and use *sophisticated* notation. For example, in a proof assistant like Coq or Agda, users can easily define simple notation like $\Gamma \vdash$ `e : t`, but to use BNF notation the user must use a preprocessing tool external to the proof assistant, which is cumbersome.

To support convenient and sophisticated extension, we can use *language extension* as a fundamental part of the design of a proof assistant. By starting from language extension we can not only facilitate convenient and sophisticated user-defined extensions, but also get a single, compositional system for writing all extensions to the core proof language.

We describe how to design a language-extension system that supports safe, convenient, and sophisticated user-defined extensions, and how to design a proof assistant based on language extension. We evaluate this design by building a proof assistant that features a small dependent type theory as the core language and implementing the following extensions in small user-defined libraries: pattern matching for inductive types, dependently typed staged meta-programming, a tactic language, and BNF and inference-rule notation for inductive type definitions.

1. Introduction

Notation is important to convey ideas quickly while ignoring uninteresting details, but notation is not fixed. Each domain has its own notation used to hide the uninteresting details common in that domain. The notation commonly used in programming languages research differs from the notation commonly used in cryptography. Even in the same domain, individual results require new notation to suit the needs of each new model or proof. Every new programming languages result may use common domain-specific notation, like BNF grammars, but may also define new notation to convey new ideas.

When working with models on papers, we may conveniently create arbitrarily sophisticated notation. We may define simple syntactic sugar by saying "we write `let x = e1 in e2` to mean `(λ x : t. e2) e1`". Or we may define sophisticated extensions that require the reader to perform computation: "we omit the type annotation and instead write `λ x. e` when the type of `x` can be inferred".

Creating these extensions is easy when developing models on paper, but not when using proof assistants. Proof assistants provide increased confidence in formal models and proofs, but lack support for allowing users to *conveniently* define *sophisticated* extensions. This lack of support has two downsides. First, formal models need to be reproduced in another medium (such as LaTeX) to communicate them effectively, which duplicates effort and risks the two models falling out of sync. Second, it decreases confidence that the specification is correct since the specification must be manually encoded into the language of the proof assistant, rather than written in familiar notation.

Some proof assistants, like Agda, enable convenient user-defined extensions as long as the extension is not sophisticated. Agda's mixfix notation (Danielsson and Norell 2008) is convenient to use but only supports simple notation definitions, like writing a function named `_ ⊢ _ : _`, where `_` indicates the position of arguments. Other proof assistants, like Coq, support sophisticated extensions, but creating these extensions is inconvenient to the point that few users can do it. Writing a Coq plugin requires the developer to use a separate toolchain to compile against the Coq implementation, and requires the user to compile and link the plugin against their Coq installation. However, these plugins support sophisticated extensions like Mtac (Ziliani et al. 2013), a new tactic language for Coq. To support more convenient but less sophisticated extensions, Coq also provides other extensions systems like notations, Ltac, and extensible parsing, but with multiple systems comes an increased learning curve for users and the challenge of composing multiple extensions from difference extension systems.

We propose to design proof assistants by using *language extension*, in the style of Lisp and its descendants, as a fundamental feature. This not only supports convenient and sophisticated user-defined extension, but provides a single and compositional system for writing extensions. Informally, we can think of this design as follows: rather than start with a proof assistant and add user-defined extensions, we start with a core language plus a language-extension system from which we can "grow" a proof assistant. We explain the design in detail by:

- Describing a core language for expressing formal models and proofs (section 2). Our core language, called Curnel, is a dependently typed λ-calculus, and does not contain any features except those required for expressing *encodings* of formal models and proofs. The Curnel implementation is less than 700 lines of code.
- Describing the design and implementation of our language-extension system (section 3). We explain what it means for language-extension systems to enable safe, convenient, and sophisticated extensions, and how to build the "seed" of a proof assistant from a core language and a language-extension system.

We evaluate this design by implementing a proof assistant called Cur that supports safe, convenient, and sophisticated language extension as defined in section 3. To evaluate convenience, we rely partially on lines of code as a proxy, although it does not take into account automatic integration into the proof assistant or compositionality of the extension system. To evaluate the level of sophistication we support, we implement proof-of-concept versions of features provided by existing proof assistants and one feature that is

only supported via external tools. Specifically, we demonstrate that Cur:

- Enables users to define syntactic sugar and surface language features that are primitive in the surface languages of other proof assistants (section 4). We build a proof assistant by implementing a surface language as a user-defined library. This library provides notation including `let`, non-dependent function arrows, automatic currying, pattern matching on inductive types, and dependently-typed staged meta-programming. This library is less than 400 lines of code.
- Enables users to define tactic languages for writing proofs, including interactive tactic-based proving (section 5). While existing proof assistants like Coq and VeriML (Stampoulis and Shao 2010) provide tactic languages, they either do not support user-defined tactic languages, or require users to use external toolchains. Our design enables writing the tactic language in a library, using the same extension system that provides syntactic sugar. Our implementation of this tactic system, excluding tactics, is less than 200 lines of code. By comparison, the Mtac plugin for Coq is over 1200 lines of code.
- Enables users to define domain-specific languages for writing formal models (section 6). In particular, we define a library that enables modeling programming languages using BNF and inference rule notation, and extracting the models to Coq and LaTeX in addition to using them in Cur directly. This library is inspired by Ott (Sewell et al. 2007), an external tool that outputs files for multiple proof assistants from a single file with BNF and inference rule notation. While Ott is a mature tools with many more feature so comparing directly is difficult, our library is less than 400 lines of code, while just the lexer for Ott is more than 400 lines of code. No other proof assistant supports BNF and inference-rule notation in the language, nor provides support for users to add the feature as a library in just 400 lines of code.

2. Core Language

Our choice of core language is not vital to our design. We choose a dependently-typed calculus because dependent types have proven to be useful not only for theorem proving but also for verified programming. Whatever the choice, we want to restrict the core language to contain only features necessary to represent the models, proofs, and programs that we are interested in. Features that are not required for *expressivity* (Felleisen 1990), that only add convenience or reduce burden for the user, should be implemented using language extension.

Our core language, Curnel, is a dependently-typed λ-calculus with inductive families (Dybjer 1994), an infinite hierarchy of cumulative predicative universes, and an impredicative universe. Curnel is inspired by TT, the core language of Idris (Brady 2013), and similar to Luo's UTT (Luo 1994). Curnel is implemented in Redex (Felleisen et al. 2009; Matthews et al. 2004) and all figures in this section are extracted from the Redex implementation, so some notation may be slightly non-standard.

The top of Figure 1 presents the syntax of Curnel. A Curnel term is either a universe U, a function $\lambda x{:}t.e$, a variable, a dependent function type $\Pi x{:}t.t$, an application $(e\ e)$, or the elimination of an inductive type elim_D motive (methods ...) e. We write universes of level i as Unv i. We usually write variables using the meta-variable x, but we use D for the name of declared inductive types and c for the names of inductive-type constructors. We use application contexts Θ to represent nested application, and Ξ to represent nested product contexts (telescopes).

$$
\begin{array}{rcl}
n, i, j, k & ::= & \textit{natural} \\
U & ::= & \text{Unv } i \\
D, x, c & ::= & \textit{variable-not-otherwise-mentioned} \\
\Gamma c & ::= & \varnothing \mid (\Gamma c\ (c : t)) \\
\Delta & ::= & \varnothing \mid \Delta, D{:}[n]t := \Gamma c \\
t, e & ::= & U \mid \lambda x{:}e.e \mid x \mid \Pi x{:}e.e \mid (e\ e) \mid \text{elim}_D\ e\ (e\ ...)\ e \\
\Xi, \Phi & ::= & [] \mid \Pi x{:}t.\Xi \\
\Theta & ::= & [] \mid (\Theta\ e)
\end{array}
$$

$$
\begin{array}{rcl}
(\lambda x{:}t_0.t_1\ t_2) & \rightarrow\beta & t_1[t_2/x] \\
\text{elim}_D\ e_{motive}\ (e_m\ ...)\ \Theta_c[c] & \rightarrow\iota & \Theta_{mi}[e_{mi}]
\end{array}
$$

Figure 1: Curnel Syntax and Dynamic Semantics

The language is parameterized by a sequence of strictly-positive inductive-type declarations Δ. The declaration $\Delta, D{:}[n]t := \Gamma c$ extends Δ with the inductive type D of type t, with n parameters, whose constructors are declared by Γc. While users must explicitly apply constructors to their parameter arguments in the core language, the elimination principle guarantees that they are invariant across the definition.

As an example of writing in Curnel, we can encode the natural numbers and write the addition function as follows. First we define the natural numbers:

```
∅,Nat:[0]Unv 0 := (∅,z:Nat (s : Πx:Nat.Nat))
```

Next, we define addition.

```
λn:Nat.λm:Nat.elim_Nat λx:Nat.Nat (m λn-1:Nat.λih:Nat.(s ih)) n
```

Note that in Cur n-1 is a valid identifier.

We annotate the eliminator with the type D being eliminated.

The next argument to the eliminator is the *motive*, a type constructor used to compute the result type of the elimination. The motive is a function that takes the indices of the inductive type and the argument being eliminated, and computes the return type. In this case, the motive is $\lambda x{:}\text{Nat}.\text{Nat}$, a constant function that tells us the result type of addition is Nat.

The next argument is a sequence of are *methods*. The eliminator requires one method for each constructor of the inductive type being eliminated. For natural numbers, there are two constructors and thus two methods. Each method takes the arguments to its corresponding constructor and *inductive hypotheses*, the result of recursively eliminating the recursive arguments to the constructor. The method for z is just the constant m; it takes no arguments since the constructor z takes no arguments. The method for s takes two arguments: one for the argument to s and one for the recursive elimination of that argument, since the argument is also a Nat.

The final argument is the *discriminant*, *i.e.*, the value to eliminate, in this case the argument n.

The bottom of Figure 1 presents the small-step reduction of Curnel. The dynamic semantics of Curnel are standard, with β-reduction and folds over inductive types D. The fold over an inductive type takes a step when the discriminant is a fully applied constructor c of the inductive type D. we step to the method corresponding to the constructor applied to arguments Θ_{mi}, where these arguments are computed from the constructor's arguments, and the recursive application of the eliminator to recursive arguments. We omit the definitions of various meta-functions used in the reduction relation as they are not instructive. We extend these small-step rules for a call-by-value normalization strategy in the usual way.

Rather than explain in more detail the semantics of eliminators in general, we give an example reducing an eliminator. Continuing with our addition example, suppose we have called the addition function with (s z) and z. Then the eliminator will take a step as follows:

$$
\begin{array}{l}
\text{elim}_{Nat}\ o\ (m_0\ m_1)\ (\text{s z}) \rightarrow ((m_1\ \text{z})\ \text{elim}_{Nat}\ o\ (m_0\ m_1)\ \text{z}) \\
\text{where:} \\
\quad U\ =\ \text{Unv } 0
\end{array}
$$

o = λx:Nat.Nat
m_0 = z
m_1 = λn-1:Nat.λih:Nat.(s ih)

Since the eliminator is applied to (s z), the second constructor for Nat, we step to a use of the second method m_1. We pass this method the argument to s, and recursively eliminate that argument.

The type system of Curnel is a standard intuitionistic dependent-type theory, but we will present it briefly so that readers are aware of any differences between our system and existing proof assistants. Figure 2 presents the type system.

Starting at the top of the figure, the judgment $(U_0, U_1) \in \mathrm{A}$ gives typing for universes. Each universe Unv i has the type Unv $i + 1$.

The judgment $(U_0, U_1, U_2) \in \mathrm{R}$ encodes the predicativity rules for universes and the types of function spaces. We use predicativity rules similar to Coq; functions are impredicative in Unv 0, but are predicative in all higher universes. Unlike Coq, we do not distinguish between Prop and Set.

The subtyping judgment $\Delta;\Gamma \vdash t_0 \preccurlyeq t_1$, in the middle of Figure 2, defines when the type t_0 is a subtype of the type t_1. A type t_0 is a subtype of t_1 if they *equivalent*, *i.e.*, they reduce to α-equivalent terms in the dynamic semantics. Any universe is a subtype of any higher universe. A function type $\Pi x{:}t_0.e_0$ is a subtype of another function type $\Pi x{:}t_1.e_1$ if t_0 is *equivalent* to t_1, and e_0 is a subtype of e_1. Note that we cannot allow t_0 to be a subtype of t_1 due to predicativity rules.

In the bottom of the figure, we define term typing. We separate term typing into two judgments to simplify algorithmic implementation of convertibility during type checking. The type-inference judgment $\Delta;\Gamma \vdash e \Rightarrow t$ infers the type t for the term e under term environment Γ and declarations Δ. Note that this inference is trivial since Curnel terms are fully annotated. Without loss of generality, we assume the inference judgment can return a type in normal form if required.

The type-checking judgment $\Delta;\Gamma \vdash e \Leftarrow t$ checks that term e has a type t_0 that is convertible to the type t. The typing for eliminators is based on CIC's typing of `case`, but extended to handle recursive arguments. We omit the presentation as it is standard.

3. Designing for Language Extension

If we were following the design of other proof assistants, we would now describe a practical surface language. The surface language would include features that are not necessary for expressivity but simplify development. We would then write an elaborator that transforms the surface language into the core language. In this paper, we instead essentially develop an expressive and extensible elaborator and expose it to the user. This elaborator is the language-extension system.

As with our choice of core language, the choice of language-extension system is not vital to our design. We choose Racket's language-extension system as it is the subject of active research and development and already supports convenient and sophisticated extension, so we need only ensure the extensions are safe. We imagine that if a proof assistant like Coq were implemented using our design, it would feature a language-extension system written in OCaml. Whatever the choice, we want the language-extension system to enable the users to conveniently write safe and sophisticated extensions.

We will refer to the "metalanguage" as the language in which users write extensions and the "object language" as the language in which users write proof and formal models. In fact, Cur supports an infinite hierarchy of metalanguages; we do not discuss this further in this paper, but refer the interested reader to the literature on multi-stage programming—Taha (2004), for example.

Safe Extension
For extensions to be *safe* in the context of a proof assistant, all extensions must be valid—in the sense that they do not introduce inconsistency—according to the core language. The simplest and most convenient way to ensure safety is to check the output of all extensions after elaboration into the core language. The biggest drawback to this approach is that type errors may arise in expanded code, rather than in the code written by the user.

To improve type errors, we enable extension authors to check subforms and report errors during expansion. While our design enables extension authors to overcome the problem, other approaches avoid the problem entirely by typing extensions. We discuss these alternatives in section 7.

Convenient Extension
To ensure extensions are *convenient* to write, we must allow extension authors to reuse language infrastructure and automatically integrate extensions into the object language. Extension authors should not need separate toolchains to write or integrate new extensions; the language-extension system should be an integral part of the proof assistant. Users should be free to write metalanguage and object language code in the same module. Users should only need to write the semantics of extension; the extensions should automatically integrate into the parser and receive parsed objects to compute over. All extensions should be automatically checked for safety.

By comparison, Coq plugins require users to compile against the Coq implementation using a separate toolchain. Users abandon their project when they find they require a plugin.[1]

Writing extensions as external preprocessors, such as Ott (Sewell et al. 2007), requires users to write a parser and compiler, and the resulting tool does not integrate into the object language. This adds a barrier to both developing and using such extensions.

Sophisticated Extension
To support *sophisticated* extension, we must allow extensions to perform computation in a general-purpose metalanguage. New extensions should integrate into the syntax of the object language as if they were native syntax. Users must be able to redefine and extend existing syntax, including base syntax like λ and application.

Language-extension should also support extensions to non-syntactic features of the language. A user may want to extend the reader to parse new literals, rather than just perform rewrites on the AST of the object language. A user may want to extend the interpretation of a module to perform advanced type inference before the existing type checker runs.

Mixfix notation in Agda only supports defining new functions whose arguments appear in non-standard positions. It does not support defining a form whose subforms are not evaluated as object language expressions; we give an example of such a form in section 6.

Coq features notations and Idris (Brady 2013) features macros, but these are limited to simple syntactic rewrites. They do not support general-purpose computation nor redefining existing syntax.

3.1 Racket Languages and Language Extension

To describe how we implement Cur and language extension in Cur, we must first explain Racket's language-extension facilities. Racket is both a language and a system for defining languages (Tobin-Hochstadt et al. 2011). We use Racket as both: we implement Cur as an object language in Racket as a system, and write language extensions in Cur using Racket as the metalanguage. We use Racket because Racket's existing language extension features support con-

[1] There are many examples on the Coq-Club mailing list.

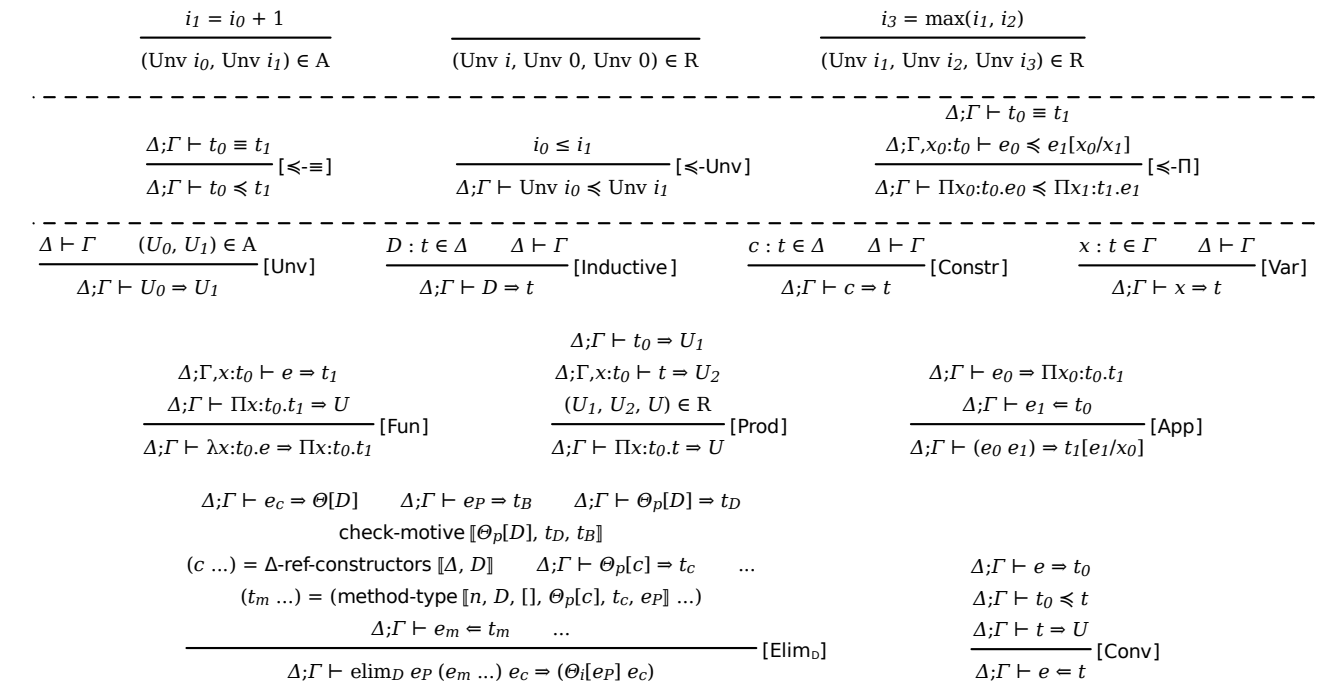

Figure 2: Cur's Type System (excerpts)

venient and sophisticated extensions as defined earlier in the section. We describe how we enforce safety later in this section.

Each Racket library provides a set of definitions that includes both syntactic forms such as **`lambda`** or **`define`** and values such as **`Nat`** or **`plus`**. Roughly speaking, definitions exist in one of two phases: compile-time and run-time. Compile-time definitions include both syntactic forms and value definitions. Run-time definitions include only value definitions. Compile-time definitions are written in a metalanguage, while run-time definitions are written in an object language. Defining new syntactic forms extends the object language. In Racket (the language), the metalanguage and object language are the same. In Cur, we create a new object language but leave Racket as metalanguage.

We define syntactic forms using syntactic macros. Each macro binds an identifier to a *transformer*, a metalanguage function on *syntax objects*—a data type representing object language syntax. For example, if we assume the object language contains the form λ, we can define a new ASCII version **`lambda`** as follows:

```
; Start a metalanguage block
(begin-for-syntax
  ; A metalanguage function definition
  (define (transform-lambda syn)
    ; Expect ":" to be a literal symbol
    (syntax-case syn (:)
      [(_ (x : t) e)
       #'(λ (x : t) e)])))

; Defines an object language "lambda" form
(define-syntax lambda transform-lambda)
```

The form **`begin-for-syntax`** starts a metalanguage block; it can contain arbitrary Racket definitions that will only be visible at compile-time.

The transformer function **`transform-lambda`** uses **`syntax-case`** to pattern match on the syntax object. It also takes a set of literals. This set, **`(:)`**, declares that **`:`** should be treated as a literal and not as a pattern variable. Where _ appears in a pattern, we do not bind a pattern variable.

In **`transform-lambda`**, we ignore the first element of the syntax, as that will be the name of the macro. In the body of the clause, we use syntax quote **`#'`** to create a template for a new syntax object. Pattern variables are bound inside the template. We simply preserve the pattern of the syntax object, and replace the macro identifier with the unicode name λ. In a practical implementation, we would add additional conditions on the pattern variables, such as check that **`x`** is an identifier rather than an arbitrary expression.

Finally, outside the meta-language block, we declare the new syntactic form using **`define-syntax`**, and it is added to the object language.

The first stage of running a program is to run the macro expander. The expander recursively traverses the syntax and calls the associated transformer when it reaches a use of a macro identifier. This recursive expansion enables macros to generate calls to other macros, and to build abstractions for defining macros.

In Racket, we can even use macros to extend and redefine language features that do not normally have an associated syntactic identifier, like the semantics of application. Language features that do not normally have an associated identifier have a secondary explicit name. For example, while we normally write application **`(f e)`**, this is just a special syntax for **`(#%app f e)`**. We can redefine application by redefining **`#%app`**, exactly as we defined **`lambda`**. Similarly, we can redefine the semantics of a module by redefining the **`#%module-begin`** form. The ability to redefine language features enables the most sophisticated language extensions and allows us to define new object languages in Racket.

We define a new language by defining a library that provides the base syntactic forms of the language and a definition for **`#%module-begin`** to implements the semantics of a module. Each Racket module begins with a line of the form **`#lang name`**, where **`name`** is the name of a library. This causes Racket to use the library **`name`** to interpret the module.

We can also define extensions to the language reader which allow adding new literals or entirely new kinds of syntax for writing object and metalanguage programs. A reader mixin is an extension to the reader. Users can use reader mixins by adding them

before the language name in the `#lang name` line. For example, `#lang sweet-exp cur` adds the sweet-expression reader `sweet-exp` to Cur, allowing users to write Cur using sweet-expressions instead of Racket's usual s-expressions. Using reader mixins does not affect how macros are written, since macros are defined on syntax objects which the reader returns, and not on, say, token streams.

3.2 Implementing Cur

We define Cur as a Racket language invoked using `#lang cur`. Cur uses Racket as the metalanguage, but replaces the object language. The base syntactic forms of Cur are the Curnel forms given section 2. For example, we can write the identity function as:

```
#lang cur

(λ (A : (Type 0)) (λ (a : A) a))
```

To eliminate some of the syntactic noise of s-expressions, we can write the same code using sweet-expressions. Sweet-expressions are similar to s-expressions, but infer some structure from indentation, provide some support for infix notation, and support the `$` operator used in Haskell for controlling precedence. S-expressions are also valid sweet-expressions, so we can still express structure manually when necessary. The rest of the example in this paper will use sweet-expression syntax.

```
#lang sweet-exp cur

λ (A : (Type 0)) $ λ (a : A) a
```

Cur also provides a `define` form for creating run-time value definitions and a `data` form for defining inductive types:

```
define id $ λ (A : (Type 0)) $ λ (a : A) a

data Nat : 0 (Type 0)
 z : Nat
 s : (Π (x : Nat) Nat)
```

The base forms plus `define` and `data` make up the default object language. The module semantics recursively expand all syntactic forms into base forms. The forms `data` and `define` generate no code, but affect the module's environment. The `data` form extends the module's inductive type declaration Δ. The `define` form extends the module's value definitions. As each syntactic form in the module is expanded, prior value definitions are inlined and added to the term environment Γ, the expanded term is type-checked, and top-level expression are normalized and printed.

We can write macros as we saw in the previous section to extend the syntax of Cur. The example of the ASCII `lambda` is a valid Cur extension. As in Racket, we can also build and use metalanguage abstractions to simplify defining new language extensions. For instance, the previous example required a lot of code to simply add another name to an existing form. Instead, we could use the following metalanguage abstraction that generate transformers that just replace the macro identifier with another identifier:

```
define-syntax lambda
  make-rename-transformer(#'λ)

; id, now without unicode
define id
  lambda (A : (Type 0)) $ lambda (a : A) a
```

3.3 Reflection API

Some of the extensions we want to write check and infer types, and run Cur terms at compile-time. We can implement staged meta-programming by running Cur terms at compile-time. We can add type errors to extensions by type-checking during macro expansion. We therefore provide a *reflection API*—a metalanguage API to the Curnel implementation—for language extensions to use. We explain the API functions that we use in the rest of this paper. Note that these API functions are added to the metalanguage, not the object language, and can only be used by extensions during expansion and before object language code is checked in the core language.

```
(cur-expand syn id ...) → SyntaxObject
  syn : SyntaxObject
  id : Identifier
```

The `cur-expand` function runs the macro expander on `syn` until the expander encounters a base form or one of the identifiers in the `id ...` list. The resulting term is not a fully expanded Curnel term; expansion halts when the top-level form begins with any of the identifiers in the `id` list or any of the Curnel base forms. For instance, `(cur-expand #'(λ (x : t) e))` does not expand `t` or `e` since `λ` is a base form. This function lets users write extensions by only considering certain syntactic forms. Since users can arbitrarily extend the syntax of Cur, using `cur-expand` before pattern matching in a new extensions ensures all unexpected extensions are already expanded.

```
(cur-type-check? term type) → Boolean
  term : SyntaxObject
  type : SyntaxObject
```

The `cur-type-check?` function returns `true` when `term` expands to e, `type` expands to t, and $\Delta;\Gamma \vdash e \Leftarrow t$ holds, and returns `false` otherwise. Note that this function is not meant to provide an error message; it is meant to be used by extensions that catch type errors in surface syntax and provide their own error messages.

```
(cur-type-infer term) → (Maybe SyntaxObject)
  term : SyntaxObject
```

The `cur-type-infer` function returns a syntax representation of t when `term` expands to the term e and $\Delta;\Gamma \vdash e \Rightarrow t$ holds, and `false` otherwise. This function allows users to build type inference into extensions and reduce annotation burden.

```
(cur-normalize term) → SyntaxObject
  term : SyntaxObject
```

The `cur-normalize` function is Cur's version of `eval`, but usable only by extensions at compile-time. It essentially calls the implementation of module semantics explained earlier: it expands all extensions, type checks the result, then normalizes the term in the Curnel. Specifically, the function returns a syntax representation of e_1 when `term` expands to a well-typed term e_0 and $e_1 = \mathsf{reduce}[\![\Delta, e_0]\!]$. This lets users explicitly reduce terms or simplify proofs when the type system or other extensions might not evaluate far enough.

This is similar to a feature in Zombie (Casinghino et al. 2014); Zombie users can write potentially non-terminating program to compute proofs, but to do so must explicitly force evaluation and thus act as a termination oracle. As Cur is terminating, the similarity is fleeting.

This function also enables us to implement staged meta-programming and run-time reflection without extending the core language.

4. Growing a Surface Language

Cur provides an object language with no more convenience than Curnel. It contains only features necessary for expressivity and nothing that is macro expressible in terms of other features (Felleisen 1990). By contrast, Gallina, the core language of Coq, includes extensions that are macro expressible in terms of other features, such as a non-dependent arrow and multi-arity functions.

Let us now take on the role of a Cur user (more precisely, an extension author) and begin implementing a surface language.

We begin by implementing the simple syntactic sugar like the non-dependent arrow notation. We then implement more sophisticated syntax sugar: we implement a `let` form with optional type annotations, and a pattern matching form for eliminating inductive types. We conclude with some extensions that go beyond syntax—extensions that add compile-time behaviors but do not necessarily generate code, like debugging features and staged metaprogramming.

4.1 Simple Syntax

Alias for `(Type 0)`

Writing `(Type 0)` for all these examples is somewhat tedious. We start with a simple example macro that elaborates `Type` to `(Type 0)`. Eventually, a more sophisticated extension could resolve all universe levels automatically, but this will do for now. First, let us import a renamed copy of the default form:

```
require
  only-in curnel
    Type default-Type
```

Next we define a simple macro with two syntaxes: one applied to an argument, one without any argument. In the first case, we simply expand to the default form. In the second case, we provide level `0` as a default. Note that the choice of surface syntax does not affect how we write syntax objects.

```
define-syntax (Type syn)
  syntax-case syn ()
    [(Type i) #'(default-Type i)]
    [Type #'(default-Type 0)]
```

Multi-Arity Syntax

Cur provides only single-arity functions in the base language. As mentioned in section 3, we can redefine existing forms like function and application syntax, so we redefine them to support multi-arity functions via automatic currying. First, let us import renamed copies of the default forms:

```
require
  only-in curnel
    #%app default-app
    λ default-λ
```

With these renamed copies, we can redefine `#%app` and λ while still generating code that uses the original forms.

Next, we define a simple recursive macro for λ that curries all arguments using `default-λ`.

```
define-syntax λ
  syntax-rules (:)
    [(λ b) b]
    [(λ (a : t) (ar : tr) ... b)
     (default-λ (a : t) (λ (ar : tr) ... b))]

define-syntax lambda
  make-rename-transformer #'λ
```

The `syntax-rules` form is similar to `syntax-case`, but specialized to support writing simple syntactic rewrites rather than arbitrary metalanguage computation. The syntax `...` is part of the pattern and template languages for syntax objects. In a pattern, it matches a list of the preceding pattern. In a template, it splices in the list indicated by the preceding pattern.

Next, we redefine application. The macro automatically curries applications using `default-app`.

```
define-syntax #%app
  syntax-rules ()
    [(#%app e1 e2)
     (default-app e1 e2)]
    [(#%app e1 e2 e3 ...)
     (#%app (#%app e1 e2) e3 ...)]
```

These forms are automatically integrated into the object language, replacing the old forms, and are ready to use even later in the same module:

```
define id $ lambda (A : Type) (a : A) a
id Nat z
```

Non-dependent Arrow Syntax

Now let us define a non-dependent arrow form. We start by defining a single-arity arrow syntax `arrow`:

```
define-syntax (arrow syn)
  syntax-case syn ()
    [(arrow t1 t2)
     #`(Π (#,(gensym) : t1) t2)]
```

Note that in Cur we must explicitly generate a fresh name using `gensym`, due to a limitation in the representation of names in our object language. We use syntax quasiquote `#`` to create a syntax template that supports escaping to compute parts of the template, and use syntax unquote `#,` to escape the template and run metalanguage expressions.

Now we can easily define the multi-arity arrow, with both ASCII and unicode names.

```
define-syntax ->
  syntax-rules ()
    [(-> a) a]
    [(-> a a* ...)
     (arrow a (-> a* ...))]

define-syntax → make-rename-transformer(#'->)

; Usage:
data Nat : 0 Type
  z : Nat
  s : {Nat → Nat}
```

Top-level Function Definition Syntax

Writing top-level function definitions using `lambda` is verbose. Most languages features special syntax for conveniently defining top-level functions, so let us add this to Cur:

```
define-syntax define
  syntax-rules (:)
    [(define (id (x : t) ...) body)
     (default-define id (lambda (x : t) ... body))]
    [(define id body)
     (default-define id body)]

define (id (A : Type) (a : A)) a
```

Notation for Formal Models and Proofs

Recall that our original goal was to provide better support for user-defined notation in formal models and proofs. Thus far, we have merely defined a surface language, which users should not be expected to do. However, the same language-extension facilities serve both purposes. As language implementers, we use language extension to build a surface language. As users, we use language extension to define our own notation—and we have the same power as language implementers when defining new notation.

For instance, suppose we as a user model the simply-typed λ-calculus. After writing a small-step evaluation relation and a type-checking relation, we want to use standard notation while doing proofs about the model. We define this notation as follows:

```
define-syntax-rule (↦ e1 e2) (steps-to e1 e2)

define-syntax ⊢
  syntax-rules (:)
    [(⊢ Γ e : t) (type-checks Γ e t)]
```

```
define-syntax nfx
  syntax-rules (⊢ :)
    [(nfx Γ ⊢ e : t) (⊢ Γ e : t)]
```

The form `define-syntax-rule` is simply syntax sugar using `define-syntax` followed by `syntax-rules`. The `nfx` macro is used by the sweet-expression reader to extend the reader with new infix notation. The reader can automatically parse some infix notation, but the `type-checks` notation is irregular so we define the `nfx` macro to assist the reader.

Now we can use the notation to state a lemma:

```
define lemma:type-preservation
  (forall (e1 : STLC-Term) (e2 : STLC-Term)
          (Γ : STLC-Env) (t : STLC-Type)
    {{Γ ⊢ e1 : t} -> {e1 ↦ e2} -> {Γ ⊢ e2 : t}})
```

The sweet-expression reader provides some support for infix notation, but other work has implemented more sophisticated support for non-s-expression based syntax extensions. Jon and Matthew (2012) used the extensible reader to implement syntax-extension for algebraic notation. Other work has even developed language-extension via syntactic macros based on parsing expression grammars, rather than on pre-parsed s-expressions representations (Allen et al. 2009).

4.2 Sophisticated Syntax

The extensions in the previous section are simple syntactic rewrites, but recall that our goal is to support *sophisticated* user-defined extensions. The extensions we study in this section use metalanguage computation to compute parts of the object language code.

Let Syntax

Let us begin by defining `let` in terms of application. Note that a proper dependent `let` requires changing the type system, so this `let` is only a programming convenience. We define the `let` construct with two syntaxes: one expects a type annotation, while the other attempts to infer the type. In the first syntax, when there is a type annotation, we manually type check the annotated term before generating code so that we can report an error in the surface syntax. In the second syntax, when there is no annotation, we attempt to infer a type and report an error if we cannot.

```
define-syntax (let syn)
  syntax-case syn (: =)
    [(let ([x = e : t]) body)
     (unless (cur-type-check? #'e #'t)
       (error 'let
         "~a does not have expected type ~a"
         #'e
         #'t)
     #'((λ (x : t) body) e))]
    [(let ([x = e]) body)
     (unless (cur-type-infer #'e)
       (error 'let
         "Could not infer type for ~a; ~a"
         #'e
         "try adding annotation via [x = e : t]")
     #`((λ (x : #,cur-type-infer(#'e)) body)
        e))]
```

Pattern variables are only bound inside syntax templates, so we use syntax quote to refer `e` and `t` in metalanguage code. The metalanguage function `error` takes a symbol naming the form in which the error occurred, a format string where `~a` is a formatting escape, and a list of arguments for the format string. Syntax objects have associated source location information, so we could even report error messages with the file name and line number of the `let` expression, but we omit this for clarity.

Pattern Matching Syntax

Recall the addition function for natural number we defined in section 2, which we redefine as `+` below:

```
define (+ [n1 : Nat] [n2 : Nat])
  elim Nat
    lambda (x : Nat) Nat
    (n2
     lambda (x : Nat) (ih : Nat) $ s ih)
    n1
```

This version of `+` is easier to read and write now that we have multi-arity functions, but still requires a lot of annotations and other syntactic overhead. Instead, we would like to define `plus` using pattern matching and to avoid writing obvious annotations, like the motive, when they can be inferred. We would like to define `plus`, for instance, like so:

```
define (+ [n1 : Nat] [n2 : Nat])
  match n1
    z n2
    (s (x : Nat)) s $ recur x
```

In this definition we use `match`, which we present shortly. The `match` form automatically infers the motive, the annotations on `elim`, and inductive hypotheses. It also provides the `recur` form to allows users to refer to generated inductive hypotheses by the name of the recursive argument from which they are derived.

Figure 3 presents the definition of the `match` extension. This simplified version demonstrates how to coordinate two different extensions, namely the `match` and `recur` forms, and how to generate and compose multiple syntax templates. It makes use of nontrivial metalanguage features, like state and user-defined datatypes structures. We omit parts of the implementation that do not contribute to the goal of demonstrating these features, and features like error checking code. Recall that `...` is an identifier used in patterns and templates, so we use `....` to indicate omitted code.

First, we transform the syntax representing a sequence of clauses into a list of syntax using `syntax->list`. We parse each clause into a structure using `clause-parse`. Each clauses consist of a pattern and a body. The pattern must be either a constructor name for constructors that take no arguments, or a constructor name followed by names with type annotations for all arguments to the constructor. We store the list of arguments and the body of the clause in a clause structure, to be used when generating methods for the eliminator.

After parsing each clause, we compute the motive. The body of the motive is the type `R` of the result of the `match`, and the argument to the motive has type `D` of the discriminant. Note that in this implementation, we do not attempt to handle indexed or parameterized inductive types. The full implementation can infer parameters and some indexed inductive types and supports optional annotation syntax for when inference fails.

Finally, we generate a method for each clause using `clause->method`. While generating methods, we infer which arguments are recursive arguments and compute the inductive hypotheses. We update a compile-time dictionary `ih-dict` that associates the name of the recursive argument to the generated name for that inductive hypothesis. The `recur` form looks up its argument in `ih-dict`.

4.3 Beyond Syntax

Thus far, all our examples demonstrate syntactic transformations. Our sophisticated language-extension system also supports creating syntactic forms that have semantic behavior at compile-time and do not necessarily generate object language code.

For example, we can create a type assertion form that allows users to check that an expression has a particular type and receive a type error if not:

```
define-syntax (:: syn)
  syntax-case syn ()
```

```
define-syntax (match syn)
  syntax-case syn ()
    ; expects discriminant e and list of clauses clause*
    (_ e clause* ...)
     let* ([clauses (map clause-parse (syntax->list #'(clause* ...)))]
           [R (infer-result clauses)]
           [D (cur-type-infer #'e)]
           [motive #`(lambda (x : #,D) #,R)]
           [U (cur-type-infer R)])
       #`(elim #,D #,motive
            #,(map (curry clause->method D motive) clauses)
            e)

define-syntax (recur syn)
  syntax-case syn ()
    (_ id)
     dict-ref ih-dict $ syntax->datum #'id

begin-for-syntax
  define-struct clause (args body)
  define ih-dict (make-hash)

  define (clause-parse syn)
    syntax-case syn
      (pattern body)
       make-clause (syntax-case pattern (:)
                     [c '()]
                     [(c (x : t) ...) (syntax->list #'((x : t) ...))])
                   #'body

  define (infer-result clauses)
    for/or ([clause clauses])
      cur-type-infer $ clause-body clause

  define (infer-ihs D motive args-syn)
    syntax-case args-syn () ....

  ; D needed to detect recursive arguments
  ; motive needed to compute type of inductive hypotheses
  define (clause->method D motive clause)
    let* ([ihs (infer-ihs D motive (clause-args clause))])
      dict-for-each ihs
        lambda (k v)
          dict-set! ih-dict k $ car v
      #`(lambda
          #,@clause-args(clause)
          #,@(dict-map ihs (lambda (k v) #`(#,(car v) : #,(cdr v))))
          #,clause-body(clause))
```

Figure 3: A Pattern Matcher for Inductive Types

```
    (_ e t)
     if $ cur-type-check? #'e #'t
         #'(void)
         (error '::
           "Inferred ~a; expected ~a."
           #'t
           (cur-type-infer #'e))
; Usage:
{z :: Nat}
```

We check during expansion that `e` has type `t`. If the check succeeds, we generate the no-op expression `#'(void)`. If the check fails, it reports an error similar to what we do for checking annotations in the `let` form. This form has no behavior in the object language but provides extra behavior at compile-time to support debugging.

We can also create a syntactic form that forces normalization at compile-time:

```
define-syntax (run syn)
  syntax-case syn ()
    (_ expr) $ cur-normalize #'expr
```

The `run` form does not provide new syntactic sugar, but transforms the syntax by normalization via the reflection API.

This can be used to simplify proofs or perform staged metaprogramming. For example, we specialize the exponentiation function `exp` to the `square` function at compile-time:

```
define square $ run $ exp (s (s z))
```

5. Tactics

In this section we describe a tactic system called *ntac* implemented in Cur. We begin with an example of using the tactic system to prove a trivial theorem:

```
ntac $ forall (A : Type) (a : A) A
  by-intro A
  by-intro b
  by-assumption
```

This example shows the type of the polymorphic identity function written using tactics. We use `ntac`, a form that builds an expression given an initial goal followed by a tactic script. This is similar to `Goal` in Coq, which introduces an anonymous goal that can be solved using an Ltac script. In this example we use the `by-intro` tactic, which takes a single optional argument representing

the name to bind as an assumption in the local proof environment. Then we conclude the proof with `by-assumption`, which takes no arguments and searches the local environment for a term that matches the current goal. Since all goals are complete at this point, we end the proof.

```
define-theorem id $ forall (A : Type) (a : A) A
  by-obvious
```

We can also use `define-theorem` to define a new identifier using an ntac script. The form `(define-theorem name goal script ...)` is simply syntax sugar for `(define name (ntac goal script ...))`. In this example, we use the `by-obvious` tactic which solves certain trivial theorems.

We begin implementing ntac by implementing the `ntac` form:

```
define-syntax (ntac stx)
  syntax-case stx ()
    [(_ goal . script) (ntac-interp #'goal #'script)]
```

The `ntac` form runs, at compile-time, the metalanguage function `ntac-interp` to generate an object language term. The function `ntac-interp` takes syntax representing an object language type, `#'goal` and syntax representing a sequence of tactics, `#'script`.

In ntac, we use proof trees `ntt` to represent partial terms in the object language with multiple holes and contextual information such as assumptions, and then use the ntac proof tree zipper `nttz` to navigate the tree and focus on a particular goal. Tactics are metalanguage functions on `nttz`s. We will not discuss this design or these data structures in more details here; the design is described in the Cur documentation.

Since tactics are just metalanguage functions, we can create syntactic sugar for defining tactics as follows:

```
define-syntax $ define-tactic syn
  syntax-rules ()
    [(_ e ...)
     (begin-for-syntax
       (define e ...))]
```

The form `define-tactic` is simply a wrapper for conveniently defining a new metalanguage function. Note that this extension generates metalanguage code, by generating a new metalanguage block containing a metalanguage definition. Until now, we have only seen extensions that compute using the metalanguage and generate code in object language, but recall from section 3 that Cur supports an infinite hierarchy of language extension.

Now let us write the tactic script interpreter. We begin by defining the function `run-tactic`, which takes a proof tree zipper and a syntax object representing a call to a tactic.

```
begin-for-syntax
  define $ run-tactic nttz tactic-stx
    define tactic $ eval tactic-stx
    tactic nttz
```

We use `eval` to evaluate the syntax representing the function name and get a function value. Then we simply apply the tactic to the proof tree zipper.

Finally, we define `ntac-interp` to interpret a tactic script and solve a goal.

```
begin-for-syntax
  define $ ntac-interp goal script
    define pt $ new-proof-tree $ cur-expand goal
    define last-nttz
      for/fold ([nttz (make-nttz pt)])
               ([tactic-stx (syntax->list script)])
        run-tactic nttz tactic-stx
    proof-tree->term $ finish-nttz last-nttz
```

We begin by generating a fresh proof tree which starts with one goal. The `for/fold` form folds `run-tactic` over the list of tactic calls with a starting proof tree zipper over the initial proof tree. After running all tactics, we check that the proof has no goals left, then generate a term from the proof tree.

Many operations work directly on the current proof tree, so it is cumbersome to define each tactic by first extracting the proof tree from the proof tree zipper. We introduce a notion of *tacticals*, metalanguage functions that take a context and a proof tree and return a new proof tree. We define a tactic `fill` to take a tactical and apply it at the focus of the proof tree zipper. With a notion of tacticals, we can easily define the tactical `intro` as follows:

```
define-tactical ((intro [name #f]) env pt)
  ntac-match $ ntt-goal pt
   [(forall (x:id : P:expr) body:expr)
    define the-name (syntax->datum (or name #'x))
    make-ntt-apply
     goal
     λ (body-pf)
       #`(λ (#,the-name : P) #,body-pf)
     list
      make-ntt-env
       λ (old-env)
         (hash-set old-env the-name #'P)
       make-ntt-hole #'body]
```

We define a new tactical `intro`, which takes one optional argument from the user `name`, and will be provided the local environment and proof tree from the ntac interpreter. In `intro`, we start by extracting the current goal from the proof tree. To pattern match on the goal we use the form `ntac-match`, a simple wrapper around the Racket `match` form that hides some boilerplate such as expanding the goal into a Curnel form and raising an exception if no patterns match. If the goal has the form of a dependent function type, we make a new node in the ntac proof tree that solves goal by taking a solution for the type of the body of the function and building a lambda expression in the object language. This node contains a subtree that makes the solution of `#'body` the new goal and adds the assumption that `name` has type `P` in the scope of this new goal.

To make the `intro` tactical easier to use at the top level, we define the `by-intro` tactic:

```
begin-for-syntax
  define-syntax $ by-intro syn
    syntax-case syn ()
      [_
       #`(fill (intro))]
      [(_ name)
       #`(fill (intro #'name))]
```

We create a metalanguage macro `by-intro` that takes a name as an optional argument. This macro expands to an application of the `fill` tactic to the `intro` tactical. We define `by-intro` as a macro so the user can enter a name for the assumption as a raw symbol, like `(by-intro A)`, rather than as a syntax object like `(by-intro #'A)`.

Since tactics are arbitrary metalanguage functions, we can define tactics in terms of other tactics, define recursive tactics, and even call to external programs or solvers in the metalanguage or even through the foreign-function interface of our metalanguage. Our next tactic, `by-obvious`, demonstrates these first two features. It will solve any theorem that follows immediately from its premises.

```
define-tactical $ obvious env pt
  ntac-match $ ntt-goal pt
    [(forall (a : P) body)
     ((intro) env pt)]
    [a:id
     (assumption env pt)]

define-tactic $ by-obvious ptz
  define nptz $ (fill obvious) ptz
  if $ nttz-done? nptz
```

```
    nptz
    by-obvious nptz
```

First we define the `obvious` tactical, which simply matches a goal and uses either `intro` or `assumption` to solve it. Then we define the `by-obvious` tactic which fills the hole using the `obvious` tactical and recurs until there are no goals left.

As we have the entire metalanguage available, we can define sophisticated tactics that do arbitrary metalanguage computation. For instance, since our metalanguage provides I/O and reflection, we can define interactivity as a user-defined tactic. We begin implementing interactive by first implementing the `print` tactic. This tactic will print the state of the focus of the proof tree zipper and return it unmodified.

```
define-tactic $ print ptz
  match $ nttz-focus ptz
    [(ntt-hole _ goal)
     for ([(k v) (in-hash (nttz-context tz))])
       printf "~a : ~a\n" k (syntax->datum v)
     printf "----------------------------\n"
     printf "~a\n\n" (syntax->datum goal)]
    [(ntt-done _ _ _)
     printf "Proof complete.\n"]
  ptz
```

We first match on the focus of proof tree zipper. If there is a goal, then we print all assumptions in the context followed by a horizontal line and the current goal. If the zipper indicates the proof has no goals left, then we simply print "Proof Complete".

Now we define the `interactive` tactic. This tactic uses the `print` tactic to print the proof state, then starts a read-eval-print-loop (REPL).

```
define-tactic $ interactive ptz
  print ptz
  let ([cmd (read-syntax)])
    syntax-case cmd (quit)
      [(quit) ....]
      [tactic
       (define ntz (run-tactic ptz tactic))
       (interactive ntz)]
```

The REPL reads in a command and runs it via `run-tactics` if it is a tactic. The REPL also accepts `quit` as a command that exits the REPL and returns the final proof state.

Now we have defined not only a user-defined tactic system, but a user-defined *interactive* tactic system. We can use the interactive tactic just like any other tactic:

```
ntac $ forall (A : Type) (a : A) A
 interactive
```

The following is a sample session in our interactive tactic:

```
---------------------
(forall (A : Type) (forall (a : A) A))

> (by-intro A)
A : Type
---------------------
(forall (a : A) A)

....

> by-assumption
Proof complete.

> (quit)
```

6. Olly - Ott-Like LibrarY

All the previous extensions are features of existing proof assistants. In this section we present Olly, a domain-specific language (DSL) for modeling programming languages, which provides features that no other proof assistant supports. Olly provides BNF notation for generating inductive types that represent programming language syntax. The BNF notation automatically converts variables in the syntax to de Bruijn indices. Olly also supports inference rule notation for modeling relations. Both notations support extracting the models to LaTeX and Coq, in addition to using the models directly in Cur.

Olly is inspired by Ott (Sewell et al. 2007), a tool for generating models of programming language semantics in different proof assistants from a single model written in a DSL. Ott is an external tool that generates files for each proof assistant, while Olly is integrated into the object language of Cur as a language extension.

BNF Notation

We begin with an example of defining of the syntax of the simply-typed λ-calculus using the `define-language` form. This language includes booleans, the unit type, pairs, and functions. Note that the `let` form is the elimination form for pairs in this language, and binds two names.

```
define-language stlc
  #:vars (x)
  #:output-coq "stlc.v"
  #:output-latex "stlc.tex"
  val  (v)    ::= true false unit
  type (A B)  ::= boolty unitty (-> A B) (* A A)
  term (e)    ::= x v (lambda (#:bind x : A) e)
              (app e e) (cons e e)
              (let (#:bind x #:bind x) = e in e)
```

The first argument to the form is a name for the language—`stlc` in this case. The next three lines are optional keyword arguments. The `#:vars` argument is a set of meta-variables that represent variables in the syntax. The `#:output-coq` argument is a string representing a file name; when given, a Coq representation of the language syntax is written to the specified file during compilation. Similarly, the `#:output-latex` argument is a string representing a file name; when given, a Latex rendering of the BNF grammar is written to the specified file during compilation. After the optional arguments, `define-language` expects an arbitrary number of non-terminal definitions from which it generates inductive types.

To better understand `define-language` non-terminal clauses, let us first look at the code generated for the `term` non-terminal.

```
data stlc-term : 0 Type
  Nat->stlc-term : {Nat -> stlc-term}
  stlc-val->stlc-term : {stlc-value ->
                         stlc-term}
  stlc-lambda : {stlc-type -> stlc-term ->
                 stlc-term}
  stlc-app : {stlc-term -> stlc-term -> stlc-
term}
  stlc-cons : {stlc-term -> stlc-term ->
               stlc-term}
  stlc-let : {stlc-term -> stlc-term ->
              stlc-term}
```

References to other non-terminals, such as the reference to `x`, result in *conversion constructors* which simply inject one non-terminal into the other. The names of the conversion constructors are generated from the types of the non-terminals with the sigil `->` between them, indicating conversion. For example, `Nat->stlc-term` is a `stlc-term` constructor that converts a `Nat` (representing a de Bruijn index) to a `stlc-term`. Other constructor names are generated from the name of the language, `stlc`, and the name of the constructor given in the syntax. For example, the constructor name generated from the `lambda` syntax is `stlc-lambda`.

More formally, the syntax of a non-terminal definition is `(nt-name (meta-variables ...) ::= syn-clause ...)`. As Cur does not currently support mutual inductive definitions, all

non-terminal definitions must be appear in order of their dependencies. Each `syn-clause` must be either a reference to a previously defined non-terminal, a terminal represented by a unique identifier, or an s-expression whose first element is a unique identifier.

For each non-terminal, we generate a new inductive type. We generate a constructors for the inductive type for each `syn-clause`. We prefix the name of each inductive type and each constructor by the language name. For references to previously defined non-terminals, we generate a constructor that act as a tag and injects the previous non-terminal into the new one.

For terminals, we generate a constructor that take no arguments and whose name is based on the terminal.

For s-expressions, we create a new constructor whose name is based on the identifier at the head of the s-expression and whose arguments' types are computed from the meta-variables that appear in the rest of the s-expression. We only use the non-meta-variable symbols such as `:` in the Latex rendering of the BNF grammar. The syntax `#:bind x` declares `x` to be a binding position, so it is not treated as an argument. Since we use de Bruijn indices for binding, binding positions are erased.

The `define-language` form allows us to create the model using BNF notation, but working with the model requires using the generated constructor names. Instead of using the constructors directly, we can write an extension that parses the `stlc` syntax into the appropriate constructors[2]. Figure 4 presents an excerpt of the parser. The form `begin-stlc` simply calls the metalanguage function `parse-stlc` with the syntax object `e` and a new hashtable. The `parse-stlc` function declares each of the constructor names and syntactic symbols as literals, and loops over the syntax object generating the constructors that correspond to the `stlc` syntax. It uses the hashtable to map variable names to de Bruijn indices. When parsing a `lambda`, it shifts each index in the hashtable. For convenience, the parser accepts the syntax `(e1 e2)` for application instead of `(app e1 e2)` and `1` for the unit type instead of `unitty`.

Inference-rule Notation

Figure 5 presents an example of using the inference rule notation. We use the `define-relation` form to model a type system for `stlc`. The `define-relation` form takes as its first argument a name for the new relation applied to the types of its arguments. This example defines the inductive type `has-type` which relates a list of `stlc-type`s, an `stlc-term`, and an `stlc-type`. Like the `define-language` form, `define-relation` takes optional arguments for generating Coq and Latex output. The rest of the form is interpreted as a sequence of inference rules. Each inference rule is a list of assumptions, followed by a horizontal line represented by an arbitrary number of hyphens, a name for the rule that will become the name of the constructor, and a conclusion that must be the relation applied to its arguments.

Figure 6 presents an excerpt of the implementation for `define-relation`. To implement this form, we use `syntax-parse` (Culpepper 2012), but only present enough of `syntax-parse` to undeerstand the key ideas in our implementation.

The `syntax-parse` form allows parsing syntax objects rather than merely pattern matching on them. The form allows specifying patterns as in `syntax-case`, but also support refining the patterns using syntax classes that specify subpatterns and side-conditions. We can apply a syntax class to a pattern variable by using the syntax `pv:class`, or by using the syntax `(~var pv class)`. Both declare the pattern variable `pv` must match the syntax class `class`, but the `~var` syntax is required when the syntax class takes arguments.

[2] It should be possible to generate this parser in the `define-language` extension, but we have not yet implemented that feature.

```
define-syntax (begin-stlc syn)
  syntax-case syn ()
    [(_ e) (parse-stlc #'e (make-immutable-
hash))]

begin-for-syntax
  define (parse-stlc syn d)
    syntax-case syn (lambda : prj * ->
                     let in cons bool
                     unit true false)
      [(lambda (x : t) e)
       #`(stlc-lambda
           #,(parse-stlc #'t d)
           #,(parse-stlc #'e
               (dict-set
                 (dict-shift d)
                 (syntax->datum #'x)
                 #`z)))]
      [(e1 e2)
       #`(stlc-app
           #,(parse-stlc #'e1 d)
           #,(parse-stlc #'e2 d))]
      [(cons e1 e2)
       #`(stlc-cons
           #,(parse-stlc #'e1 d)
           #,(parse-stlc #'e2 d))]
      ....
      [false #'(stlc-val->stlc-term stlc-false)]
      [bool #'stlc-boolty]

  define (dict-shift d)
    for/fold ([d (make-immutable-hash)])
             ([(k v) (in-dict d)])
      dict-set d k #`(s #,v)
```

Figure 4: Parser for STLC Syntax (excerpt)

```
define-relation
  (has-type (List stlc-type) stlc-term stlc-type)
  #:output-coq "stlc.v"
  #:output-latex "stlc.tex"
  [(g : (List stlc-type))
   ----------------------- T-Unit
   (has-type g (begin-stlc unit) (begin-stlc 1))]
  ....

; Generates:
data has-type : 0 (-> (List stlc-type)
                      stlc-term
                      stlc-type
                      Type)
  T-Unit : (forall (g : (List stlc-type))
                   (has-type
                     g
                     (stlc-val->stlc-term stlc-unit)
                     stlc-unitty))
  ....
```

Figure 5: STLC Type System Model (excerpt)

In the definition of `define-relation`, we declare that the name of the relation must be an identifier using the colon syntax and the syntax class `id`.

We declare that the next two arguments are optional using the special form `~optional`, which takes a pattern as its argument. We specify the pattern is a sequence of the keyword `#:output-coq` followed by a pattern variable `coq-file`. We use the syntax class `str` to refine the `coq-file` pattern, indicating that it must be a string literal. If this optional pattern matches, then the `coq-file` pattern variable is bound in an attribute map. The form `attribute` references the attribute map and returns `false` if the attribute is not bound.

```
define-syntax (define-relation syn)
  syntax-parse syn
    [(_ (name:id index ...)
        (~optional
          (~seq #:output-coq coq-file:str))
        (~optional
          (~seq #:output-latex latex-file:str))
        (~var rule (inference-rule
                     (attribute name)
                     (attribute index)))
        ...)
      ....]

begin-for-syntax
  define-syntax-class horizontal-line
    pattern
     x:id
     #:when (regexp-match?
              #rx"-+" (syntax->string #'x))

  define-syntax-class (conclusion n args r)
    pattern
     (name:id arg ...)
     #:fail-unless
       (equal? (syntax->symbol #'name)
               (syntax->symbol #'n))
       (format "Rule ~a: conclusion mismatch" r)
     ....

  define-syntax-class (inference-rule name
                                      indices)
    pattern (h ...
             line:horizontal-line
             rule-name:id
             (~var t (conclusion
                       name
                       indices
                       (attribute rule-name))))
     ....
```

Figure 6: Implementation of `define-relation` (excerpt)

After the optional arguments, we declare a pattern variable `rule` using the `~var` syntax, refine it using the `inference-rule` syntax class, and use `...` to match a list of inference rules.

Next we define the syntax class `inference-rule`. The `inference-rule` syntax class takes two arguments: the name of the relation and the list of indices. This syntax class matches when the syntax has the pattern of a list of hypothesis, followed by a horizontal line, a name, and a conclusion. The rule name must be an identifier.

The syntax class for a horizontal line converts the syntax to a string and uses a regular expression to match when the string is an arbitrary number of hyphens.

The syntax class for a conclusion uses the name of the relation and the indices to ensure the conclusion is the original relation applied to the right number of arguments.

7. Related Work

Proof assistants remain an active area of study, but much of this work focuses on aspects other than notation.

VeriML features a proof assistant with a small core over which the user can use ML for writing extensions (Stampoulis and Shao 2010). All extensions in VeriML are ML functions, and are therefore strongly typed. This enabled writing strongly typed tactics and decision procedures to manipulate or generate proof terms. Our work and the VeriML work would complement each other. Extensions in VeriML are ML functions, so the user is limited to the notation of function application but gets strongly typed extensions. Our work enables advanced notation, but we do not consider the issue of static guarantees for notation. A language extensions system designed with a strongly typed language such as ML would enable defining strongly typed notation.

Fisler et al. (1999) study the linguistic issues related to designing a "plug-and-play" theorem prover, in which users can select a set of logics and interfaces that can be that can be easily composed. In their theorem prover, the users must write a plugin that is loaded by the system. While their focus is on extending the logics of the prover, each extensions can define its own notation. However, notations from one extension cannot be used within the notation from another extension.

We ensure safe language extension by checking all code in the core language after expansion. This approach can result in type errors in expanded code that the user did not write, if the extension author is not careful. Alternative approaches ensure all type errors occur in the source language rather than the expanded language.

Lorenzen and Erdweg (2016) demonstrates how to define well-typed extensions that justify new typing rules in terms of the old type system. Since each extension is well-typed, type checking happens in the extended language rather than after expansion. However, these extensions are currently limited to desugaring and do not enable general purpose computation in a metalanguage.

Pombrio and Krishnamurthi (2015) develop techniques for *resugaring*; rather than catch errors before expanding extensions, these systems undo the expansion before reporting errors. Integrating this with a language-extension system could reduce the burden of manually catching errors in new extensions.

The Milawa theorem prover (Myreen and Davis 2014) allows the user to redefine the proof-checking function, after establishing that the new proof checker is valid. The new proof checker may admit new syntax and new axioms can report errors in terms of the extended proof language. This enables sophisticated and safe extensions, and ensures type errors occur in the surface language, but requires defining a new proof checker to define macro expressible syntactic transformations.

While we focus on notation in this work, language extension also provides support for meta-programming. As we saw in section 4, we can add support for using Cur as its own dependently-typed staged meta-programming language *without extending the core language*.

Previous work on dependently-typed staged meta-programming in Idris required extensions to the core language TT (Brady and Hammond 2006). Similar work in Agda requires extending Agda with a set of primitives (Devriese and Piessens 2013). Neither work demonstrates the soundness of these extensions.

Later work on meta-programming in Idris adds the ability to quote surface-language Idris, i.e. Idris code before elaboration (Christiansen 2014). By starting from a language-extension system with quasiquotation and syntactic macros, rather than adding quasiquotation to an existing elaborator, Cur supports not only quasiquoting surface syntax, but quasiquoting any user-defined extension to the surface syntax.

Bibliography

Eric Allen, Ryan Culpepper, Janus Dam Nielsen, Jon Rafkind, and Sukyoung Ryu. Growing A Syntax. In *Proc. of Workshop on Foundations of Object-Oriented Languages*, 2009.

Edwin Brady and Kevin Hammond. Dependently Typed Meta-Programming. In *Proc. of 7th Symposium on Trends in Functional Programming.*, 2006. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.106.7073&rank=3`

Edwin C. Brady. Idris, A General-Purpose Dependently Typed Programming Language: Design and Implementation. *Journal of Functional Programming* 23, pp. 552–593, 2013. `http://journals.cambridge.org/article_S095679681300018X`

Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Combining Proofs and Programs in a Dependently Typed Language. In *Proc. of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2014. http://doi.acm.org/10.1145/2535838.2535883

David Raymond Christiansen. Type-Directed Elaboration of Quasiquotations: A High-Level Syntax for Low-Level Reflection. In *Proc. of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages*, 2014.

Ryan Culpepper. Fortifying Macros. *Journal of Functional Programming* 22, pp. 439–476, 2012. http://journals.cambridge.org/article_S0956796812000275

Nils Anders Danielsson and Ulf Norell. Parsing Mixfix Operators. In *Proc. 20th International Symposium on Implementation and Application of Functional Languages*, 2008. http://dx.doi.org/10.1007/978-3-642-24452-0_5

Dominique Devriese and Frank Piessens. Typed Syntactic Metaprogramming. In *Proc. of the 18th ACM SIGPLAN International Conference on Functional Programming*, 2013. http://doi.acm.org/10.1145/2500365.2500575

Peter Dybjer. Inductive Families. *Formal Aspects of Computing* 6(4), pp. 440–465, 1994. http://dx.doi.org/10.1007/BF01211308

Matthias Felleisen. On the expressive power of programming languages. In *Proc. of the 3rd European Symposium on Programming*, 1990. http://dx.doi.org/10.1007/3-540-52592-0_60

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics engineering with PLT Redex*. 2009.

Kathi Fisler, Shriram Krishnamurthi, and Kathryn E Gray. Implementing Extensible Theorem Provers. In *Proc. of the International Conference on Theorem Proving in Higher-Order Logic: Emerging Trends*, 1999.

Rafkind, Jon and Flatt, Matthew. Honu: Syntactic Extension for Algebraic Notation Through Enforestation. In *Proc. Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, 2012. http://doi.acm.org/10.1145/2371401.2371420

Florian Lorenzen and Sebastian Erdweg. Sound Type-Dependent Syntactic Language Extension. In *Proc. of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016. http://doi.acm.org/10.1145/2837614.2837644

Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. 1994.

Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. A Visual Environment for Developing Context-Sensitive Term Rewriting Systems. In *Proc. Rewriting Techniques and Applications*, 2004.

Magnus O. Myreen and Jared Davis. The Reflective Milawa Theorem Prover is Sound. In *Proc. Interactive Theorem Proving*, 2014.

Justin Pombrio and Shriram Krishnamurthi. Hygienic Resugaring of Compositional Desugaring. In *Proc. Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, 2015. http://doi.acm.org/10.1145/2784731.2784755

Peter Sewell, Francesco Zappa Nardelli, Scott Owens , Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective Tool Support for the Working Semanticist. In *Proc. of the 12th ACM SIGPLAN International Conference on Functional Programming*, 2007. http://doi.acm.org/10.1145/1291151.1291155

Antonis Stampoulis and Zhong Shao. VeriML: Typed Computation of Logical Terms Inside a Language with Effects. In *Proc. of the 15th ACM SIGPLAN International Conference on Functional Programming*, 2010. http://doi.acm.org/10.1145/1863543.1863591

Walid Taha. A Gentle Introduction To Multi-Stage Programming. In *Proc. Domain-Specific Program Generation*, 2004.

Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages As Libraries. In *Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011. http://doi.acm.org/10.1145/1993498.1993514

Beta Ziliani, Derek Dreyer, Neelakantan R Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. Mtac: A Monad for Typed Tactic Programming in Coq. In *Proc. Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, 2013. http://doi.acm.org/10.1145/2500365.2500579