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 introduces 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. Probably the best-known lambda encoding is the Church encoding [2], typable in System F [1, 3]. The inherent inefficiency of predecessor, proved by Parigot [10], was addressed later by the same author, who proposed a new lambda encoding with constant-time predecessor [9]. 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 [13, 5].

In dependent type theory, lambda encodings were abandoned in the 1980s, due to several serious problems: induction is provably not derivable [4], and one cannot use lambda encodings across multiple levels of the type theory (so it is not possible 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 metatheory 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 [14].

** Guneratne and Reynolds are doctoral students.

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) [12]. 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 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 equational reasoning in Cedille (Section 3). We then consider examples with the natural numbers (Section 4) and lists (Section 5), and conclude (Section 6).

2 Type checking in Cedille

The starting point for the CDLE type theory on which Cedille is based is the Calculus of Constructions, extended with support for implicit arguments (as found in other systems like the Implicit Calculus of Constructions [8]). To this, typing features are added to support induction and large eliminations with lambda encodings. We will introduce these features and informally introduce the type-checking algorithm with 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 give types for the constructors of the datatype (here `tt` and `ff`) in terms of the name of the datatype (here `Bool`). Next, a type is given which the named datatype is defined to equal. We call this the **body** of the datatype definition. In this case, the body is

```
∀ 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). 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 [6], though the Cedille implementation hides all other details for these.

Finally, the definition in Figure 1 gives definitions for the constructors `tt` and `ff`, 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 (denoted with a minus sign in Cedille's input syntax), for purposes of dependent typing. For `Bool`, they need not be mentioned at all. Cedille implements local type inference [11], 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.

```
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. The definition uses the symbol ⇐ to separate the type against which the defining term will be checked, from the symbol being defined. 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 symbol ⇐ is usually used to denote this mode. 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" [7]: 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 is currently needed for type-level conversions involving these lifting types. A basic example of such a conversion 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`.

```
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 Equational reasoning in Cedille

syntax	description
β	prove T if it is an equation whose sides are, after erasure, β-equivalent
ρ `t - t'`	if `t` proves an equation, replace the lhs with the rhs in T, and check `t'` against this
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`)
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 $\lambda\bar{x}$.

Fig. 4. Term constructs for equational reasoning in Cedille, when checking against a type T

While it is well-known that certain 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'`. Figure 4 summarizes some of Cedille's built-in equational reasoning principles. These are term constructs, used to prove equations or deduce facts from equations. 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. Also, equational reasoning is performed on erased terms, which are pure untyped lambda terms where all typing annotations have been erased. This is justified by the semantics of CDLE [12].

4 Examples with natural numbers

The definition we use for the natural numbers (Nats) in Cedille is

```
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 provide a lambda expression 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 standardly 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 (so S 1 is 2, for instance). Z is simply zero.

To see how we use this to verify theorems, 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 .
      % Proof that ( add px y ≅ add px z )
```

```
      h ( Succ-Inj ( add px y ) ( add px z ) (
       % Proof that ( S ( add px y ) ≅ S ( 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 case. 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, and is displayed via the user interface. 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 4 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.

5 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.

One of the main strengths of Cedille is that the lambda encoded types also bring this conciseness to the proofs. Below, we demonstrate this conciseness by proving 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 is equivalent to applying the predicate to the parameter l.

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:

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

Nil case:

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

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.

6 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 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. Further future work includes standard engineering improvements for interactive theorem provers like support for automated theorem proving, and improving the user interface to reduce the burden of proof for users.

References

1. Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theor. Comput. Sci.*, 39:135–154, 1985.
2. Alonzo Church. *The Calculi of Lambda Conversion.* Princeton University Press, 1941. Annals of Mathematics Studies, no. 6.
3. Steven Fortune, Daniel Leivant, and Michael O'Donnell. The expressiveness of simple and second-order type structures. *J. ACM*, 30(1):151–185, 1983.
4. Herman Geuvers. Induction Is Not Derivable in Second Order Dependent Type Theory. In *Typed Lambda Calculi and Applications (TLCA)*, pages 166–181, 2001.
5. 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.

6. 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.
7. 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.
8. 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.
9. 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.
10. 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.
11. Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.
12. Aaron Stump. The Calculus of Dependent Lambda Eliminations, 2016. Under review, draft paper available from the author's homepage.
13. Aaron Stump and Peng Fu. Efficiency of lambda-encodings in total type theory. *Journal of Functional Programming*, 26:e3 (31 pages), 2016.
14. Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. *J. Funct. Program.*, 18(1):87–140, 2008.