Project Report: Dependently typed programming with lambda encodings in Cedille

Ananda Guneratne, Chad Reynolds, and Aaron Stump

Computer Science, The University of Iowa, Iowa City, Iowa, USA
ananda-guneratne@uiowa.edu, chad-reynolds@uiowa.edu,
aaron-stump@uiowa.edu**

Abstract. This project report presents Cedille, a dependent type theory based on lambda encodings. Cedille is an extension of the Calculus of Constructions with new type features enabling induction and large eliminations (computing a type by recursion on a term) for lambda encodings, which are not available for lambda-encoded data in related type theories. Cedille is presented through a number of examples, including both programs and proofs.

1 Introducing Cedille

In this report, we describe a new project aimed at developing a dependently typed programming language called Cedille, based on lambda encodings. In dependent type theory, lambda encodings were abandoned in the 1980s, due to several serious problems: induction is provably not derivable [9], and one cannot use lambda encodings across multiple levels of the type theory making it impossible to compute both terms and types by recursion on lambda-encoded data. For these reasons, languages like Coq and Agda are based on a datatype subsystem, including case expressions or pattern matching, and special additional typing and reduction rules. With lambda encodings, one can avoid all this, and work with a pure lambda calculus. This simplifies the design and meta-theory of the language. Furthermore, one can use higher-order encodings, which correspond to datatype definitions with negative occurrence of the datatype being defined. These are disallowed in systems like Coq and Agda, but are allowed in languages like System F. Such datatypes have been proposed for representing expressions with binders, a long-standing challenge in functional programming and type theory (see, e.g., [26]).

The third author has developed new solutions to the problems of induction and large eliminations, in a type theory called the Calculus of Dependent Lambda Eliminations (CDLE), which has been proven consistent as a logic [23]. The main ideas are (1) to add a special form of recursive types where constructors are first declared, for purposes of stating a dependent elimination principle (which must mention the constructors), and then defined using a lambda encoding; and (2) to

** Guneratne and Reynolds are doctoral students.

use a lifting operation to lift simply typed terms to the type level, for large eliminations. In this report, we describe our initial experience with an implementation of CDLE called Cedille. We begin with an informal look at type checking (Section 2), and the user interface for Cedille (Section 3). Next, we recall the Parigot encoding, which is recommended for Cedille programming (Section 4). We then discuss equational reasoning in Cedille in general (Section 5), and through some examples with Parigot-encoded natural numbers (Sections 6 and 7). Next we give some examples with lists (Section 8), and a somewhat longer proof example, for transitivity of `compare` on natural numbers (Section 9). Finally, we consider a beginning example of a higher-order lambda encoding (Section 10), and conclude (Section 11).

1.1 Related Work

Probably the best-known lambda encoding is the Church encoding [4], typable in System F [3, 8]. The inherent inefficiency of predecessor, proved by Parigot [20], was addressed later by the same author, who proposed a new lambda encoding with constant-time predecessor [19]. The size of the normal form of Parigot-encoded natural number n is $O(2^n)$, but this does not result in inefficient computation using modern functional programming implementations [11]. Stump and Fu survey various lambda encodings and compare their runtime performance [24].

Coq and Agda are two prominent interactive theorem provers based on constructive type theory [15, 7]. Coq is based on the Calculus of Inductive Constructions (CIC) [27]. This formalism adds inductive datatypes to the Extended Calculus of Constructions [14], which is itself based on the original Calculus of Constructions [5]. The problems noted above – underivability of induction, lack of large eliminations (needed for deriving propositions like $0 \neq 1$), and (inessentially) inefficiency of accessor functions like predecessor – motivated the addition of inductive types as primitive [6]. Agda has a somewhat different basis than Coq: it uses predicative polymorphism only, where Coq has both predicative and impredicative quantification;, and it is based on pattern-matching equations including axiom K, which is not derivable in CIC [10]. Still, for purposes of comparison with Cedille, it is very similar to Coq, as it is also based on a primitive notion of inductive datatype. In contrast, in Cedille, all data are lambda-encoded and there are no primitive inductive types.

It is worth noting that while Coq and Agda remain active testbeds for new research in theorem proving and verification based on constructive type theory, ours is not the only project seeking new foundations. Geuvers and Basold propose a new type theory based solely on inductive and co-inductive types, without even function space as a primitive type form [1]. Work continues to create new systems for homotopy type theory (e.g., [25, 2]). One motivation for that work is to allow interchange of isomorphic structures, a central concern also for Morphoid Type Theory [16]. These efforts focus on different aspects of type theory than that targeted by Cedille, which seeks to provide a more fundamental account of data via lambda encodings.

2 Type checking in Cedille

The starting point for the CDLE type theory on which Cedille is based is the Calculus of Constructions (CC), extended with support for implicit arguments, taken from the Implicit Calculus of Constructions [18]. Implicit inputs to functions are ones which exist just for purposes of specification: they can be mentioned in the types of later arguments. A basic example for dependently typed programming is taking in the length of a vector as an implicit argument. Vector operations like reverse, for example, do not need to inspect this length computationally; it is just there to allow the statement of the type of the input (and output) vector. In Cedille, an **erasure function** is applied to terms to erase these implicit arguments, and other typing annotations, from terms, before equational reasoning.

To this basic type theory (CC plus implicit arguments), Cedille adds typing features to support induction and large eliminations with lambda encodings. Let us now consider these features and informally introduce the type-checking algorithm, using the example of the datatype of booleans.

```
rec Bool | tt : Bool , ff : Bool =
  ∀ P : Bool → ★ .
    P tt → P ff → P self
  with
    tt = Λ P . λ a . λ b . a ,
    ff = Λ P . λ a . λ b . b .
```

Fig. 1. The Cedille definition of the datatype of booleans

Figure 1 shows the Cedille definition for the inductive type `Bool`. Cedille supports top-level definition of inductive types using the `rec` keyword. Such definitions first declare types for the constructors of the datatype (here `tt` and `ff`) in terms of the name of the datatype (here `Bool`). At the end of the `rec` statement, after the `with` keyword, the constructors are then defined using a lambda-encoding. In between, a type is given which the named datatype is defined to equal. We call this the **body** of the datatype definition, in this case:

```
∀ P : Bool → ★ . P tt → P ff → P self
```

This type is allowed to mention the name of the datatype recursively, but only in a *positive* position (to the left of an even number of functional constructs in the type). Note that Coq and Agda require strict positivity (to the left of no functional constructs in the type). Here, we are saying that `Bool` is the type of those terms t such that for any property `P` of booleans, if one proves the property for `tt` and also for `ff`, then the property is proved also for t. The quantification over predicates on booleans is why the definition must be recursive. The dependence of the type of t on t itself is captured using the special variable `self`. Technically, this self reference is expressed using dependent intersection types [12], though the Cedille implementation hides all other details for these.

As just noted, the definition in Figure 1 ends with definitions for the constructors `tt` and `ff`, in this case using the Church encoding. These definitions are type-checked in a context where the name of the datatype is definitionally equal to the body, and the constructors are all assumed to have the typings and definitions given in the `rec` declaration. Importantly, however, the constructors may not occur free in the erasure of the body. This means that they can only be mentioned in erased positions, for purposes of dependent typing. Erased arguments are denoted with a minus sign in Cedille's input syntax. For `Bool`, they need not be mentioned at all. Cedille implements local type inference [21], allowing us to elide the types for the bound variables in the definitions of `tt` and `ff`. The capital lambda symbol is used to bind erased arguments. Erasing the definitions of `tt` and `ff` of Figure 1 indeed gives the standard definitions for Church-encoded boolean true and false (namely `λ a . λ b . a` and `λ a . λ b . b`).

```
and ⇐ Bool → Bool → Bool =
  λ x . λ y . x · (λ b : Bool . Bool) y ff .
```

Fig. 2. Definition of boolean conjunction

Figure 2 shows the Cedille definition of boolean conjunction. Cedille definitions use the symbol ⇐ to separate the symbol being defined from the type against which the defining term will be checked. Such definitions are processed in checking mode with local type inference; i.e., the term and the type are both inputs to the typing algorithm. The definition in Figure 2 erases to the standard one for Church-encoded booleans.

Note that to apply `x`, we must specify the instantiation of the predicate `P` from Figure 1, here `(λ b : Bool . Bool)`. Application of an expression to a type is denoted using ·. This instantiation is rather verbose, and can be avoided using McBride's idea of "elimination with a motive" [17]: we can instruct the type checker to use the type against which we are checking the application of `x` (namely `Bool`), to construct this predicate automatically. The Cedille notation for elimination with a motive is θ, and the more concise definition is then just:

```
and ⇐ Bool → Bool → Bool =
  λ x . λ y . θ x y ff .
```

In this case, the motive `(λ b : Bool . Bool)` ignores the input `b`. When we come below to proving theorems, however, the motives will make use of such inputs, to state non-trivial predicates to be proved.

As an example of Cedille's second novel typing construct, Figure 3 shows the standard impredicative definitions of types `True` and `False`, and a function mapping booleans to these types. In consistent pure type theories like the Calculus of Constructions, one cannot apply an expression at multiple levels of the language: Church-encoded booleans can be defined at the term level or at the type level, but they cannot be used across levels. Cedille features a lifting operator ↑ for lifting predicatively typed terms (no type quantifiers) to the type level. The expression `↑ X . b · (λ b : Bool . X) : (☆ → ☆ → ☆)` says that we are lifting the term `b · (λ b : Bool . X)` to the type level. The bound type variable `X` is a name that can be used to stand, at the type level, for the kind ⋆. The lifting expression specifies the resulting kind using a *lifting type* ☆ → ☆ → ☆. This lifting type is needed for type-level conversions involving these lifting types; otherwise the conversion relation would need to depend on the typing relation. We avoid this by including the lifting type in the syntax. We intend in future work to infer the lifting type during typing so it does not have to appear in the input syntax (while remaining part of the abstract syntax during conversion). A basic example of a conversion for lifting types is seen if we apply `Bool-to-type` to `tt`. Lifting `tt` to the type level yields, with the help of the lifting type, the following:

```
λ A : ⋆ . λ B : ⋆ . A
```

This is applied to `True` and `False`, reducing via type-level β-reduction to `True`. (Note that `True` and `False` are types, while `tt` and `ff` are terms.)

Note that lifting enables us to perform type-level computation with instantiations of impredicatively typed data, while competing approaches like level-polymorphism are restricted to predicative quantification (and hence not usable in practice for lambda encodings, though see Leivant's work on finitely stratified polymorphism [13]).

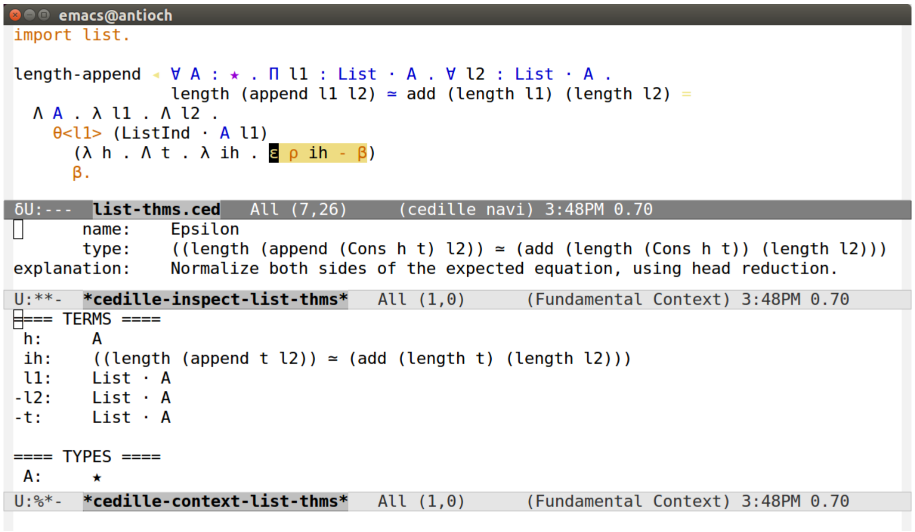
Typing is done modulo a relation of definitional equality, which automatically equates certain terms and types. Terms are equated if $\beta\eta$-equal, and types if they are β-equal or related by certain conversions for lifting types as just mentioned. As usual in type theory, it is customary to consider some notion(s) of provable equality, in addition to definitonal. We consider this topic shortly (Section 5).

```
True ⇐ ⋆ = ∀ X : ⋆ . X → X .
False ⇐ ⋆ = ∀ X : ⋆ . X .
Bool-to-type ⇐ Bool → ⋆ =
  λ b : Bool . ↑ X . b · (λ b : Bool . X) : (☆ → ☆ → ☆) · True · False .
```

Fig. 3. Function mapping booleans to `True` and `False`

3 User interface for Cedille

Before we look at further aspects of the Cedille language and examples of using it, let us consider briefly the user interface. Users interact with Cedille via an emacs interface, whose version at the time of writing is in around 1800 lines of elisp. This interface communicates with the backend Cedille tool, whose current version is around 4300 lines of Agda, not including parsers automatically generated from just under 300 lines of grammars. The basic interaction between the frontend (the emacs interface) and the backend is for the frontend to request that the backend parse a `.ced` file, namely the one the user is viewing in emacs. If the buffer is parsable, the backend sends back an annotated parse tree in a certain standard format. The format for the parse tree is as a list of **spans**, where a span is a starting and ending character position in the file, together with a name for the kind of parse-tree node this span represents, and a list of extra annotations. The frontend uses a generic elisp library called `se` (for "structured editing") to reassemble a parse tree from the list of spans. It is then possible to navigate through the text in the emacs buffer following the syntactic structure of the code. One highlights a span, and then can navigate up, down, left, or right in the parse tree with single keystrokes (e.g., "p" for up in the parse tree), shifting the highlighted span as one goes.



```
import list.

length-append ◂ ∀ A : ★ . Π l1 : List · A . ∀ l2 : List · A .
                length (append l1 l2) ≃ add (length l1) (length l2) =
  Λ A . λ l1 . Λ l2 .
    θ<l1> (ListInd · A l1)
      (λ h . Λ t . λ ih . ε ρ ih - β)
      β.

δU:---  list-thms.ced    All (7,26)     (cedille navi) 3:48PM 0.70
       name:    Epsilon
       type:    ((length (append (Cons h t) l2)) ≃ (add (length (Cons h t)) (length l2)))
explanation:    Normalize both sides of the expected equation, using head reduction.

U:**-  *cedille-inspect-list-thms*   All (1,0)      (Fundamental Context) 3:48PM 0.70
==== TERMS ====
 h:     A
 ih:    ((length (append t l2)) ≃ (add (length t) (length l2)))
 l1:    List · A
-l2:    List · A
-t:     List · A

==== TYPES ====
 A:     ★
U:%*-  *cedille-context-list-thms*   All (1,0)      (Fundamental Context) 3:48PM 0.70
```

Fig. 4. Screenshot showing Cedille navigation mode within emacs. The top buffer displays a Cedille source file with a proof of a simple theorem (the length of appending two lists is equal to the sum of the lengths of the lists). The middle buffer is the inspect buffer, showing information about the span which is highlighted in the top (source) buffer. The bottom buffer shows the context for the highlighted span; the context is the set of local variables with their types or kinds.

This basic architecture has so far proven extremely flexible. The backend exports essentially all available typing information as extra annotations on the spans it is sending to the frontend. So for each node in the parse tree, the user of the emacs mode can view the type of that node, and any further annotations the backend has supplied. This information is brought up in a buffer called the **inspect** buffer. Once the input file has been parsed by the backend, the emacs mode enters a special minor mode for navigation. During navigation, the buffer is read-only. Single keystrokes are then used for navigation commands and to toggle display of the inspect buffer, which is updated automatically during navigation. This makes it very quick to comb through the substantial amount of type information one gets with dependent typing. There are also keystrokes to navigate through type errors reported by the backend. The frontend recognizes which spans have errors through the inclusion of an additional attribute `error`. A screenshot showing the inspect buffer for a simple example is shown in Figure 4.

As the interface has been developed, further functionality has almost always been easily supported by simply adding more attributes to the spans. Attributes that are just used for such functionality are filtered out of the inspect buffer by default. An example is functionality to jump to the position in a (possibly different) source file where a symbol is defined. This is implemented in the frontend with the help of a `location` attribute, which the backend adds to the spans for every occurrence of a symbol. With similar additional help from the backend, to recognize which nodes are binders and which children of those nodes are the bound variables, the frontend computes the typing context at any given point in the code, and displays it in a **context** buffer. This buffer has its own minor mode allowing sorting and filtering of the context based on various parameters. A final additional buffer is a **summary** buffer, which lists all the top-level typings in a file. This is useful to see a summary of the lemmas proved in a long source file, for example. From the summary buffer, one can jump to the position in the source file for a summarized typing, with a single keystroke.

So Cedille's interface is based on the goal of providing easy navigation of all available typing information. The architecture commits to this goal by having the backend provide all this information at once as annotations to the parse tree it sends to the frontend. This is fundamentally different from the interfaces for tools like Coq and Agda, which are based on a more traditional querying style of interaction with the backend. In these tools, it is not possible to navigate through source files and see all the typing information for the subexpressions in the file. The Cedille implementation provides this additional power in a very small number of lines of code (the total line count is just over 5000 lines), thanks to this basic architectural decision to have the backend send all typing information to the frontend. The frontend then helps the user navigate and sort through the information. This architecture should be relevant for other implementations seeking to aid a user in understanding complex information computed locally for some sources. Note that similar functionality is provided by the Merlin tool for OCaml.

4 Parigot-encoded natural numbers

Having considered the user interface, let us return to the details of lambda encodings in Cedille. The Parigot encoding combines the best qualities of the Church and Scott encodings, allowing for both recursion and pattern matching. Where the Church encoding defines numbers as their own iterators, the Parigot encoding defines them as their own primitive recursors: the predecessor number is made available to the function which is applied in the successor case. We illustrate this encoding for the standard example of natural numbers. In Cedille, this is defined as follows:

```
rec Nat | S : Nat → Nat , Z : Nat =
  ∀ P : Nat → ⋆ .
    (Π n : Nat . P n → P (S n)) → P Z → P self
  with
    S = λ n . Λ P . λ s . λ z . s n (n · P s z) ,
    Z = Λ P . λ s . λ z . z.
```

The first line declares the constructors S and Z for Nat. The second and third lines define Nat as the type for lambda expressions that can verify any property P about nats, given a proof that P n implies P (S n) and a proof that P Z. In other words, it provides a way to prove statements about Nats using mathematical induction. Note that, as is often done in other languages like Coq or Agda, one could use this form of induction to derive other induction principles, like strong induction, as theorems. The fourth through sixth lines define the constructors for Nat. S is the successor, a lambda term that takes a Nat and gives back the next higher Nat; and Z is zero.

Under this definition, the first four Nats are:

```
0 = Λ P . λ s . λ z . z.
1 = Λ P . λ s . λ z . s 0 z
2 = Λ P . λ s . λ z . s 1 (s 0 z)
3 = Λ P . λ s . λ z . s 2 (s 1 (s 0 z))
```

Thus, each Nat contains within it every nat that came before it, allowing for easy retrieval of the predecessor. To accomplish this, any function f that is input for s must be of type ∀ X . Nat → (X → X), which is to say that it must begin by taking in the predecessor. Note that this means that the Church encoding can easily be recovered by simply choosing an s that discards the predecessor argument. For example, we can convert Parigot-encoded 3 to Church-encoded 3 as follows:

```
λ s' . 3 · (λ x : Nat . Nat) (λ pn . s')
 = λ s' . λ z . (λ pn . s') 2 ((λ pn . s') 1 ((λ pn . s') Z z))
 = λ s' . λ z . s' (s' (s' z))
```

To define add, we write:

```
add ⇐ Nat → Nat → Nat = λ n . λ m . θ n (λ pn . S) m .
```

where `pn` binds the predecessor of `n` (and is not used at all in the computation). As for the predecessor itself, retrieving it is simple:

```
P ⇐ Nat → Nat = λ n . θ n ( λ pn . λ _ . pn ) n .
```

5 Equational reasoning in Cedille

syntax	description
β	prove T if it is an equation whose sides are, after erasure, β-equivalent
ε `t`	if T is an equation, β-reduce both sides to head-normal form, and check `t` against this. With ε`l` instead of ε, just reduce the lhs of T (and similarly, just the rhs with ε`r`)
ρ `t - t'`	if `t` proves an equation, replace the lhs with the rhs in T, and check `t'` against this
δ `t`	if `t` proves an equality between head-normal forms with distinct head variable, then prove (any) T, as such an equality contradicts the theory of β-equality
π `n t`	if `t` proves an equality between head-normal forms with the binders $\lambda\bar{x}$ and same head variable, then prove that the `n`'th argument of the head of the lhs is equal to the corresponding argument of the rhs; where those arguments must be prefixed by the same binders $\lambda\bar{x}$.
χ `T' - t`	confirm that T and T' are convertible, and then check t against T'.
χ `- t`	synthesize a type T' for t and then confirm that T and T' are convertible.

Fig. 5. Term constructs for equational reasoning in Cedille, when checking against a type T

While it is well-known that various forms of equality can be defined in type theory, our experience so far has suggested that there are some desirable forms of reasoning which require a built-in equality type, denoted `t ≃ t'` in Cedille. The intended semantics is that the term `t` is β-equivalent to `t'`. For pure closed terms, this coincides with being $\beta\eta$-equivalent, providing a semantic justification for the use of $\beta\eta$-equivalence in Cedille.

Figure 5 summarizes some of Cedille's built-in equational reasoning principles. These are term constructs, used to prove equations or deduce facts from equations. Greek symbols are used to separate the namespace for these constructs from that for user-defined symbols. An important concept here is that of head-normal form. Recall that a head-normal form is a term of the form $\lambda\bar{x}.x_i\ \bar{t}$, where $\bar{x}$ is a nonempty sequence of variables bound at the top of the term, x_i is one of these variables, called the **head variable**, and $\bar{t}$ is a possibly empty sequence of terms given as arguments to that head variable. We found that reduction to head-normal form keeps terms more readable than reduction to normal form. Note that as mentioned above, equational reasoning is performed on erased terms, which are pure untyped lambda terms where all typing annotations have been erased.

As explained in Figure 5, β is used to prove an equation T where both sides are $\beta\eta$-equivalent. For example, the simple lemma

```
  and-tt-0 ⇐ Π x : Bool . and tt x ≃ x = λ x. β .
```

uses this construct to prove our `and` operation applied to boolean true, `tt`, and any boolean, x, is equivalent to the value of that boolean x.

The ε construct is used to β-reduce portions of an equation to head-normal form. In our Cedille programming, the typical use case we have for ε is reduction of an equation in preparation for use of ρ. Our ρ construct is applied to rewrite the left-hand side of the current equation, T, being reasoned about using its argument equation, for the purpose of stepping towards a proof of T. Examples of ε and ρ in Cedille code can be seen later in Sections 6 and 8.

Another equational principle is π, which is a kind of injectivity for head-normal forms. For a simple example, suppose we have a proof p that $\lambda c.c\ a$ equals $\lambda c.c\ b$. Then $\pi 0\ p$ would prove that $\lambda c.a$ equals $\lambda c.b$. A more realistic example of a proof using this construct is given in Section 7.

The δ construct is for the case where a proof that both sides are equal according to the built-in equality is impossible. As explained in Figure 5, this means that each side of the equation has distinct head variables while in head-normal form. As an example:

```
Bool-contra ⇐ (tt ≃ ff) → False = λ u . δ u .
```

shows δ in use. Looking back at the definitions of `tt` and `ff`, we can see the difference in head variable, as the terms are already in head-normal form. This is similar to the "absurd pattern" in Agda.

In theorem provers like Coq, there are tactics (`change`) for changing the current goal type T which the user is seeking to inhabit (i.e., prove) to some other goal type which is definitionally equal to the T. There are many reasons to want to do this, but the most critical is that while the core type theory is defined modulo definitional equality, one often has additional operations, defined via tactics or some other construct outside the core type theory, which do <u>not</u> work modulo definitional equality. These operations may be sensitive to the exact syntactic form of the goal type. In the case of Cedille, our ρ construct for rewriting is such an operation. To check a ρ-term, Cedille looks for a subexpression of the goal type T which is definitionally equal to the left-hand side of a proven equation. It may happen that a certain goal type T does not contain any such subexpression, but is convertible to a type which does. For example, consider trying to prove

```
(λ x . f (g x)) a ≃ f a
```

assuming `g a ≃ a`. The left-hand side of the goal equation is convertible to `f (g a)`, to which a rewrite with the assumed equation could be applied. But the original left-hand side (of the goal) does not contain any subterm convertible to `g a`. It is necessary first to β-reduce the left-hand side of the goal, and then such a subexpression appears. In a case like this, one can use a χ-term to change the form of the goal equation to `f (g a) ≃ f a`. The rewrite can then be applied.

6 Proving the injectivity of addition

In Section 4, we defined natural numbers (Nats) using the Parigot encoding. Now, we want to make use of this definition to prove statements about the Nats. For example, consider the following proof that the function add x is injective for all x of type Nat:

```
Add-inj ⇐ Π x : Nat . Π y : Nat . Π z : Nat .
            add x y ≃ add x z → y ≃ z =
  λ x . λ y . λ z . θ x

  % Inductive Step
    ( λ px . λ h . λ pf .
       h ( Succ-Inj ( add px y ) ( add px z ) (
        ρ ( Add-Succ-Comm-0 px y ) -
        ρ ( Add-Succ-Comm-0 px z ) -
        pf )))

  % Base Case
    ( λ pf . ρ ( Add-Ident-0 y ) -
             ρ ( Add-Ident-0 z ) - pf ).
```

We divide this theorem into a proof for the base case and a proof for the inductive step. To verify its correctness, we must ensure that the β terms in both steps are recognized as valid by Cedille. This involves making substitutions until we have an equality of the form `t ≃ t`.

The θ `x` term at the head of the λ expression tells Cedille that we are going to do a proof by induction on `x` (a "split" on `x`). Given this splitting, we are then expected to provide a proof for the inductive step followed by a proof for the base case.

Consider first the base case. The λ `pf` means that we are giving as input a proof that

```
  add Z y ≃ add Z z
```

where the substitution of `Z` for `x` is inferred from the context. The type of λ `pf` is determined using Cedille's local type inference algorithm. Cedille's user interface can display this type to the user. Given this proof, we then do a rewrite (using the ρ construct) with an instance of the `Add-Ident-0` theorem, which states that for all `n`,

```
  add Z n ≃ n
```

i.e. that zero is the identity for addition. Recall from Figure 5 that the syntax of ρ expressions is ρ `t - t'`, where `t` is a proof an equation which is used to rewrite the type for checking `t'`. By applying `Add-Ident-0` to `y`, we get a proof of `add Z y ≃ y`; similarly we apply `Add-Ident-0` to `z`. The two ρ expressions rewrite the type of `pf` using these equations, obtaining a proof that `y ≃ z`.

Now we consider the more complex inductive step proof. First, we have as inputs the following: `px`, the predecessor to `x`; `h`, our inductive hypothesis which states that `( add px y ≃ add px z ) → y ≃ z` ; and `pf`, a proof that `add (S px) y ≃ add (S px) z`. The obvious route to take here is to eliminate the `S` terms on both sides of `pf` and then give `pf` as an input to `h` to obtain `y ≃ z`. The first step is to get `S` to the outside of the `add` terms, where it will be easier to eliminate. We achieve this with the help of the `Add-Succ-Comm-0`, which proves for any Nats `n,m` that

```
add ( S n ) m ≃ S ( add n m )
```

By applying this theorem to both `px,y` and `px,z`, we transform `pf` to

```
 S ( add px y ) ≃ S ( add px z )
```

Now that `S` is on the outside of the term, we can make use of `Succ-Inj`, a proof of the injectivity of the successor function, which states that

```
( S x ≃ S y ) → x ≃ y
```

for all Nats `x,y`. By using `( add px y ),( add px z )` as our `x,y`, we obtain a proof of

```
 add px y ≃ add px z
```

We can plug this into our hypothesis `h` to get `y ≃ z` , completing our proof.

7 Using the pi construct

The example proof of the injectivity of `add x` worked by making use the ρ construct to rewrite the input proof. In this section, we showcase a different proof tool: the π construct. We want to prove that

```
Π x : Nat . Π y : Nat . (S x ≃ S y) → x ≃ y
```

in order to support our proof of `Add-inj`. The proof may be written as follows:

```
S-inj ⇐ Π x : Nat . Π y : Nat . (S x ≃ S y) → x ≃ y =
  λ x . λ y . λ pf . π1 pf .
```

Here, the purpose of `π1` is to prove that the terms at argument position 1 (where these positions start from 0) of the head-normal form of `S x ≃ S y` are equal. The terms in the resulting equality are prefixed by the same λ-abstractions as the starting head-normal form. The head-normal form of `S x ≃ S y` is

```
λ s . λ z . s x (x s z) ≃ λ s . λ z . s y (y s z)
```

So applying π1 to h produces a proof of

```
λ s . λ z . x s z ≃ λ s . λ z . y s z
```

The sides of this equation are then η-equivalent to x and y, respectively. So the type of the proof π1 h is definitionally equal to x ≃ y, as desired.

8 Examples with lists

Here is the Cedille definition for lists:

```
rec List (A : ⋆) : | Cons : A → List → List , Nil : List =
  ∀ P : List → ⋆ .
      (Π h : A . Π t : List . P t → P (Cons h t)) →
    P Nil →
    P self
  with
    Cons = λ e . λ l .  Λ P . λ c . λ n . c e l (l · P c n),
    Nil = Λ P . λ c . λ n . n .
```

Looking at this definition, we can see that operations on the list datatype will follow a specific pattern. Each function will accept arguments for the Cons and Nil cases, and then pass these functions to the list. This encoding inherently captures the foldr form familiar to many functional programmers. If we give a Cedille foldr definition:

```
foldr ⇐ ∀ A : ⋆ . ∀ B : ⋆ .
        (A → (List · A) → B → B) → B → List · A → B =
  Λ A . Λ B . λ f . λ b . λ l . θ l f b .
```

We can see that the list encoding does the work of foldr here. Below is an example of how similar the foldr and non-foldr definitions are, using the example of reversing the elements of a list:

```
singleton ⇐ ∀ A : ⋆ . A → List · A =
  Λ A . λ a . (Cons · A) a (Nil · A) .

reverseCons ⇐ ∀ A : ⋆ .
              A → (List · A) → (List · A) → (List · A) =
  Λ A . λ h . λ t . λ r . (append · A) r (singleton · A h) .

reverse ⇐ ∀ A : ⋆ . (List · A) → (List · A) =
  Λ A . λ l . θ l (reverseCons · A) (Nil · A) .

reverse2 ⇐ ∀ A : ⋆ . (List · A) → (List · A) =
  Λ A . (foldr · A · (List · A)) (reverseCons · A) (Nil · A) .
```

Typically in functional programming fold allows for smaller definitions, but the need to pass types to our polymorphic definition gives the non-foldr version a slight edge in conciseness.

For an example theorem, let us prove that the inverse of the reverse function is itself. This requires a lemma, reverse-last, but the proof of these two properties is only marginally longer than the statement of the properties themselves.

```
reverse-last ⇐
  ∀ A : ⋆ . Π l : List · A . Π a : A .
    reverse (append l (Cons a Nil)) ≃ Cons a (reverse l) =
  Λ A . λ l . λ a . θ l
    (λ h . λ t . λ ih . εl ρ ih - β)
    β .

reverse-involution ⇐
  ∀ A : ⋆ . Π l : List · A . reverse (reverse l) ≃ l =
  Λ A . λ l . θ l
    (λ h . λ t . λ ih .
       εl ρ (reverse-last · A (reverse · A t) h) - ρ ih - β)
    β .
```

The reverse-involution definition is our focus here, and we can break down this definition into two main pieces: our statement to prove and then the proof of these statements. Looking at the first and second lines of the definition, we see that it is stating that for any type, A, and for any list, l, the reverse of the reverse of a list is beta-equivalent to that list. On the third line we need parameters for the type and the list, and finally the θ l computes the predicate with which to instantiate the parameter l (as explained in Section 2 above).

Next, we have the proofs for both the `Cons` and `Nil` instances of a list. The equational steps in these two proofs are as follows:

`Cons` case:

$$\begin{aligned}\texttt{Cons h t} &= \texttt{reverse (reverse (Cons h t))}\\ &=_{\beta} \texttt{reverse (append (reverse t) (Cons h Nil))}\\ &=_{\rho\ \text{reverse-last}} \texttt{Cons h (reverse (reverse t))}\\ &=_{\rho\ \text{IH}} \texttt{Cons h t}\end{aligned}$$

`Nil` case:

$$\begin{aligned}\texttt{Nil} &= \texttt{reverse (reverse Nil)}\\ &=_{\beta} \texttt{Nil}\end{aligned}$$

This reasoning is mirrored in the Cedille code. The ϵl is used to beta-reduce the reverse side of the equation. Then we use our ρ construct to rewrite our equation using the reverse-last lemma, and again to rewrite the equation using our induction hypothesis. Our β construct then finalizes the `Cons` case and is the entire proof for the `Nil` case, stating that both sides of the equation are beta-equivalent.

9 Proving the transitivity of the comparison operator

In this section we will define a comparison operator on Nats and prove that it is transitive. This operator returns an element of the following type:

```
rec compare-t | LT : compare-t , EQ : compare-t , GT : compare-t =
  ∀ P : compare-t → ⋆ .
    P LT → P EQ → P GT → P self
  with
    LT = Λ P . λ a . λ b . λ c . a ,
    EQ = Λ P . λ a . λ b . λ c . b ,
    GT = Λ P . λ a . λ b . λ c . c .
```

Note the similarity to Church-encoded booleans; compare-t is a type which may have one of three values, and one of three distinct expressions will be evaluated depending on which of the three values is found. The definition of the `compare` function is then:

```
compare ⇐ Nat → Nat → compare-t =
  λ n . θ n
    (λ pn . λ hn . λ m . θ m (λ pm . λ _ . hn pm) GT)
    (λ m . θ m (λ _ . λ _ . LT) EQ) .
```

This function splits on the first input (`n`). If it is a successor, then `compare` checks the second input, and returns either `compare pn pm` (if that is a successor too) or `GT` (if it is zero). Otherwise, if `n ≃ Z`, `compare` checks the second input and returns either `LT` (if it is a successor) or `EQ` (if it is zero).

Suppose we want to prove a statement about `compare`. For instance, we might want to prove that `compare` is transitive. This proof can also be written through case splitting:

```
compare-trans ⇐
    Π x : Nat . Π y : Nat . Π z : Nat . Π o : compare-t .
    ( compare x y ≃ o ) → ( compare y z ≃ o ) →
    ( compare x z ≃ o ) =
  λ x . θ x
    ( λ px . λ hx . λ y . θ y
      ( λ py . λ _ . λ z . θ z
        ( λ pz . λ _ . λ o . λ pf-xy . λ pf-yz .
            ρ ( hx py pz o pf-xy pf-yz ) - β )
        ( λ _ . λ _ . λ pf-yz . ρ ( ς pf-yz ) - β ) )
      ( λ z . θ z
        ( λ _ . λ _ . λ _ . λ pf-xy . λ pf-yz .
      δ ( ρ ( ς pf-xy ) - pf-yz ) )
    ( λ _ . λ pf-xy . λ pf-yz .
      δ ( ρ ( ς pf-xy ) - pf-yz ) ) ) )
  ( λ y . θ y
   ( λ _ . λ _ . λ z . θ z
    ( λ _ . λ _ . λ _ . λ pf-xy . λ _ .
      ρ ( ς pf-xy ) - β )
    ( λ _ . λ pf-xy . λ pf-yz .
      δ ( ρ ( ς pf-xy ) - pf-yz ) ) )
   ( λ z . θ z
    ( λ _ . λ _ . λ _ . λ pf-xy . λ pf-yz .
      δ ( ρ ( ς pf-xy ) - pf-yz ) )
    ( λ _ . λ pf-xy . λ _ . pf-xy ) ) ) .
```

In this proof, there are four inputs: `x`, `y`, `z`, and a comparison type `o`. The type of the proof also provides the assumptions that `compare x y ≃ o` (labeled pf-xy), and `compare y z ≃ o` (labeled pf-yz). The body of the proof simply splits on each of the `x`, `y`, `z` variables in order for a total of 8 cases. Some of these cases follow trivially from the input proofs. For example, in the case where `x`, `y`, and `z` are all zero (the last line of the proof), we have the proof `pf-xy = compare Z Z ≃ o` (where the two zeros are `x` and `y`) and we are required to prove that `pf-xy = compare Z Z ≃ o` (where the two zeros are `x` and `z`). So we can simply regurgitate the input proof to solve the case

Other cases may be proven by contradiction using δ. For example, suppose `x` and `y` are zero, but `z` is a successor (the penultimate line of the proof). In this case, we are given proofs that

```
compare Z Z ≃ o
compare Z (S pz) ≃ o
```

and are asked to prove that `compare Z (S pz) ≃ o`. If we substitute the former into the latter, we end up with a proof of `compare Z (S pz) ≃ compare Z Z`. However, we also know by the definition of compare that `compare Z Z ≃ EQ` and `compare Z (S pz) ≃ LT`. Thus, we use δ along with this contradiction to prove anything we want (in this case, that `compare Z (S pz) ≃ o`).

The most complicated case is when all three Nats are successors. In this case, we call the inductive hypothesis

```
hx py pz o ⇐
  compare px py ≃ o → compare py pz ≃ o → compare px pz ≃ o
```

with the proofs

```
compare (S px) (S py) ≃ o
compare (S py) (S pz) ≃ o
```

Of course, as we know from the definition of compare, when both inputs are successors `compare (S a) (S b)` will return `compare a b`, so these input proofs satisfy the type requirements and thus our inductive hypothesis duly produces a proof of `compare px pz ≃ o`.

10 Reasoning about higher-order datatypes

Lambda terms can themselves be encoded as lambda terms. One of the simplest ways to accomplish this is using the Church-term (or ctrm) encoding scheme. A ctrm is a lambda term of type

```
ctrm ⇐ ⋆ = ∀ X : ⋆ .
  (X → X → X) → ((X → X) → X) → X .
```

where the first argument determines what happens when one ctrm is applied to another (App) and the second argument determines what happens when a symbol is bound to a lambda (Lam). Together, they allow for operations that depend on the structure of the term itself. To encode a term as a ctrm, we simply replace each occurrence of an application or lambda binding with the appropriate function. For example, to encode

```
λ x . λ y . x y
```

we would write

```
example ⇐ ctrm =
  Λ X . λ A . λ L . L ( λ x . L ( λ y . A x y ) ) .
```

Now suppose we want to measure the size of a term, defined by the number of nodes in its abstract syntax tree. Then we must encode two pieces of information:

1. The size of an application node is 1 plus the sum of the sizes of its child nodes.
2. The size of a lambda node is 2 (one for the lambda binder itself, and one for the binding occurrence of the variable) plus the size of the bound term, where each occurence of the bound symbol has a size of 1.

We can write a size function using this idea:

```
size-app = λ s1 : Nat . λ s2 : Nat . S (add s1 s2) .
size-lam = λ f : Nat → Nat . S ( S ( f (S Z) ) ) .
size = λ t : ctrm . t · Nat size-app size-lam .
```

Applying `size` to the above `example` term and β-reducing, we indeed get the expected value (7).

Unlike previous examples, ctrm is not declared using a `rec`-statement, so there is no way to prove statements inductively about ctrms. However, some equational reasoning about ctrms is still possible. For example, suppose we want to prove that the ctrm representations of boolean true and false (`tt` and `ff`) are not equal. We can write:

```
% tt = λ x . λ y . x
ctt ⇐ ctrm = Λ X . λ A . λ L . L (λ x . L (λ y . x)) .

% ff = λ x . λ y . y
cff ⇐ ctrm = Λ X . λ A . λ L . L (λ x . L (λ y . y)) .

ctt-cff-neq ⇐ (ctt ≃ cff) → False = λ pf . δ (π0 (π0 pf)) .
```

Here, we use `π0` twice to strip away the lambdas inside the term, leaving us with

```
λ a . λ l . λ x . λ y . x ≃ λ a . λ l . λ x . λ y . y
```

At this point, Cedille is able to recognize a contradiction, allowing us to derive `False` using a δ.

11 Conclusion and future work

This report has introduced the Cedille project, which is seeking to develop a new proof assistant and dependently typed programming language, based on lambda encodings. We have seen proofs of standard theorems, carried out here with lambda encodings instead of primitive datatypes. For future work, we intend to explore much further the power of higher-order lambda encodings for datatypes that cannot be defined in traditional type theories with primitive inductive types. Examples are datatypes for representing expressions with locally scoped names, such as object-language syntax, typing derivations, and similar structures. We have preliminary results on devising extended versions of the `ctrm` datatype above, using `rec`-defined types, to support inductive reasoning with higher-order encodings. We intend to develop this preliminary work, to take advantage of the latent higher-order power of lambda encodings. We are also working on extending these methods to other classes of datatypes including coinductive and inductive-recursive. Further future work includes continuing development of the frontend and backend tools, based on the architectural decision to export all typing information from the backend to the frontend. There are some operations, like indexing into a library, that need more back-and-forth interaction because the amount of possible information to send is too great. We intend to extend the communication protocol between frontend and backend to account for these. An important development since the work described in this paper is the derivation of induction without using `rec`, but just dependent intersections, implicit products, and equality types alone [22]. A public release of Cedille remains a little ways off, as we make what will hopefully be the final improvements to the core type theory. Furthermore, developing the tools and library to compete with well-established theorem provers like Coq and Agda will take time.

Acknowledgments. Thanks to the anonymous TFP 2016 pre-proceedings reviewer for helpful comments which helped improve the paper. This work has been supported by NSF on a grant titled "Lambda Encodings Reborn" (award

1524519), and through the DoD MURI project "Semantics, Formal Reasoning, and Tools for Quantum Programming" (award FA9550-16-1-0082). Thanks also to Matthew Heimerdinger and Richard Blair who made contributions to the tool summer 2016.

References

1. Henning Basold and Herman Geuvers. Type Theory based on Dependent Inductive and Coinductive Types. In Natarajan Shankar, editor, *IEEE Symposium on Logic in Computer Science (LICS)*, 2016.
2. Marc Bezem, Thierry Coquand, and Simon Huber. A model of type theory in cubical sets. In Ralph Matthes and Aleksy Schubert, editors, *19th International Conference on Types for Proofs and Programs, TYPES 2013*, volume 26 of *LIPIcs*, pages 107–128. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014.
3. Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theor. Comput. Sci.*, 39:135–154, 1985.
4. Alonzo Church. *The Calculi of Lambda Conversion.* Princeton University Press, 1941. Annals of Mathematics Studies, no. 6.
5. Thierry Coquand and Gerard Huet. The Calculus of Constructions. *Information and Computation*, 76(2-3):95–120, 1988.
6. Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88, International Conference on Computer Logic*, pages 50–66, 1988.
7. The Agda development team. *Agda*, 2016. Version 2.5.1.
8. Steven Fortune, Daniel Leivant, and Michael O'Donnell. The expressiveness of simple and second-order type structures. *J. ACM*, 30(1):151–185, 1983.
9. Herman Geuvers. Induction Is Not Derivable in Second Order Dependent Type Theory. In *Typed Lambda Calculi and Applications (TLCA)*, pages 166–181, 2001.
10. Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *Twenty-five years of constructive type theory*, volume 36 of *Oxford Logic Guides*, pages 83–111. Oxford University Press, 1998.
11. Pieter Koopman, Rinus Plasmeijer, and Jan Martin Jansen. Church Encoding of Data Types Considered Harmful for Implementations. In Rinus Plasmeijer and Sam Tobin-Hochstadt, editors, *26th Symposium on Implementation and Application of Functional Languages (IFL)*, 2014. Presented version.
12. Alexei Kopylov. Dependent intersection: A new way of defining records in type theory. In *18th IEEE Symposium on Logic in Computer Science (LICS)*, pages 86–95, 2003.
13. Daniel Leivant. Finitely Stratified Polymorphism. *Inf. Comput.*, 93(1):93–113, 1991.
14. Zhaohui Luo. *An Extended Calculus of Constructions.* PhD thesis, 1990.
15. The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2016. Version 8.5.
16. David A. McAllester. Implementation and abstraction in mathematics. *CoRR*, abs/1407.7274, 2014.
17. Conor McBride. Elimination with a Motive. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *Types for Proofs and Programs, International Workshop, TYPES 2000, Durham, UK, December 8-12, 2000, Selected Papers*, volume 2277 of *Lecture Notes in Computer Science*, pages 197–216. Springer, 2000.

18. Alexandre Miquel. The Implicit Calculus of Constructions Extending Pure Type Systems with an Intersection Type Binder and Subtyping. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications*, pages 344–359. 2001.
19. Michel Parigot. Programming with proofs: a second order type theory. In H. Ganzinger, editor, *European Symposium On Programming (ESOP)*, volume 300 of *Lecture Notes in Computer Science*, pages 145–159. 1988.
20. Michel Parigot. On the representation of data in lambda-calculus. In Egon Börger, HansKleine Büning, and Michael Richter, editors, *Computer Science Logic (CSL)*, volume 440 of *Lecture Notes in Computer Science*, pages 309–321. 1989.
21. Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.
22. Aaron Stump. From Realizability to Induction via Dependent Intersection. available from the author's web page, under review as of June 19, 2017.
23. Aaron Stump. The Calculus of Dependent Lambda Eliminations. *J. Funct. Program.*, 27:e14, 2017.
24. Aaron Stump and Peng Fu. Efficiency of lambda-encodings in total type theory. *Journal of Functional Programming*, 26:e3 (31 pages), 2016.
25. The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.
26. Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. *J. Funct. Program.*, 18(1):87–140, 2008.
27. Benjamin Werner. *Une Théorie des Constructions Inductives.* PhD thesis, Paris Diderot University, France, 1994.