A Weakly Initial Algebra for Higher-Order Abstract Syntax in Cedille

Aaron Stump
Computer Science
The University of Iowa
Iowa City, Iowa, USA
`aaron-stump@uiowa.edu`

Cedille is a relatively recent tool based on a Curry-style pure type theory, without a primitive datatype system. Using novel techniques based on dependent intersection types, inductive datatypes with their induction principles are derived. One benefit of this approach is that it allows exploration of new or advanced forms of inductive datatypes. This paper reports work in progress on one such form, namely higher-order abstract syntax (HOAS). We consider the nature of HOAS in the setting of pure type theory, comparing with the traditional concept of environment models for lambda calculus. We see an alternative, based on what we term Kripke function-spaces, for which we can derive a weakly initial algebra in Cedille. Several examples are given using the encoding.

1 Introduction

Modern constructive type theory is based on a decades-long development of formal systems, culminating in current tools like Coq and Agda, to name two of the most widely used [32, 31]. To summarize the relevant history: in the 1980s Coquand and Huet proposed the Calculus of Constructions (CC) as a synthesis of impredicative type theory as independently proposed by Girard and Reynolds [9, 22], and dependent type theory as found in de Bruijn's Automath and further developed by Martin-Löf [3, 13]. What was initially believed by researchers working on CC was confirmed in the early 2000s by Geuvers: induction is not derivable in CC (although note that technically, Geuvers's theorem is about just the second-order fragment of CC) [8]. So in the late 1980s and early 1990s, researchers explored various ways of adding primitive inductive datatypes to CC [18, 20]. At the same time, Luo analyzed an extension of CC with an ω-indexed predicative hierarchy of universes [12], still found in Coq today. A practically viable solution to the problem of inductive datatypes was reached in Werner's development of the Calculus of Inductive Constructions (CIC), which added a specific class of inductive datatypes to CC (note that the predicative hierarchy is not included in CIC as analyzed by Werner) [36]. Subsequent work on the theory and practice of Coq has built upon these results, resulting in a tool that is both widely used and rightly generally considered a great success.

Despite these excellent achievements, there are two notable issues with CIC's solution to the problem of datatypes in type theory:

1. The class of datatypes is fixed as part of the definition of the theory.
2. The core theory upon which the complex edifice of the rest of the proof assistant is built must include support for that class of inductive datatypes, as they are primitive to the theory.

(1) is an issue because it means that subsequent discoveries and proposals for advanced forms of datatypes are excluded from CIC. One would have to rework the entire metatheory of CIC to add them. Or one could adopt the approach taken in Agda, which is to extend the datatype system without requiring full

metatheoretic justification. While this facilitates exploration of advanced forms of datatypes, it comes at the risk of introducing inconsistency into the theory (through a novel form of datatype that would turn out to be logically unsound). (2) is an issue because it means that the trusted computing base of a tool like Coq is rather large. At present, for example, the kernel of Coq – the internal code which one must trust when the type-checker accepts a theorem (this does not count parsers and printers; cf. [37]) – is just over 30k lines of OCaml. This includes powerful features like byte-code compilation for faster conversion-checking, which could be excluded from the line count just for core typing; but even the files for inductive types (`indtypes.ml` and `inductive.ml`) total just under 2200 lines (see `https://github.com/coq`). It would be very nice to have a core checker under, say, 1000 lines of functional code.

Cedille is a recently released proof assistant based on a novel minimalistic extension of CC, which allows derivation of inductive datatypes with their induction principles. So the core theory does not include a primitive notion of inductive datatype, and indeed can be checked in under 1000 lines of Haskell [29]. Cedille is briefly described in Section 2. The focus of the current paper is on work in progress deriving an advanced form of datatype in Cedille, namely higher-order abstract syntax (HOAS) [19]. Section 3 discusses what HOAS should be taken to mean in the context of pure lambda calculus (where every term is encoded functionally), considering (and rejecting) the traditional environment models for algebraic semantics of lambda calculus. Section 4 presents an alternative implemented in Cedille, for which we have a weakly initial algebra. This approach uses what we term *Kripke function spaces* to allow construction of an encoded nested λ-abstraction. It turns out that for what has been achieved so far, the full power of Cedille is not needed, and the code can also be written in Haskell with a few language extensions (Section 5). Section 8 discusses a possible way to extend this to obtain induction, based on parametricity.

2 Cedille and its Type Theory

We briefly summarize the type theory of Cedille, called the Calculus of Lambda Eliminations (CDLE). The system has evolved from an initial version [26], to its current form [28]. Several other works demonstrate applications of the theory to derivation of inductive datatypes [6, 7, 27], and to zero-cost coercions between related datatypes [5]. The main metatheoretic property proved in previous work is logical consistency: there are types which are not inhabited. All the code appearing in this paper can be checked using Cedille 1.0. (Cedille 1.1 adds datatypes which elaborate down to the pure type theory of CDLE, but we do not make use of this feature here.)

CDLE is an extrinsic (i.e. Curry-style) type theory, whose terms are exactly those of the pure untyped lambda calculus (with no additional constants or constructs). The type-assignment system for CDLE is not subject-directed, and thus cannot be used directly as a typing algorithm. Indeed, since CDLE includes Curry-style System F as a subsystem, type assignment is undecidable [35]. To obtain a usable type theory, Cedille combines bidirectional checking [21] with a system of annotations for terms, to obtain algorithmic typing. But true to the extrinsic nature of the theory, these annotations play no computational role, and are erased both during compilation and before formal reasoning about terms within the type theory, in particular by definitional equality. We summarize the central rules and clauses of the erasure function in Figure 1 and following text. As this is, by necessity of space, quite brief, please see a report for full details, including semantics and soundness results [28].

CDLE extends the (Curry-style) Calculus of Constructions (CC) with a primitive intensional untyped equality, intersection types, and implicit products (in the following explanation we use `tt` fonts to introduce the concrete syntax, very close to the mathematical one, expected by Cedille):

$$\frac{\Gamma, x : T' \vdash t \Leftarrow T \quad x \notin FV(|t|)}{\Gamma \vdash \Lambda x.t \Leftarrow \forall x{:}T'.T} \qquad \frac{\Gamma \vdash t \Rightarrow \forall x{:}T'.T \quad \Gamma \vdash t' \Leftarrow T'}{\Gamma \vdash t \text{ -}t' \Rightarrow [t'/x]T}$$

$$\frac{\Gamma \vdash FV(t) \subseteq dom(\Gamma)}{\Gamma \vdash \beta\{t'\} \Leftarrow \{t \simeq t\}} \qquad \frac{\Gamma \vdash t' \Rightarrow t_1 \simeq t_2 \quad \Gamma \vdash t \Leftrightarrow [t_1/x]T}{\Gamma \vdash \rho\ t' \text{ - } t \Rightarrow [t_2/x]T}$$

$$\frac{\Gamma \vdash t \Leftarrow T \quad \Gamma \vdash t' \Leftarrow [t/x]T' \quad |t| =_{\beta\eta} |t'|}{\Gamma \vdash [t,t'] \Leftarrow \iota x{:}T.T'} \qquad \frac{\Gamma \vdash t \Rightarrow \iota x{:}T.T'}{\Gamma \vdash t.1 \Rightarrow T}$$

$$\frac{\Gamma \vdash t \Rightarrow \iota x{:}T.T'}{\Gamma \vdash t.2 \Rightarrow [t.1/x]T'} \qquad \frac{\Gamma \vdash T \Leftarrow \star \quad \Gamma \vdash t \Leftarrow T \quad T \cong T'}{\Gamma \vdash \chi\ T \text{ - } t \Leftarrow T'}$$

$$\frac{\Gamma \vdash T \Leftarrow \star \quad \Gamma \vdash t \Rightarrow T' \quad T \cong T'}{\Gamma \vdash \chi\ T \text{ - } t \Rightarrow T} \qquad \frac{\Gamma \vdash t \Rightarrow \{t' \simeq t''\} \quad \Gamma \vdash t' \Leftrightarrow T}{\Gamma \vdash \phi\ t \text{ - } t'\{t''\} \Leftrightarrow T}$$

$$\begin{array}{lcl}
|\Lambda x{:}T.t| & = & |t| \\
|t \text{ -}t'| & = & |t| \\
|\beta\{t\}| & = & |t| \\
|\rho\ t' \text{ - } t| & = & |t| \\
|\phi\ q \text{ - } t_1\{t_2\}| & = & |t_2| \\
|[t_1,t_2]| & = & |t_1| \\
|t.1| & = & |t| \\
|t.2| & = & |t|
\end{array}$$

Figure 1: Introduction, elimination, and erasure rules for additional type constructs. Note that $\Leftarrow$ is for checking mode, $\Rightarrow$ is for synthesizing, and $\Leftrightarrow$ refers to either mode.

- $\{\ t_1 \simeq t_2\ \}$, an intensional equality type between terms t_1 and t_2 which need not be typable at all. We introduce this with a constant β`{t}` which erases to erasure of `t` (so our type-assignment system has no additional constants, as promised); β`{t}` proves `{ t' ≃ t' }` for any term `t'` with free variables all in scope. Combined with definitional equality, β`{t}` proves $\{\ t_1 \simeq t_2\ \}$ for any $\beta\eta$-equal t_1 and t_2 whose free variables are all declared in the typing context. If the term `t` is omitted from β`{t}`, then it is assumed to be λ `x. x`. We eliminate the equality type by rewriting, with a construct ρ `t' - t`. Suppose `t'` proves $\{\ t_1 \simeq t_2\ \}$ and we are checking the ρ-term against a type `T`, where `T` has several occurrences of terms definitionally equal to t_1. Then bidirectional typing proceeds by checking `t` against type `T` except with those occurrences replaced by t_2. We also adopt a strong form of Nuprl's **direct computation rules** [4]: if we have a term t' of type T and a proof t that $\{t' \simeq t''\}$, then we may conclude that t'' has type T by writing the annotated term $\phi\ t$ - $t'\{t''\}$, which erases to t''.

- ι `x : T. T'`, the dependent intersection type of Kopylov [11]. This is the type for terms `t` which can be assigned both the type `T` and the type `[t/x]T'`, the substitution instance of `T'` by `t`. There are constructs `t.1` and `t.2` to select either the `T` or `[t.1/x]T'` view of a term `t` of type ι `x : T. T'`. We introduce a value of ι `x : T. T'` by construct $[t_1,\ t_2]$, where t_1 has type `T`, t_2 has type $[t_1/x]T'$, and t_1 and t_2 must have the same erasure (as the intersection type is intended as to represent two typings of the same underlying erased term).

- $\forall$ `x : T. T'`, the implicit product type of Miquel [16]. This can be thought of as the type for functions which accept an erased input of type `x : T`, and produce a result of type `T'`. There are term constructs Λ `x. t` for introducing an implicit input `x`, and `t -t'` for instantiating such an input with `t'`. This use of a dash in the notation should not be confused with the uses of dash in the notations for ρ and ϕ terms, where it is just punctuation intended to help separate subexpressions. The implicit arguments exist just for purposes of typing so that they play no computational role and equational reasoning happens on terms from which the implicit arguments have been erased. Note that similar notation is used for quantifications $\forall X : \kappa.T$ over types (more generally, type constructors), although we use notation $t \cdot T$ instead of t-T to indicate instantiating the quantified type of t with type T (that is, for $\forall$-elimination). These notations bind tighter than function space. If variable x is not free in T', we write just $T \Rightarrow T'$ for $\forall x{:}T.T'$.

3 HOAS and semantics

The well-known central idea of higher-order abstract syntax (HOAS) is to encode object-language binders, like λ in untyped λ-calculus, with meta-language binders. In a pure type theory, without introduction of special constructs explicitly for representation of binders (as in [15]), but rather using only λ-abstractions, some puzzles arise:

1. In pure type theory, all data must be λ-encoded (e.g., Church-encoded), and hence object-language binders would seem automatically to be transformed to λ-abstractions, since all data are. So it is not clear what could distinguish HOAS from a first-order approach to encoding binders.
2. Using λ-abstractions to encode object-language binders appears too strong, as the set of functions even under a strong typing discipline will be much larger than the set of weak functions intended to represent the bodies of object-language abstractions.

Washburn and Weirich proposed a solution to (2): use parametric polymorphism to ensure that, for example, the functions intended to represent bodies of object-language abstractions cannot pattern-match on their inputs (which would not correspond to any object-language abstraction under the usual approach to binding syntax) [34]. They connect their approach to an earlier work of Schürmann et al., which used modal types to enable similarly restricting the function space [24]. We will adopt Washburn and Weirich's idea below (Section 4), though a twist is required to obtain a (weakly) initial algebra.

For (1), we may compare with the traditional approach to algebraic semantics of λ-calculus (as object language), based on what are sometimes called environment λ-models (see Definition 15.3 of [10], and cf. [25]). Such a model is a structure $\langle D, \bullet, [\![-]\!]_{-} \rangle$, where D is a set of cardinality at least two, consisting of some mathematical objects to be the interpretations of λ-terms; $\bullet$ is a binary operation on D intended to model application; and $[\![-]\!]_{-}$ is an interpretation function mapping (object-language) terms t and valuations $\rho \in \mathit{Vars} \to D$ to D. The interpretation function is required to satisfy various conditions, which suffice to ensure that the usual equational theory $\lambda\beta$ of λ-calculus is sound with respect to $[\![-]\!]_{-}$: if $\vdash t =_{\beta} t'$, then $[\![t]\!]_{\rho} = [\![t']\!]_{\rho}$ for any valuation ρ. One of these conditions, central to soundness of the β axiom (scheme), is that semantic application of the interpretation of a λ-abstraction must be the same as evaluating the body with an updated environment: $[\![\lambda\, x.t]\!]_{\rho} \bullet d = [\![t]\!]_{\rho[x \mapsto d]}$.

If we are looking to universal algebra for ideas on λ-encoding HOAS – as indeed it is profitable to do for encoding first-order datatypes (see [33] for a tutorial, or previous work using Cedille like [7]) – we will be misled at this point. For environment models presuppose a first-order approach to syntax, so that they can model instantiation of a λ-bound variable by environment update. And here, even if we functionally encode valuations, variables, and terms, we will have not achieved anything beyond usual first-order representations of terms. To λ-encode HOAS, we need a new approach to the semantics of λ-calculus that does not use environments.

Categorically, given a endofunctor F on a category $\mathscr{C}$, it is standard to consider the category of F-algebras whose objects are as $\mathscr{C}$-morphisms from $F\ A$ to A for $\mathscr{C}$-objects A (the *carrier* of the algebra), and whose morphisms are $\mathscr{C}$-morphisms h from A to B that form a commuting square (in $\mathscr{C}$) with the $F\ A$ to A morphisms, and an $F\ A$ to $F\ B$ morphism derived from h. An initial algebra is then an initial object in this category, for which various appealing properties can be proved, in particular that its carrier

C is the least carrier isomorphic to $F\ C$. From such developments induction principles are then readily derived. The difficulty with HOAS is that the type scheme F one wishes to use is not a functor, due to a negative occurrence of X in $F\ X$.

4 An encoding of lambda-terms in Cedille

The basic Church-encoding of inductive types can be carried out in a type theory like Cedille's, following the categorical perspective. Given a functorial type scheme F, define (within the type theory) the type $Alg \cdot A$ for algebras over type A as $F\ A \to A$ (recall that in Cedille we use center dot for applying an expression to a type). Then the carrier C of a weakly initial algebra has type $\forall A\!:\!\star.\,(F \cdot A \to A) \to A$. In the following discussion, let us write C_A for the type $(F \cdot A \to A) \to A$. As an example of the definition of C: if F is the functor for the type of natural numbers (and allowing ourselves infix notation for sum and later product types, and 1 for unit type), we obtain the type $\forall A\!:\!\star.\,(1 + A \to A) \to A$ (let us abbreviate this *Nat*), which is isomorphic to the usual type $\forall A\!:\!\star.A \to (A \to A) \to A$ for Church-encoded natural numbers. The main effort is then to define the algebra itself (not just its carrier), which in general must have type $Alg \cdot C$. In the case of *Nat*, we need something of type $Alg \cdot Nat$, which is easily obtained: from $1 + Nat$ return Church-encoded zero in the first case, and Church-encoded successor of the given *Nat* in the second.

4.1 Starting from Washburn and Weirich

The approach by Washburn and Weirich, which is not (directly) based on this perspective, does not allow definition of this algebra. Their separate definitions of constructor for object-language λ-abstractions and applications can be seen in our terms as constituting, for the functor F for λ-terms (which is $\lambda\, X\!:\!\star.\,(X \to X) + (X \times X)$), a function of type $\forall A\!:\!\star.\,F \cdot C_A \to C_A$. But this is not the type needed for the weakly initial algebra, which instead should be $\forall A\!:\!\star.\,F \cdot C \to C$. Without a definition of a weakly initial algebra, there is no hope, on the categorical perspective, to define an initial algebra with induction principle (nor is this claimed in [34]).

But we may still make use of the basic insight of Washburn and Weirich that parametricity can be used to restrict the function spaces intended to represent bodies of object-language abstractions. To simplify the discussion (and Cedille code), we consider from here on a reduced syntax of λ-terms that omits applications. So one may only form terms of the form $\lambda\, x_1. \cdots \lambda\, x_n.y$ (and closed terms require $y \in \{x_1, \ldots, x_n\}$). This reduced syntax focuses attention on binding and variable occurrences; adding applications back in should be completely straightforward.

To return to parametricity: what should be the type of a function `lam` constructing the encoding of an object-language λ-abstraction? The more fundamental question is, what should the form *Alg* of algebras be, which will allow construction of a weakly initial algebra $Alg \cdot Trm$, where *Trm* is the desired carrier for encodings of λ-terms (without applications)? It is almost immediately clear that we cannot use the same notion of algebra as for the Church encoding. The type scheme F (it is not a functor) in question is simply $X \to X$, and thus to inhabit $Alg \cdot Trm$ we would have to construct a (meta-language) term of type $(Trm \to Trm) \to Trm$ (corresponding to $F \cdot C \to C$ in our general discussion above), and this seems to be impossible.

Drawing inspiration from Selinger's idea of adjoining indeterminates to an algebra to represent free variables [25], let us think of a binder as introducing a new constructor for the *Trm* datatype. So an algebra should be given, for an encoded lambda abstraction, not just a subterm for the body, but rather a subterm possibly using a new constructor. We use parametric polymorphism to enforce that this binder is abstract. So we would like to give our X-algebras a function f of type $\forall Y : \star. Y \to Trm_Y$, and obtain from the algebra then a value of type X. Note that this requires some form of recursive type so that the type for algebras for *Trm* can reference *Trm*. As will be described in a future work (but see also [6]), these are derivable in Cedille. We elide calls to fold and unfold these in the following. The (candidate) weakly initial algebra would then have type

$$(\forall Y : \star. Y \to Trm_Y) \to Trm \tag{1}$$

But there is a problem with this definition. A requirement we should impose for the encoding of any datatype is that elements of the datatype can be built up by successive applications of the constructors of the datatype (as 3 can be built by three applications of the successor constructor to zero). But if we use Type 1, we will not be able to represent object-language λ-terms like $\lambda\, x. \lambda\, y. x$. For Type 1 requires that the body of the abstraction construct a Trm_Y from a Y, where Y is abstract. So the representation of $\lambda\, y. x$ is not well-typed, because x has some first abstract type Y, while y has a second Z, and the body requires a Trm_Z. There is no way to convert x of type Y to Z to embed in a Trm_Z.

4.2 A solution using Kripke function spaces

Seen as just considered, we need a way to embed the type of some outer encoded binder into the types of inner ones. This is quite reminiscent of the Kripke semantics for intuitionistic logic, where implication is interpreted as a modal operator: for $T \to T'$ to be true at the current world w, it must be the case that for all future worlds w' where T holds, T' also holds. An X-algebra needs the ability to move the body of the encoded λ-abstraction to any world reachable from X. To make the structure of the positive-recursive type more clear, let us first define a notion like C_A above, but where the notion of algebra is also a parameter:

$$Trmga = \lambda\, Alg : \star \to \star. \lambda\, X : \star. Alg \cdot X \to X$$

We may then give the following positive-recursive definition of algebra:

$$Alg = (\forall Y : \star. (X \to Y) \to Y \to Trmga \cdot Alg \cdot Y) \to X \tag{2}$$

What we are terming *Kripke function space* rooted at X is a type of the form $\forall Y : \star. (X \to Y) \to T$. It is the type for functions that can be moved to any type Y reachable from X.

This is not the final definition of algebra, though, because as formulated so far, there is no support for iteration. So the encoding would be more like a Scott encoding than a Church encoding (see [30] for a comparison). To support iteration, the algebra must be given a way to evaluate the value of type $Trmga \cdot Alg \cdot Y$ returned by its input. For this, we use Mendler's technique of polymorphically abstracting problematic type occurrences, to allow an algebra to take in a type-abstracted version of itself [14].

$$\begin{array}{rcl} Alg & = & \forall Alga : \star \to \star. (\forall Y : \star. (X \to Y) \to Y \to Trmga \cdot Alga \cdot Y) \\ & & Alga \cdot X \to \\ & & (Cast2 \cdot Alg \cdot Alga) \Rightarrow \\ & & X \end{array} \tag{3}$$

Here, we have introduced a universal quantification over the type *Alga* of algebras (one may think of these as *algebra candidates*, similar to Girard's reducibility candidates). This allows an algebra to be given an input of type $Alga \cdot X$; with just $Alg \cdot X$ this would not be possible as it occurs at a negative position in the recursive definition of *Alg*. The final input to an algebra is a second-order cast from *Alg* to *Alga*. Eliding the details, this allows us to embed any $Alg \cdot X$ to an $Alga \cdot X$. This provides the critical ability for an algebra to interpret encoded terms it is given, possibly using a different algebra.

Based on this final notion of algebra, we define:

$$Trm \;=\; \forall X\!:\!\star.\, Trmga \cdot Alg \cdot X.$$

Evaluation of a term using an algebra is then trivial; terms are functions from algebras to carriers, and so we just apply the term (*t* below) to the algebra (*alg*):

$$fold \;:\; \forall X\!:\!\star.\, Alg \cdot X \to Trm \to X \;=\; \Lambda X.\, \lambda\, alg.\, \lambda\, t.\, t\ alg.$$

More interestingly, we may now define the following algebra with carrier *Trm*, which we will prove below (Section 7) is weakly initial:

$$\begin{array}{lcl} lamAlg : Alg \cdot Trm & = & \Lambda Alga.\, \lambda\, f.\, \Lambda emb.\, \lambda\, talg. \\ & & \Lambda X.\, \lambda\, alg.\, alg \cdot Alga\ (\Lambda Y.\, \lambda\, mx.\, f \cdot Y\ (\lambda\, t.\, mx\ (t\ alg)))\ \text{-}emb\ (cast2\ \text{-}emb\ alg). \end{array}$$

All the components discussed above are required here. We use the ability to change algebras to invoke *alg* at abstract type *Alga*, and to make use of *alg* rather than *talg*. We can notice that *talg* is not even used (note that in the application *mx* (*t alg*), we have *t* applied to *alg*, not *talg*). So rather than recursing through the body of the encoded λ-abstraction as given by f using the algebra which is being given to *lamAlg*, *lamAlg* instead switches algebras to use the one being given to the *Trm* which it (*lamAlg*) is being asked to produce. A cast changes the type of *alg* to the instance $Alga \cdot X$ of the abstracted algebra.

For use in nested construction of terms, the following variant of *lamAlg* is needed:

$$\begin{array}{l} lam : \forall X\!:\!\star.\, (\forall Y\!:\!\star.\, (X \to Y) \to Y \to Trmga \cdot Alg \cdot Y) \to Trmga \cdot Alg \cdot X \\ \quad = \Lambda X.\, \lambda\, f.\, \lambda\, alg.\, alg \cdot Alg\ f\ \text{-}(castId2 \cdot Alg)\ alg \end{array}$$

The difference from *lamAlg* is that here the Kripke function space is rooted at any type X, where *lamAlg* is rooted at *Trm*. Quantifying over the root of the Kripke function space allows nested applications of *lam*, as in the encoding of the second-projection function (first defining a convenience function *place*):

$$\begin{array}{lcl} place : \forall X\!:\!\star.\, X \to Trmga \cdot Alg \cdot X & = & \Lambda X.\, \lambda\, x.\, \lambda\, alg x. \\ & & \\ proj2 : Trm & = & \Lambda O.\, lam\ (\Lambda X.\, \lambda\, mo.\, \lambda\, x. \\ & & \qquad lam\ (\Lambda Y.\, \lambda\, mx.\, \lambda\, y.\, place\ (mx\, x))) \end{array}$$

Notice how the outer meta-language bound variable *x* is used inside the (meta-language) binding of *y*, using *mx* to move it from X to Y.

The inspiration of Kripke semantics for semantics of lambda calculus may also be found in works like Mitchell and Moggi's [17]. There, explicit environments are used to interpret terms, and so the semantics fails to be a suitable basis for a higher-order encoding, for the reasons discussed above.

5 Haskell listing

The above development actually does not make use of the special features of Cedille beyond (derivable) positive-recursive types. In fact, it can be carried out in any language supporting impredicative

```
module WeaklyInitialHoas where

type Trmga alg x = alg x -> x

newtype Alg x =
  MkAlg { unfoldAlg :: forall (alga :: * -> *) .
                        (forall (y :: *) . (x -> y) -> y -> Trmga alga y) ->
                        (forall (z :: *) . Alg z -> alga z) ->
                        alga x -> x}
newtype Trm = MkTrm { unfoldTrm :: forall (x :: *) . Alg x -> x}

fold :: Alg a -> Trm -> a
fold alg t = unfoldTrm t alg

lamAlg :: Alg Trm
lamAlg = MkAlg (\ f embed talg ->
           MkTrm (\ alg ->
             unfoldAlg alg (\ mx -> f (\ t -> mx (unfoldTrm t alg)))
               embed (embed alg)))

lam :: forall (x :: *) .
       (forall (y :: *) . (x -> y) -> y -> Trmga Alg y) -> Trmga Alg x
lam = \ f alg -> unfoldAlg alg f (\ x -> x) alg

place :: forall (x :: *) . x -> Trmga Alg x
place = \ x -> \ alg -> x
```

Figure 2: Haskell definitions for the Cedille code above

quantification and positive recursive types, such as Haskell (impredicativity has to be mediated by inductive datatypes in a certain way, but is essentially present). To aid the reader more familiar with Haskell than Cedille, Figure 2 gives a Haskell listing of the functions discussed above. This requires Haskell `LANGUAGE` extensions `KindSignatures`, `ExplicitForAll`, and `RankNTypes`. Some uses of implicit function space in the Cedille code have been converted to the regular (explicit) function spaces of Haskell. Impressively, Haskell's type inference is powerful enough to allow us to avoid type annotations except for marking the places where universal generalization (with constructors `MkAlg` and `Trm`) and instantiation (with eliminators `unfoldAlg` and `unfoldTrm`) occur.

6 Examples

Based on the Haskell implementation of Figure 2, let us consider several examples of algebras. For testing, we will use the following simple term, representing $\lambda x.\lambda y.x$:

```
test :: Trm
test = MkTrm (lam (\ mo x ->
                lam (\ mx y -> place (mx x))))
```

```
sizeAlg :: Num a => Alg a
sizeAlg = MkAlg (\ f embed alg -> 1 + f id 1 alg)
```

Figure 3: An algebra for the size of a term

```
vars :: Int -> [String]
vars n = ("x" ++ show n) : vars (n + 1)

printTrmAlg :: Alg ([String] -> String)
printTrmAlg =
  MkAlg (\ f embed alg vars ->
           let x = head vars in
               "\\ " ++ x ++ ". " ++ f id (\ vars -> x) alg (tail vars))

printTrm :: Trm -> String
printTrm t = fold printTrmAlg t (vars 1)
```

Figure 4: Algebra and related functions for converting a term to a string

6.1 Size

Figure 3 gives an algebra for computing the size of a term. The algebra is given the body `f`, the embedding `embed` from `Alg` to abstract type `Alga` (which is not needed for this example), and the algebra itself under the abstract type. Interpreting a λ-abstraction (as is done by this and all algebras) is done by interpreting the body using `alg`, where `1` is given as the value to use for the bound variable. The use of the `id` (identity function in Haskell) is to map trivially from the carrier of the algebra to the type at which we are interpreting the body, namely also the carrier.
Interpreting our test term with `sizeAlg` gives us the following interaction using `ghci`:

```
*WeaklyInitialHoas> fold sizeAlg test
3
```

This is as expected, size we count one for each λ and then one for the use of the variable x.

6.2 Converting to strings

Figure 4 defines an algebra `printTrmAlg` for use in converting a term to a string. The carrier of the algebra is `[String] -> String`; a term is interpreted as a function from a stream of variable names (the `[String]`) to a `String` representation of the term. The algebra simply peels off the first name (`x` in the code) from the stream and uses it for the binding occurrence of the variable. For any bound occurrences in the body `f`, the algebra passes to `f` the function `\ vars -> x` as the interpretation for the variable. This function (of type `[String] -> String`) simply discards the stream of names it is given and returns `x` as the interpretation (i.e., the string representation) of the bound variable.

For `printTerm`, we fold the algebra over the input term, and then apply the resulting function to the simple stream of variable names `vars 1`. For our test term, we can observe the following result with `ghci`:

```
*WeaklyInitialHoas> putStrLn (printTrm test)
\ x1. \ x2. x1
```

```
data Dbtrm = Lam Dbtrm | Var Int deriving Show

toDebruijnAlg :: Alg (Int -> Dbtrm)
toDebruijnAlg = MkAlg (\ f embed alg -> \ v ->
                    let v' = v + 1 in Lam (f id (\ n -> Var (n - v')) alg v'))
```

Figure 5: An algebra for converting to de Bruijn notation

```
IsHom : Π X1 : ⋆ . (Alg · X1) ➔
        Π X2 : ⋆ . (Alg · X2) ➔
        Π h : X1 ➔ X2 . ⋆ =
  λ X1 : ⋆ . λ alg1 : Alg · X1 .
  λ X2 : ⋆ . λ alg2 : Alg · X2 .
  λ h : X1 ➔ X2 .
    ∀ Alga : ⋆ ➔ ⋆ .
    ∀ f : ∀ Y : ⋆ . (X1 ➔ Y) ➔ Y ➔ Trmga · Alga · Y .
    ∀ c : Cast2 · Alg · Alga .
    { h (alg1 f alg1) ≃ alg2 (λ mx . f (λ a . mx (h a))) alg2 }.
```

Figure 6: Definition of homomorphism

6.3 Converting to de Bruijn notation

Figure 5 defines a datatype `Dbtrm` for untyped λ-terms in de Bruijn notation (without application, similarly to our running example). The figure also defines an algebra for converting a term to a `Dbtrm`. More precisely, the carrier of the algebra is `Int -> Dbtrm`; the algebra converts a term to a function which takes in the number `v` to use as the current depth of nesting within λ-abstractions. The algebra interprets the body with the successor nesting depth `v'`. It supplies the function `\ n -> Var (n - v')` for the interpretation of the bound variable. This function takes in the current depth `n` and subtracts off `v'`, which is one plus the depth at which the binding occurrence of the variable was encountered (subtracting one ensures that the starting de Bruijn index is zero). For the test term, we confirm the expected result with `ghci`:

```
*WeaklyInitialHoas> fold toDebruijnAlg test 1
Lam (Lam (Var 1))
```

7 Weak initiality of lamAlg

We would now like to consider the above development from a categorical perspective, as is standard for simpler classes of inductive datatypes like those arising from polynomial functors (see [2] for a summary in service of functional programming). Given two algebras `alg1` and `alg2` with carriers `X1` and `X2`, we must first define what it means for a function `h : X1 ➔ X2` to be a homomorphism from the first algebra to the second. The definition is given, in Cedille notation, in Figure 6. It states that such an `h` is a homomorphism iff for all components required by `alg1` – that is, for all algebra candidates `Alga`, bodies `f`, and embeddings `c` – the following equation holds:

```
{ h (alg1 f alg1) ≃ alg2 (λ mx . f (λ a . mx (h a))) alg2 }
```

```
IdHom : ∀ X : ⋆ . ∀ alg : Alg · X .
        IsHom · X alg · X alg (λ x . x)

ComposeHom : ∀ X1 : ⋆ . ∀ alg1 : Alg · X1 .
             ∀ X2 : ⋆ . ∀ alg2 : Alg · X2 .
             ∀ X3 : ⋆ . ∀ alg3 : Alg · X3 .
             ∀ h1 : X1 → X2 .
             ∀ h2 : X2 → X3 .
             IsHom · X1 alg1 · X2 alg2 h1 →
             IsHom · X2 alg2 · X3 alg3 h2 →
             IsHom · X1 alg1 · X3 alg3 (λ x . h2 (h1 x))

foldHom : ∀ X : ⋆ . ∀ alg : Alg · X .
IsHom · Trm lamAlg · X alg (fold alg)
```

Figure 7: Algebras form a category with `lamAlg` as a weakly initial object

This is an adaptation of the usual commutation condition one desires for homomorphisms. It says that applying the homomorphism and then `alg1` (to `f`) is the same as applying `alg2` to a modified version of `f`, which applies `h` internally.

Using this definition of homomorphism, we can prove (in Cedille) the theorems shown in Figure 7. The first says that `λ x . x` is a homomorphism from any algebra to itself. The second states that homomorphisms compose. The third is the main result of the paper. It states that `lamAlg` (defined in Section 4.2 above) is a weakly initial algebra: for any algebra `alg`, the function `fold alg` is a homomorphism from `lamAlg` to `alg`. These theorems have short simple proofs, as one would anticipate.

8 Conclusion

In this paper, we have seen how to derive a weakly initial algebra for a very simple datatype using higher-order abstract syntax. The crucial next step of this work in progress is to extend the development to derive an initial (not just weakly initial) algebra, for the *Trm* datatype. The strategy I am following for this is to form a dependent intersection of *Trm* as defined above with a statement of unary parametricity [23]. It should be possible to do this for any type (and hence for *Trm*), as studied by Bernardy and Lasson [1]). And with a reflection principle that can hopefully be baked into the definition of the datatype, unary parametricity implies induction. The next bigger step is to try to give a generic development of induction with HOAS, for any type scheme satisfying certain (as yet to be delineated) restrictions. The final goal is to extend Cedille's datatype notations to allow HOAS, and elaborate those notations down to the generic version of induction for HOAS.

Acknowledgments. I gratefully acknowledge NSF support under award 1524519, and DoD support under award FA9550-16-1-0082 (MURI program). Many thanks to the anonymous LFMTP '19 reviewers for very helpful feedback, incorporated into the final version, including the suggestion to include conversion to de Bruijn notation as an example. AMDG.

References

[1] Jean-Philippe Bernardy & Marc Lasson (2011): *Realizability and Parametricity in Pure Type Systems*. In Martin Hofmann, editor: *Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, *Lecture Notes in Computer Science* 6604, Springer, pp. 108–122, doi:10.1007/978-3-642-19805-2_8.

[2] Richard S. Bird & Oege de Moor (1997): *Algebra of programming*. Prentice Hall International series in computer science, Prentice Hall, doi:10.1017/S095679689922326X.

[3] N.G. de Bruijn (1994): *A Survey of the Project Automath**Reprinted from: Seldin, J. P. and Hindley, J. R., eds., To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, p. 579-606, by courtesy of Academic Press Inc., Orlando*. In R.P. Nederpelt, J.H. Geuvers & R.C. de Vrijer, editors: *Selected Papers on Automath*, *Studies in Logic and the Foundations of Mathematics* 133, Elsevier, pp. 141 – 161, doi:10.1016/S0049-237X(08)70203-9.

[4] Robert L. Constable, Stuart F. Allen, Mark Bromley, Rance Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, Todd B. Knoblock, N. P. Mendler, Prakash Panangaden, James T. Sasaki & Scott F. Smith (1986): *Implementing mathematics with the Nuprl proof development system*. Prentice Hall.

[5] Larry Diehl, Denis Firsov & Aaron Stump (2018): *Generic Zero-cost Reuse for Dependent Types*. *Proc. ACM Program. Lang.* 2(ICFP), pp. 104:1–104:30, doi:10.1145/3236799.

[6] Denis Firsov, Richard Blair & Aaron Stump (2018): *Efficient Mendler-Style Lambda-Encodings in Cedille*. In Jeremy Avigad & Assia Mahboubi, editors: *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, *Lecture Notes in Computer Science* 10895, Springer, pp. 235–252, doi:10.1007/978-3-319-94821-8_14.

[7] Denis Firsov & Aaron Stump (2018): *Generic Derivation of Induction for Impredicative Encodings in Cedille*. In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, ACM, New York, NY, USA, pp. 215–227, doi:10.1145/3167087.

[8] Herman Geuvers (2001): *Induction Is Not Derivable in Second Order Dependent Type Theory*. In: *Typed Lambda Calculi and Applications (TLCA)*, pp. 166–181, doi:10.1007/3-540-45413-6_16.

[9] Jean-Yves Girard (1972): *Interprétation functionelle et élimination des coupures dans l'arithmétique d'ordre supérieure*. Ph.D. thesis, Université Paris VII.

[10] J. Roger Hindley & Jonathan P. Seldin (2008): *Lambda-Calculus and Combinators: An Introduction*, 2 edition. Cambridge University Press, New York, NY, USA, doi:10.1017/CBO9780511809835.

[11] A. Kopylov (2003): *Dependent intersection: a new way of defining records in type theory*. In: *18th Annual IEEE Symposium of Logic in Computer Science, 2003. Proceedings.*, pp. 86–95, doi:10.1109/LICS.2003.1210048.

[12] Zhaohui Luo (1990): *An extended calculus of constructions*. Ph.D. thesis, University of Edinburgh, UK. Available at `http://hdl.handle.net/1842/12487`.

[13] P. Martin-Löf (1975): *An intuitionisitc theory of types, Predicative part*. In: *Logic Colloquium 1973*, pp. 73–118.

[14] N. P. Mendler (1991): *Inductive Types and Type Constraints in the Second-Order lambda Calculus*. *Ann. Pure Appl. Logic* 51(1-2), pp. 159–172, doi:10.1016/0168-0072(91)90069-X.

[15] Dale Miller & Alwen Tiu (2005): *A proof theory for generic judgments*. *ACM Trans. Comput. Log.* 6(4), pp. 749–783, doi:10.1145/1094622.1094628.

[16] Alexandre Miquel (2001): *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*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 344–359, doi:10.1007/3-540-45413-6_27.

[17] John C. Mitchell & Eugenio Moggi (1991): *Kripke-Style Models for Typed lambda Calculus*. *Ann. Pure Appl. Logic* 51(1-2), pp. 99–124, doi:10.1016/0168-0072(91)90067-V.

[18] Christine Paulin-Mohring (1993): *Inductive Definitions in the System Coq - Rules and Properties*. In: *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, TLCA '93, Springer-Verlag, London, UK, UK, pp. 328–345, doi:10.1007/BFb0037116.

[19] Frank Pfenning & Conal Elliott (1988): *Higher-Order Abstract Syntax*. In Richard L. Wexelblat, editor: *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, ACM, pp. 199–208, doi:10.1145/53990.54010.

[20] Frank Pfenning & Christine Paulin-Mohring (1989): *Inductively Defined Types in the Calculus of Constructions*. In M. Main, A. Melton, M. Mislove & D. Schmidt, editors: *Proceedings of the Fifth Conference on the Mathematical Foundations of Programming Semantics, Tulane University, New Orleans, Louisiana*, Springer-Verlag LNCS 442, pp. 209–228, doi:10.1007/BFb0040259.

[21] Benjamin C. Pierce & David N. Turner (2000): *Local type inference*. *ACM Trans. Program. Lang. Syst.* 22(1), pp. 1–44, doi:10.1145/345099.345100.

[22] John C. Reynolds (1974): *Towards a theory of type structure*. In Bernard Robinet, editor: *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974*, *Lecture Notes in Computer Science* 19, Springer, pp. 408–423, doi:10.1007/3-540-06859-7_148.

[23] John C. Reynolds (1983): *Types, Abstraction and Parametric Polymorphism*. In: *IFIP Congress*, pp. 513–523.

[24] Carsten Schürmann, Joëlle Despeyroux & Frank Pfenning (2001): *Primitive recursion for higher-order abstract syntax*. *Theor. Comput. Sci.* 266(1-2), pp. 1–57, doi:10.1016/S0304-3975(00)00418-7.

[25] Peter Selinger (2002): *The lambda calculus is algebraic*. *J. Funct. Program.* 12(6), pp. 549–566, doi:10.1017/S0956796801004294.

[26] Aaron Stump (2017): *The calculus of dependent lambda eliminations*. *Journal of Functional Programming* 27, p. e14, doi:10.1017/S0956796817000053.

[27] Aaron Stump (2018): *From realizability to induction via dependent intersection*. *Ann. Pure Appl. Logic* 169(7), pp. 637–655, doi:10.1016/j.apal.2018.03.002.

[28] Aaron Stump (2018): *Syntax and Semantics of Cedille*. *CoRR* abs/1806.04709. Available at `http://arxiv.org/abs/1806.04709`.

[29] Aaron Stump (2018): *Syntax and Typing for Cedille Core*. *CoRR* abs/1811.01318. Available at `http://arxiv.org/abs/1811.01318`.

[30] Aaron Stump & Peng Fu (2016): *Efficiency of lambda-encodings in total type theory*. *Journal of Functional Programming* 26, doi:10.1017/S0956796816000034.

[31] The Agda development team (2016): *Agda*. Available at `http://wiki.portal.chalmers.se/agda/pmwiki.php`. Version 2.5.1.

[32] The Coq development team (2016): *The Coq proof assistant reference manual*. LogiCal Project. Available at `http://coq.inria.fr`. Version 8.5.

[33] Philip Wadler (1990): *Recursive types for free!* Available at `http://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt`.

[34] Geoffrey Washburn & Stephanie Weirich (2008): *Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism*. *J. Funct. Program.* 18(1), pp. 87–140, doi:10.1017/S0956796807006557.

[35] J. B. Wells (1999): *Typability and Type Checking in System F are Equivalent and Undecidable*. *Ann. Pure Appl. Logic* 98(1-3), pp. 111–156, doi:10.1016/S0168-0072(98)00047-5.

[36] Benjamin Werner (1994): *Une Théorie des Constructions Inductives*. Ph.D. thesis, Université Paris-Diderot - Paris VII. Available at `https://tel.archives-ouvertes.fr/tel-00196524`.

[37] Freek Wiedijk (2012): *Pollack-inconsistency*. *Electr. Notes Theor. Comput. Sci.* 285, pp. 85–100, doi:10.1016/j.entcs.2012.06.008.