JFP **27**, e14, 41 pages, 2017. © Cambridge University Press 2017
doi:10.1017/S0956796817000053

*The calculus of dependent lambda eliminations**

AARON STUMP
Computer Science, The University of Iowa, Iowa City, IA, USA
(*e-mail:* astump@acm.org)

Abstract

Modern constructive type theory is based on pure dependently typed lambda calculus, augmented with user-defined datatypes. This paper presents an alternative called the Calculus of Dependent Lambda Eliminations, based on pure lambda encodings with no auxiliary datatype system. New typing constructs are defined that enable induction, as well as large eliminations with lambda encodings. These constructs are constructor-constrained recursive types, and a lifting operation to lift simply typed terms to the type level. Using a lattice-theoretic denotational semantics for types, the language is proved logically consistent. The power of CDLE is demonstrated through several examples, which have been checked with a prototype implementation called Cedille.

1 Introduction

Lambda encodings are schemes for representing datatypes and related operations as pure lambda terms. The Church encoding, where data are encoded as their own fold functions, is the best known (Church, 1941), and is typable in System F (Fortune *et al*., 1983; Böhm & Berarducci, 1985). Lambda encodings were abandoned as a basis for constructive type theory almost thirty years ago, due to the following problems, identified some time ago by several authors (Coquand & Paulin, 1988; Werner, 1992):

1. Accessors (like predecessor for numerals, or head and tail for lists) are provably asymptotically inefficient with the Church encoding (Parigot, 1989).
2. Induction principles are not derivable for lambda encodings (Geuvers, 2001).
3. Large eliminations, which compute types from data, are not possible with lambda-encoded data, at least not in normalizing type theories. This is because such theories distinguish different levels of the language, such as terms, types, kinds, etc., and one cannot apply a function at one level to compute a term at a higher level. Also, using impredicative quantification $\forall X : \star$ one level up leads to failure of normalization, and hence logical consistency (Coquand, 1986).
4. Without large eliminations, it is not possible to prove basic negative facts about lambda-encoded data, like $0 \neq 1$ (Werner, 1992).

* I gratefully acknowledge NSF support of this project under award 1524519, and DoD support under award FA9550-16-1-0082 (MURI program).

On the positive side, there is one powerful benefit of typed lambda encodings, not available with primitive datatypes:

- Higher order encodings – where datatypes contain embedded functions whose types have negative occurrences of the datatype symbol – are permitted without violating normalization. With primitive datatypes as in Coq or Agda, negative occurrences of the datatype in the datatype definition very easily lead to failure of normalization.

Parigot (1988) solved the first problem with an alternative lambda encoding, which is typable in System F plus positive-recursive types, where accessors are computable in constant time, as expected. By Geuvers's (2001) result, there is no alternative but to add something to the core impredicative dependent type theory, to solve even just the problem of induction. The present paper proposes two new type constructs for this, called constructor-constrained recursive types and lifting. The former deepens earlier work by Fu and Stump (2014) on System S, which solves the problem of induction using a typing construct called self types to allow the type to refer to the subject of the typing via bound variable x in $\iota x.T$. To prove consistency, they rely on a dependency-eliminating translation to System F_ω plus positive-recursive types. This method is not applicable to analyze a system with large eliminations, where dependence of types on terms is fundamental.

In the present paper, a deeper analysis of intrinsically inductive lambda encodings is undertaken, with a direct lattice-theoretic semantics, which can account for large eliminations. In the rest of this section, the two new features that enable intrinsically inductive lambda-encoding and large eliminations with lambda-encodings, respectively, are surveyed. Then, we turn to the definition (Sections 2 and 3) and analysis (Sections 5 and 6) of the new type theory incorporating these features, called the Calculus of Dependent Lambda Eliminations (CDLE). This system is a type-assignment system, not suitable for implementation. An algorithmic approach to CDLE, which has been implemented in a prototype tool called Cedille, is then considered, together with examples (Sections 7 and 8). A comparison with related work is in Section 10. We begin by looking in a little more detail at the problems with lambda encodings in pure type theory.

1.1 The problems, in more detail

Church's (1941) encoding of natural numbers in untyped lambda calculus defines each numeral n as follows:

$$\lambda s.\lambda z.\underbrace{s\ \cdots\ (s}_{n}\ z)$$

With this encoding, every function on the natural numbers is to be computed by iteration, and numbers are identified with iterators. Kleene found a clever way to compute the predecessor of Church-encoded n in this framework, but the operation requires $O(n)$ reduction steps, instead of the expected $O(1)$. This limitation has been stressed many times in the literature as a point against lambda-encodings. But Parigot (1988) solved this problem some time ago, with an encoding where data are represented not as iterators but as recursors. Every call to the iterated function is presented with the predecessor number as well as the result of iteration on that number. So, 2 is encoded as $\lambda s.\lambda z.s\ 1\ (s\ 0\ z)$. While in theory the space required for normal forms is exponential, in practice closure-based implementations of lambda calculus compute efficiently with Parigot encodings, as has been found in several studies (Koopman *et al.*, 2014; Stump & Fu, 2016). And Parigot encodings can be typed in a normalizing extension of System F with positive-recursive types (cf. Mendler (1988), Abel & Matthes (2004)). So efficiency of accessors is not a problem for lambda encodings in total type theory, if one uses the Parigot encoding.

Let us consider then the problem of induction. Based on the Church encoding in untyped lambda calculus, Fortune *et al.* (1983) proposed an encoding of natural number n in the second-order lambda calculus; i.e., System F (Girard *et al.*, 1989):

$$\Lambda X.\lambda s : X \to X.\lambda z : X.\underbrace{s \cdots (s}_{n}\ z)$$

(Here, we are writing ΛX for abstraction over types.) This idea was extended to a schematic encoding for a class of inductive datatypes by Böhm and Berarducci (1985). An even more general encoding for inductive datatypes in the Calculus of Constructions (CC) was proposed by Pfenning and Paulin-Mohring (1989). The type for natural numbers in these typed encodings is

$$Nat \ = \ \forall X.(X \to X) \to X \to X$$

The constructors Z (zero) and S (successor) are defined in the following way:

$$\begin{array}{lcl} Z & = & \Lambda X.\lambda s : X \to X.\lambda z : X.z \\ S & = & \lambda n : Nat.\Lambda X.\lambda s : X \to X.\lambda z : X.s\ (n\ X\ s\ z) \end{array}$$

The definition of Nat above is second order, but not dependent. So, it is sufficient for computation – and indeed one can define the basic numeric functions using it – but it is not adequate for proofs. For example, one can define addition thus

$$add \ = \ \lambda n : Nat.\lambda m : Nat.n\ Nat\ S\ m$$

And one might then wish to prove a theorem like commutativity of addition

$$\forall n : Nat.\forall m : Nat.Eq\ Nat\ (add\ n\ m)\ (add\ m\ n)$$

This is standardly proved by induction (with two subsidiary lemmas also proved by induction). Under the Curry–Howard correspondence widely used in constructive type theory, a proof of such a theorem is a closed term, which inhabits that dependent type, using a standard representation of Leibniz equality Eq in type theory. So for induction, needed for proving such theorems, we are seeking an inhabitant of the type

$$\forall P : Nat \to \star.(\forall n : Nat.P\ n \to\ P\ (S\ n)) \to P\ Z \to \forall n : Nat.P\ n$$

Geuvers (2001) proved that this type cannot be inhabited in second-order dependent type theory, for any choice of $Nat : \star$, $S : Nat \to Nat$, and $Z : Nat$. This remarkable result, proved by a model construction, would seem to close the door on lambda encodings for inductive theorem proving. This is the first main problem.

Another way to see the difficulty is to consider how to extend the definition of *Nat*

$$Nat \ = \ \forall X.(X \to X) \to X \to X$$

to a dependently typed version. So the goal is to define numbers not as their own iteration principles, but rather as their own induction principles. We must go from a type $X : \star$ to a predicate $P : Nat \to \star$. And instead of step and base case of iteration of type $X \to X$ and X respectively, we need step and base cases of induction. One could try out something like the following:

$$Nat \ = \ \forall P : Nat \to \star.(\forall x : Nat.P\ x \to P\ (S x)) \to P\ Z \to P\ ?$$

There are several issues here. First, the definition needs to be recursive, if we are to define *Nat* in terms of predicates P on *Nat*. Fortunately, the occurrence of *Nat* on the right-hand side of this equation is positive, so we do not violate the positivity requirement needed to preserve normalization. But then we have some puzzles. The definition needs to refer to the constructors S and Z for this datatype. But how could we hope to define those prior to this definition, since they are operations on the type *Nat*? Even if somehow some simultaneous definition were possible, we have the question of what to put for the question mark. An intrinsically inductive natural number n must prove any given property P for n itself, given proofs of the step and base cases. It is completely unclear *a priori* how one could set this up. Indeed, Geuver's result implies that there is no way to do this, without an extension to the type theory. We will how this is solved with CDLE, in Section 1.2.

The second main problem is that of large eliminations, or computing types by recursion on terms. In System F, with the usual definition of the *Nat* type, large eliminations are impossible, since to compute anything recursively from n of type *Nat*, we must first instantiate the universally quantified type variable in the definiens of *Nat*, to the type which we will compute by recursion on n. In order to compute a type, this instantiation should be by $\star$, since this is the type for types (and we are seeking to compute a type). But, it is well known that positing that "type" is a type (i.e., $\star : \star$) leads to failure of normalization for the language (see Coquand (1986) and Meyer & Reinhold (1986)).

So, there is no way to compute a type by recursion on a *Nat* in System F; in other words, large eliminations are not possible. This is bad enough, but there is another undesirable consequence. The usual proof in type theory that constructors have disjoint ranges – so for example, $0 \neq 1$ – relies on large eliminations. Leibniz equality states that equal expressions satisfy the same predicates, and using large eliminations we can define a predicate P on natural numbers n, which is *True* if n is zero and *False* otherwise. Here, *True* can be taken as any inhabited type, such as $\forall X : \star.X \to X$; and *False* as any uninhabited one, like $\forall X : \star.X$. If 0 equals 1, then $P\ 0$ implies $P\ 1$. Since $P\ 0$ is *True* and $P\ 1$ is *False*, we can inhabit *False* from an assumption that 0 equals 1. Without large eliminations, this proof method fails, and indeed as Werner (1992) argues, the erasure of the statement of Leibniz equality of 0 and 1 is just $\forall P.P \to P$, where one erases types of CC by dropping all term parts of types. So if we could inhabit $(Eq\ Nat\ 0\ 1) \to \forall X : \star.X$ in CC, we could also inhabit $True \to False$ in System F_ω (to which one erases CC terms and types); but this type is not inhabited. So not only does the proof method using large eliminations fail, the type $0 \neq 1$ simply cannot be inhabited, or else its erasure $True \to False$ would be, too (and the latter is not).

Traditionally, the solution proposed to these problems has been to add primitive inductive types to type theory. One way is to follow the methodology of Martin-Löf (1984) and Constable *et al.* (1986), and work with open type theories, where new inductive types can be added as extensions of the theory. This approach has been proposed also for impredicative type theory, by Coquand and Paulin (1988). Alternatively, one can define a closed type theory with type and term constructs for some class of inductive types. This is the approach of the Calculus of Inductive Constructions developed by Werner (1994), which is the foundation of the Coq interactive theorem prover (The Coq development team, 2015). One can also find an interesting intermediate approach in the literature: in Pfenning and Paulin-Mohring's (1989) approach, inductive types are lambda-encoded but their induction principles and associated reduction rules are added as extensions of the theory.

This paper proposes new solutions to the two main problems of induction and large elimination for lambda-encoded data, in a closed type theory, without primitive inductive types or primitive induction. Let us take a brief initial look at the two new typing constructs.

1.2 Constructor-constrained recursive types

To define intrinsically inductive lambda-encodings, we begin with the dependent intersection types of Kopylov (2003). We will denote these types with prefix notation $\iota x : T.T'$ instead of Kopylov's $x : T \cap T'$. Let S and Z be meta-level abbreviations for $\lambda n.\lambda s.\lambda z.s\ (n\ s\ z)$ and $\lambda s.\lambda z.z$, respectively. Also, we will make use of a top type $\mathcal{U}$, inhabited by all closed λ-abstractions. Now at the meta-level, define a sequence of types by recursion on meta-level natural number k, with increasing support for dependent typing as follows:

$$\begin{array}{lcl} Nat_0 & := & \mathcal{U} \\ Nat_{k+1} & := & \iota n : Nat_k.\forall P : Nat_k \to \star. \\ & & (\forall n : Nat_k.P\ n \to P\ (S\ n)) \to P\ Z \to P\ n \end{array}$$

Nat_{k+1} denotes the subset of Nat_k for which induction holds, for predicates on Nat_k. We use intersection types, because the natural proof that n is inductive may be identified, in a type assignment system such as we will consider, with n itself. This striking observation is due to Leivant (1983). We will see this in more detail below (Section 4).

Now, the goal is to internalize the limit of this sequence of types as a single type $\mathbb{N}$, using a positive-recursive type. This is not possible with standard forms of recursive types, due to type dependency. For suppose, we tried to define $\mathbb{N}$ as

$$\begin{array}{l} \mu\, Nat : \star.\ \iota\, n : Nat. \\ \quad \forall\, P : Nat \to \star.(\forall\, n : Nat.\, P\ n \to P\ (S\ n)) \to P\ Z \to P\ n \end{array}$$

To kind this type, we would have to kind $(P\ Z)$, which requires showing that $\lambda s.\lambda z.z$ can be assigned type *Nat*. To do this, we would unfold the definition of *Nat*, and then before we could add local variables s and z to the context, we would be forced again to kind $(P\ Z)$, since this would be the type for z. There is a circularity here, which System S avoided by using an ad-hoc form of mutually recursive types (Fu & Stump, 2014).

Here, we handle the problem with a closer connection to the semantics for types we will develop. We introduce the novel type form $\nu\, X : \kappa \,|\, \Theta.\, T'$, for what we call *constructor-constrained recursive types*. Here κ is a kind, and Θ is a set of typings that hold for $\mathscr{U}$ and are preserved by T'. We will define $\mathbb{N}$ to be

$$\begin{array}{l}\nu\, Nat : \star \,|\, S \in Nat \to Nat,\ Z \in Nat.\iota\, n : Nat.\\ \quad \forall\, P : Nat \to \star.(\forall\, n : Nat.\, P\ \ n \to P\ \ (S\ \ n)) \to P\ \ Z \to P\ \ n\end{array}$$

Semantically, this will be interpreted as the greatest lower bound of the decreasing sequence of meanings for Nat_k, defined above. The key new idea is to include this set Θ (here, $S \in Nat \to Nat$, $Z \in Nat$) of typings, which hold for $\mathscr{U}$ and are preserved as we pass further into the sequence. This is so that we can kind the body of the ν-type. For the semantic analysis, it will turn out to be critical for Θ to hold not just for the decreasing sequence of meanings, but also for the greatest lower bound of that sequence. Without some restriction, this appears not to be guaranteed. Here, we require that each typing constraint in Θ must be of the form $\Pi x_1 : T_1. \cdots \Pi x_n : T_n.\ T$, where the ν-bound variable occurs only positively in $T_1, \ldots, T_n$, and only at the head of T (i.e., T is either X or X applied to some X-free expressions). $Nat \to Nat$ meets this requirement, as a simple example, but so do more complex types.

CDLE's type system has a rule for folding and unfolding ν-types. There is also a rule for typing of constructors: $\Gamma \vdash t : [N/X]T$ is derivable for all $t \in T$ in the constructor set Θ, once a ν-type $N = \nu\, X : \kappa \,|\, \Theta.\, T'$ has been kinded in context Γ.

1.3 Lifting terms to the type level

The basic idea for supporting large eliminations with lambda encodings is to lift expressions explicitly from the term level of the language to the type level. While it is well known that one cannot lift the entire term language to the type level without losing normalization (Coquand, 1986), there is no problem with lifting simply typed terms. For example, the term $\lambda s.\lambda z.s\ (s\ z)$ representing 2 in the Church encoding can be lifted to the type level as $\lambda s : \kappa \to \kappa.\lambda z : \kappa.s\ (s\ z)$, for any particular kind κ (for example, $\star$, the kind which classifies types). Certainly the ability to do arithmetic with simply-typed lambda encodings is limited (cf. Leivant (1991)). But typically for large eliminations, one seeks just to do a single fold over the datatype to compute a type from the data. For example, for statically typed `printf`, as proposed by Augustsson (1998), one wishes to compute the type of the rest of the arguments to `printf` from the format string. This requires just a single fold.

CDLE introduces a novel construct $\uparrow_L t$, representing the type obtained by lifting a simply-typed term t to the type level. The type L is a *lifting type*, which serves to constrain the type of t to be simply typed, and also shows how that type should be lifted to a kind. For example, to lift Church-encoded 2 to the type level, one writes $\uparrow_{(*\to *)\to *\to *} 2$, where $*$ is a primitive lifting type used to represent the kind $\star$. We are not lifting 2 at its polymorphic type $\forall X : \star.(X \to X) \to X \to X$, of course, as this type is not permitted at the kind level. Instead, we are lifting an instantiation $(X \to X) \to X \to X$ of this type, where $*$ indicates the instantiation points.

One technical issue that must be addressed with this idea is the presence of variables, which occur free inside a lifting expression. For a simple example, suppose we have a free variable x of type $\forall X : \star : X \to X.$, and consider this type, where x is being instantiated to $*$:

$$\uparrow_{*\to *} \lambda y.x\ y$$

It is tempting always to push lifting across λ-abstractions, but if we do that here, we will get

$$\lambda y : \star.\uparrow_{*} (x\ y)$$

The body is not typable, because x (instantiated to have type $* \to *$) is being applied to a type, namely y of type $\star$.

One can imagine several solutions to this problem. Here, we opt not to push lifting across a series of λ-abstractions unless the body is of the form $x\ \overline{t}$, where x is bound in that series. We will form type-level β-redexes for the arguments $\overline{t}$, in case they are not headed by a variable in the series. So, we will lift the successor $\lambda s.\lambda z.s\ (n\ s\ z)$ of Church-encoded n to

$$\lambda\, s : \star \to \star.\, \lambda\, z : \star.\, s\ ((\uparrow_{(*\to *)\to(*\to *)} \lambda s.\lambda z.n\ s\ z)\ s\ z)$$

Despite this trick, we will still need some additional conversion principles for lifting, which we will see below.

Note that in this paper, to avoid further technicalities, we only formalize lifting pure simple types over $*$. The Cedille implementation, however, also supports the obvious extension of this mechanism to simple types whose domain types are not lifting types, like $* \to \mathit{Nat} \to *$. This example lifts to the kind $\star \to \mathit{Nat} \to \star$.

2 Syntax

Figure 1 gives the syntax of CDLE. We are separating clauses of the grammars with $||$, to avoid confusion with the single vertical bar in the syntax for ν-types. We use $\forall$ consistently in the types for functions for which no argument is explicitly given when the function is called. So these are *implicit products*, as introduced by Miquel (2001). Π is used for explicit products, where an argument is required when applied. We do not use kind-level implicit products, so the bound variable in any type-level λ-abstraction must be annotated.

The type $\mathcal{U}$ is a universal type, inhabited by all closed λ-abstractions. In the construct $\nu X : \kappa \,|\, \Theta.T$, the scope of bound variable X is Θ and the *body* T. We are using ν instead of μ, because our semantics will make $\nu X : \kappa \,|\, \Theta.T$ the greatest fixed-point of T. Nevertheless, we will focus here on using this type for inductive datatypes, not coinductive ones (which are outside the scope of this paper). Several

variables x, X		
terms t	$::=$	$x \mid\mid \lambda x.t \mid\mid t\ t'$
kinds κ	$::=$	$\star \mid\mid \Pi x{:}T.\kappa \mid\mid \Pi X{:}\kappa.\kappa'$
types T	$::=$	$X \mid\mid \Pi x{:}T.T' \mid\mid \forall X{:}\kappa.T \mid\mid \forall x{:}T.T' \mid\mid$
		$\iota x{:}T.T' \mid\mid \nu X{:}\kappa \mid \Theta.T' \mid\mid \lambda x{:}T.T' \mid\mid$
		$\lambda X{:}\kappa.T \mid\mid T\ t \mid\mid T\ T' \mid\mid \mathscr{U} \mid\mid \uparrow_L t$
lifting types L	$::=$	$* \mid\mid L \to L'$
constructor sets Θ	$::=$	$\cdot \mid\mid t \in T, \Theta$
typing contexts Γ	$::=$	$\cdot \mid\mid \Gamma, X:\kappa \mid\mid \Gamma, t:T \mid\mid \Gamma, t \in T$

Fig. 1. Syntax of CDLE, and typing contexts.

rules related to ν-types will make use of a notation $\mathscr{U}_\kappa$ for a top type at kind κ. We define this by recursion on κ:

$$\begin{array}{lcl} \mathscr{U}_\star & = & \mathscr{U} \\ \mathscr{U}_{\Pi\, x:T.\,\kappa} & = & \lambda\, x : T.\, \mathscr{U}_\kappa \\ \mathscr{U}_{\Pi\, X:\kappa.\,\kappa'} & = & \lambda\, X : \kappa.\, \mathscr{U}_{\kappa'} \end{array}$$

We usually elide the final ", $\cdot$" from constructor sets Θ and typing contexts Γ. We use other standard notations for typed lambda calculus, in particular $T \to T'$ for $\Pi\, x : T.\, T'$ when x is not free in T'. The type $\iota\, x : T.\, T'$ is a dependent intersection type, as introduced by Kopylov (2003). We use $t \in T$ to denote a constraint that term t has type T, as opposed to a declaration of a variable x to have type T (written $x : T$). Here, we see one unusual feature of the type system, which is that the context may contain hypotheses that a term has a given type ($t \in T$). This feature comes in with the constructor-constrained recursive types $\nu\, X : \kappa \,|\, \Theta.\, T$. We will see how to avoid it when we turn to the Cedille implementation of CDLE (Section 7). We implicitly assume that Γ does not declare any variable x or X twice, and that bound variables are renamed to enforce this. If the set Θ is empty, we may write $\nu\, X : \kappa.\, T$ instead of $\nu\, X : \kappa \,|\, \Theta.\, T$.

3 Type assignment

We consider now the type assignment rules for CDLE. These include a number of features that would make them unsuitable for direct use in a practical implementation. By accepting some non-algorithmic features, we can more easily establish, in CDLE, a firm theoretical foundation for the practical implementation of dependent typing based on pure lambda encodings. We will see how this works out when we turn to the Cedille implementation (Section 7).

The typing rules for terms and constructor sets are in Figure 2. We also use kinding rules for types, in Figure 3. Figure 4 gives kinding rules for constructor sets, and superkinding rules. (Note that the $\Box$ in the superkinding judgment is considered part of the syntax of the judgment, not a separate form of expression.) Figure 6 defines judgements imposing the restriction mentioned above on the form of types in constructor sets Θ. To express our positivity requirement for kinding ν-types, we use a judgment $X \in^p T$ for $p \in \{+, -\}$. The definition is unsurprising, so we relegate it to the Appendix. Note, however, that a more flexible approach is proposed in Abel and Matthes (2004), using kind-level variance annotations. Adding these to CDLE

$$\frac{(x:T)\in\Gamma}{\Gamma\vdash x:T}\qquad\frac{FV(\lambda x.t)\subseteq decl(\Gamma)}{\Gamma\vdash\lambda x.t:\mathscr{U}}$$

$$\frac{\Gamma\vdash t:T'\quad\Gamma\vdash T\triangleright T'\quad\Gamma\vdash T:\star}{\Gamma\vdash t:T}\qquad\frac{\Gamma\vdash t:T'\quad\Gamma\vdash T'\triangleright T}{\Gamma\vdash t:T}\qquad\frac{\Gamma\vdash t':T\quad t=_\beta t'}{\Gamma\vdash t:T}$$

$$\frac{\Gamma\vdash T:\star\quad\Gamma,x:T\vdash(\lambda x.t)\,x:T'}{\Gamma\vdash\lambda x.t:\Pi x{:}T.T'}\qquad\frac{\Gamma\vdash t:\Pi x{:}T_1.T_2\quad\Gamma\vdash t':T_1}{\Gamma\vdash t\,t':[t'/x]T_2}\qquad\frac{(t\in T)\in\Gamma}{\Gamma\vdash t:T}$$

$$\frac{\Gamma\vdash\kappa:\Box\quad\Gamma,X:\kappa\vdash t:T}{\Gamma\vdash t:\forall X{:}\kappa.T}\qquad\frac{\Gamma\vdash t:\forall X{:}\kappa.T\quad\Gamma\vdash T':\kappa}{\Gamma\vdash t:[T'/X]T}\qquad\frac{\Gamma\vdash t:\iota x{:}T.T'}{\Gamma\vdash t:[t/x]T'}$$

$$\frac{\Gamma\vdash T:\star\quad\Gamma,x:T\vdash t:T'\quad x\notin FV(t)}{\Gamma\vdash t:\forall x{:}T.T'}\qquad\frac{\Gamma\vdash t:\forall x{:}T_1.T_2\quad\Gamma\vdash t':T_1}{\Gamma\vdash t:[t'/x]T_2}\qquad\frac{\Gamma\vdash t:\iota x{:}T.T'}{\Gamma\vdash t:T}$$

$$\frac{\Gamma\vdash t:T\quad\Gamma\vdash t:[t/x]T'}{\Gamma\vdash t:\iota x{:}T.T'}\qquad\frac{\Gamma\vdash t:T\quad\Gamma\vdash\Theta}{\Gamma\vdash t\in T,\Theta}\qquad\frac{}{\Gamma\vdash\cdot}$$

$$\frac{N=\nu X{:}\kappa\,|\,\Theta_1,t\in T,\Theta_2.T'\quad\Gamma\vdash N:\kappa}{\Gamma\vdash t:[N/X]T}$$

Fig. 2. Typing of terms and constructor sets.

$$\frac{\Gamma\vdash T_1:\star\quad\Gamma,x:T_1\vdash T_2:\star}{\Gamma\vdash\forall x{:}T_1.T_2:\star}\qquad\frac{\Gamma\vdash\kappa:\Box\quad\Gamma,X:\kappa\vdash T:\star}{\Gamma\vdash\forall X{:}\kappa.T:\star}\qquad\frac{\Gamma\vdash T_1:\star\quad\Gamma,x:T_1\vdash T_2:\star}{\Gamma\vdash\Pi x{:}T_1.T_2:\star}$$

$$\frac{\Gamma\vdash T:\star\quad\Gamma,x:T\vdash T':\star}{\Gamma\vdash\iota x{:}T.T':\star}\qquad\frac{\Gamma\vdash T:\star\quad\Gamma,x:T\vdash T':\kappa}{\Gamma\vdash\lambda x{:}T.T':\Pi x{:}T.\kappa}\qquad\frac{\Gamma\vdash\kappa:\Box\quad\Gamma,X:\kappa\vdash T':\kappa'}{\Gamma\vdash\lambda X{:}\kappa.T':\Pi X{:}\kappa.\kappa'}$$

$$\frac{\Gamma\vdash T:\Pi x{:}T'.\kappa\quad\Gamma\vdash t:T'}{\Gamma\vdash T\,t:[t/x]\kappa}\qquad\frac{\Gamma\vdash T:\Pi X{:}\kappa.\kappa'\quad\Gamma\vdash T':\kappa}{\Gamma\vdash T\,T':[T'/X]\kappa'}\qquad\frac{(X:\kappa)\in\Gamma}{\Gamma\vdash X:\kappa}$$

$$\frac{}{\Gamma\vdash\mathscr{U}:\star}\qquad\frac{\Gamma,X:\star\vdash t:|L|_X}{\Gamma\vdash\uparrow_L t:\mathit{lift}(L)}$$

$$\frac{\begin{array}{lll}X\in^+T & & \\ \Gamma\vdash\kappa:\Box & \mathrm{Ctors}_X\,\Theta & \Gamma,X:\kappa\vdash\Theta:\star\\ \Gamma\vdash[\mathscr{U}_\kappa/X]\Theta & \Gamma,X:\kappa,\Theta\vdash[T/X]\Theta & \Gamma,X:\kappa,\Theta\vdash T:\kappa\end{array}}{\Gamma\vdash\nu X{:}\kappa\,|\,\Theta.T:\kappa}$$

Fig. 3. Kinding of types.

should be straightforward future work. We also write $FV(T)$ for the set of free variables (term and type) in T, and $decl(\Gamma)$ for the set of variables (term and type) declared in Γ via $x : T$ or $X : \kappa$. We write $terms(\Theta)$ for the set of terms t with constraint $t \in T$ listed in Θ for some T.

Our system has a direct-computation typing rule, as in Nuprl (Constable *et al.*, 1986). This rule uses a relation $=_\beta$, which is just standard β-equivalence of pure untyped lambda calculus. Direct computation allows us to use a more general typing rule for λ-abstractions: In the premise, we apply the λ-abstraction, rather than typing its body. Note that the rule also implies type preservation under β-reduction; the soundness of this will be established with our semantics. CDLE has forward and

$$\frac{}{\Gamma \vdash \cdot : \star} \qquad \frac{\Gamma \vdash T : \star \quad \Gamma \vdash \Theta : \star}{\Gamma \vdash (t \in T, \Theta) : \star} \qquad \frac{}{\Gamma \vdash \star : \Box}$$

$$\frac{\Gamma \vdash T : \star \quad \Gamma, x : T \vdash \kappa : \Box}{\Gamma \vdash \Pi x{:}T.\,\kappa : \Box} \qquad \frac{\Gamma \vdash \kappa : \Box \quad \Gamma, X : \kappa \vdash \kappa' : \Box}{\Gamma \vdash \Pi X : \kappa.\,\kappa' : \Box}$$

Fig. 4. Kinding of constructor sets, and superkinding.

$$\frac{N = \nu X : \kappa \,|\, \Theta.T}{\Gamma \vdash N \triangleright [N/X]T} \qquad \frac{}{\Gamma \vdash (\lambda\, x{:}T.T')\; t \triangleright [t/x]T'} \qquad \frac{}{\Gamma \vdash (\lambda\, X{:}\kappa.T)\; T' \triangleright [T'/X]T}$$

$$\frac{t \rightsquigarrow^* t'}{\Gamma \vdash T\; t \triangleright T\; t'} \qquad \frac{\Gamma \vdash t : T \quad X \notin FV(T')}{\Gamma \vdash \forall X{:}T.T' \triangleright T'} \qquad \frac{\mathit{lift}_{L,\emptyset}(t) = T}{\Gamma \vdash \uparrow_L (t) \triangleright T}$$

$$\frac{\bar{x} \notin FV(t) \quad |\bar{L}| = |\bar{x}|}{\Gamma \vdash \uparrow_{\bar{L}\to L'} \lambda\bar{x}.(t\;\bar{x}) \triangleright \uparrow_{\bar{L}\to L'} t} \qquad \frac{|\bar{L}| = |\bar{x}| = |\bar{T}|}{\Gamma \vdash (\uparrow_{\bar{L}\to L'\to L} \lambda\bar{x}.t)\; \bar{T}\; ((\uparrow_{\bar{L}\to L'} t')\; \bar{T}) \triangleright \uparrow_{\bar{L}\to L} (\lambda\bar{x}.(t\; (t'\; \bar{x})))\; \bar{T}}$$

Fig. 5. Computation rules for conversion.

backward conversion rules for typing, using a directed conversion relation $\triangleright$. The computation rules (central axioms) for $\triangleright$ are given in Figure 5. Additional rules including transitivity, reflexivity, and rules making the relation a congruence are relegated to the Appendix. Note that the congruence rules augment the context when relating the bodies of abstractions. Using a directed (non-symmetric) relation just means that it may be necessary to perform a sequence of forward and backward conversions; the key point is that the backward conversions require an additional kinding derivation. One could also consider a conversion rule for kinding, but simple situations that would require this can be solved by type-level η-expansion, and including kind-level conversion complicates inversion on kinding. So to avoid such distractions, this is omitted from CDLE. We will consider the nature of CDLE conversion further in Section 5.1 below. Several of the rules deal with lifting. We will see more about how they work below (Section 9.2).

The kinding rule for types $\uparrow_L t$ (in Figure 3), uses a meta-level function *lift*(—), defined in Figure 7, which maps lifting types to kinds as follows. The idea is to lift a type like $* \to *$ to the kind $\star \to \star$. We could also allow a lifting type $\Pi\, x : T.\, L$ to enable lifting the bodies of abstractions without lifting the classifier for the bound variable, for quantifications over terms. We omit this here for simplicity, and because it is not required for our examples. We cannot lift implicit products, because CDLE does not have these at the type level, and adding them introduces semantic

$$\frac{\mathrm{CtorTp}_X\; T \quad \mathrm{Ctors}_X\; \Theta \quad t \notin \mathit{terms}(\Theta)}{\mathrm{Ctors}_X\; (t \in T, \Theta)} \qquad \frac{}{\mathrm{Ctors}_X\; \cdot} \qquad \frac{X \in^+ T_1 \quad \mathrm{CtorTp}_X\; T_2}{\mathrm{CtorTp}_X\; \Pi x{:}T_1.T_2}$$

$$\frac{\mathrm{HeadOnly}_X\; T}{\mathrm{CtorTp}_X\; T} \qquad \frac{}{\mathrm{HeadOnly}_X\; X} \qquad \frac{X \notin FV(T)}{\mathrm{HeadOnly}_X\; T}$$

$$\frac{\mathrm{HeadOnly}_X\; T}{\mathrm{HeadOnly}_X\; (T\; t)} \qquad \frac{\mathrm{HeadOnly}_X\; T \quad X \notin FV(T')}{\mathrm{HeadOnly}_X\; (T\; T')}$$

Fig. 6. Definition of helper judgments for constructor sets.

$$
\begin{array}{lcl}
\mathit{lift}(*) & = & \star \\
\mathit{lift}(L \to L') & = & \mathit{lift}(L) \to \mathit{lift}(L') \\
\\
|*|_X & = & X \\
|S \to S'|_X & = & |S|_X \to |S'|_X \\
\\
\mathit{lift}_{L_1 \to L_2, v}(\lambda x.t) & = & \lambda\, x : \mathit{lift}(L_1).\, \mathit{lift}_{L_2,(v,(x \mapsto L_1))}(t) \\
\mathit{lift}_{L,v}(x\, \bar{t}) & = & x\ \mathit{liftargs}_{L',v}(\bar{t}),\ \text{if } (x \mapsto L') \in v \\
\\
\mathit{liftargs}_{L_1 \to L_2, v}(t\, , \bar{t}) & = & ((\uparrow_{v \to L_1} \lambda v.t)\ v)\, ,\, \mathit{liftargs}_{L_2,v}(\bar{t}) \\
\mathit{liftargs}_{L,v}(\cdot) & = & \cdot
\end{array}
$$

Fig. 7. Meta-level functions related to lifting.

complications. We also use a meta-level function $|L|_X$, also defined in Figure 7, to turn a lifting type into a type, replacing $*$ with X.

Figure 7 defines a third function $\mathit{lift}_{\_,\_}(-)$, which attempts to lift a term to a type (but may be undefined). We use vector notation $\bar{t}$ for a possibly empty sequence $t_1, \ldots, t_n$ of terms, where $\cdot$ denotes the empty sequence. We write $|\bar{t}|$ for the length of the sequence. The notation $x\ \bar{t}$ means that x is applied in a left-nested fashion to the terms $\bar{t}$. This $\mathit{lift}_{\_,\_}(-)$ function attempts to push the lifting operator ($\uparrow$) down into a λ-abstraction. Roughly speaking, it tries to turn $\uparrow\ (\lambda \bar{x}.\ x_i\ \bar{t})$ into $\lambda\, \bar{x} : \bar{\kappa}.\, x_i\ \bar{T}'$, where the kinds $\bar{\kappa}$ are derived from the lifting type given as the first argument to $\mathit{lift}_{\_,\_}(-)$, and the types $\bar{T}'$ are new lifting types derived from the arguments $\bar{t}$.

In describing these syntactic operations, we use some special notational conventions in Figure 7 with the meta-variable v, which ranges over sequences of bindings $x \mapsto L$ (where L is a lifting type). In the first equation for the $\mathit{liftargs}_{\_,\_}(-)$ helper function, we write $\lambda v.t$ to mean that all the variables listed in t should be λ-bound around t, in the order they appear in v. Also, we write $v \to L$ to mean that the lifting types in v should be added as domain types, in order, for an arrow type around L. And we write $t\ v$ to mean that the variables in v should be given as arguments, in order, for an application of t. These notations are used to implement the idea discussed in Section 1.3, of creating type-level β-redexes when pushing lifting to arguments.

4 Church-encoded natural numbers

As discussed in Section 1.2, we use the following definition for the type $\mathbb{N}$ of the natural numbers, where S and Z are meta-level abbreviations for $\lambda n.\lambda s.\lambda z.s\ (n\ s\ z)$ and $\lambda s.\lambda z.z$:

$$
\begin{array}{l}
\nu\ \mathit{Nat} : \star \mid S \in \mathit{Nat} \to \mathit{Nat},\ \ Z \in \mathit{Nat}. \\
\quad \iota\, n : \mathit{Nat}.\, \forall\, P : \mathit{Nat} \to \star. \\
\quad (\forall\, n : \mathit{Nat}.\, P\ \ n \to P\ \ (S\ \ n)) \to P\ \ Z \to P\ \ n
\end{array}
$$

These are Church-encoded numbers, because the type for the input s for successor, namely $\forall\, n : \mathit{Nat}.\, P\ \ n \to P\ \ (S\ \ n)$, uses an implicit product ($\forall$). For the Parigot encoding, one just changes this to an explicit product (Π). We will mostly focus on the Church encoding in this paper, since it is somewhat simpler and more familiar than the asymptotically more time-efficient Parigot encoding.

Let us see now in detail how to kind this type, using the ν-kinding rule as follows:

$$\frac{\begin{array}{lll} X \in^{+} T & & \\ \Gamma \vdash \kappa : \Box & \mathrm{Ctors}_X\ \Theta & \Gamma, X : \kappa \vdash \Theta : \star \\ \Gamma \vdash [\mathscr{U}_\kappa / X]\Theta & \Gamma, X : \kappa, \Theta \vdash [T/X]\Theta & \Gamma, X : \kappa, \Theta \vdash T : \kappa \end{array}}{\Gamma \vdash \nu\, X : \kappa \,|\, \Theta.\, T : \kappa}$$

The first premise is obvious, though note that *Nat* occurs positively but not *strictly positively*; the occurrences in the body of the type are in the domain parts of an even number of abstractions. The second premise is trivial. For the third premise, we can confirm easily that the constructor set for this example satisfies Ctors_{Nat}, as required. For the fourth premise: with $Nat : \star$ in the context, we can kind the constructor set $S \in Nat \to Nat, Z \in Nat$. For the fifth, we can assign $\mathscr{U} \to \mathscr{U}$ to S, using our direct-computation rule as follows:

$$\cfrac{\cfrac{\cfrac{}{\Gamma, n : \mathscr{U} \vdash \lambda s.\lambda z.s\ (n\ s\ z) : \mathscr{U}} \quad S\ n =_\beta \lambda s.\lambda z.s\ (n\ s\ z)}{\Gamma, n : \mathscr{U} \vdash S\ n : \mathscr{U}}}{\Gamma \vdash S : \mathscr{U} \to \mathscr{U}}$$

We can also assign $\mathscr{U}$ to Z.

For the sixth premise, we must show that our constructor set is preserved by the body of the ν-type. So in the context (call it Γ) $Nat : \star, S \in Nat \to Nat, Z \in Nat$, we must show the following typings, where we write *NAT* to abbreviate the body of the ν-type:

- $\Gamma \vdash S : NAT \to NAT$
- $\Gamma \vdash Z : NAT$

Let us just consider the second (the first also holds). Expanding *NAT*, we see we must show

$$\begin{array}{l} \Gamma \vdash Z : \iota\, n{:}Nat.\, \forall\, P{:}Nat \to \star. \\ \quad (\forall\, n{:}Nat.\, P\ n \to P\ (\lambda s.\lambda z.s\ (n\ s\ z))) \to \\ \quad P\ (\lambda s.\lambda z.z) \to P\ n \end{array}$$

From our constraints in Γ, we have that $\Gamma \vdash Z : Nat$. So, we can assign the first type in the dependent intersection. It remains to assign the second type, where n is instantiated with Z. For this, we can apply some introduction rules (together with direct computation) to reduce the problem to the following typing, where types like $P\ Z$ are kindable, from the constraints in Γ:

$$\Gamma, P : Nat \to \star, s : \forall n : Nat.P\ n \to P\ (S\ n), z : P\ Z \vdash z : P\ Z$$

This holds by the variable typing rule.

For the seventh premise, we must be able to assign kind $\star$ to the body of the ν-type, assuming $Nat : \star$ and the constructor set have been added to the context. The interesting observation for this is that the applications of P can be kinded. For example, to kind $P\ (S\ n)$, we use the constraint $S \in Nat \to Nat$ to assign type *Nat* to $S\ n$.

$$\begin{aligned}\overset{JK}{=} \;:=\; & \lambda A:\star.\,\lambda a:A.\,\nu E:A\to\star \mid \lambda x.x\in E\;a.\,\lambda b:A.\\ & \quad(\iota u:E\;b.\,\forall P:\forall b:A.E\;b\to\star.\,(P\;a\;\lambda x.x)\to P\;b\;u)\\ & \wedge(\iota u:E\;a.\,\forall P:E\;a\to\star.\,(P\;\lambda x.x)\to P\;u))\end{aligned}$$

Fig. 8. Equality type satisfying axioms J and K both.

If we have a term of type $\mathbb{N}$, then by unfolding the ν-type and then taking the second projection of the dependent intersection, we can use that term for dependently typed iterations; for example, inductive proofs. Of course, we can also use it for simply typed iterations as a special case, so we can implement basic terminating functions like addition, multiplication, predecessor, and so forth, in the usual way for Church-encoded numbers. We will see this more in our Cedille implementation below (Section 8).

4.1 A note on equality types

It may be of interest to some readers to know that CDLE validates axiom K for equality types (Hofmann & Streicher, 1998). K, which is equivalent to uniqueness of identity proofs, is all one must add to Martin-Löf Type Theory (MLTT) to support dependent pattern-matching, and thus is desirable for practical programming with dependent types (Goguen *et al.*, 2006). But K is incompatible with Homotopy Type Theory (HoTT) (Univalent Foundations Program, 2013), where distinguishing proofs of the same equality is essential to the approach. So CDLE is not appropriate, without significant modification, for HoTT.

In more detail, CDLE allows one to define an equality type with both J- and K-style elimination. The definition is in Figure 8, where we are writing $T \wedge T'$ for $\iota x : T. T'$ when $x \notin FV(T')$. Note that here, the top type $\mathscr{U}_{A\to\star}$ that is used when kinding $\overset{JK}{=}$ is defined (at the meta-level) to be $\lambda x : A.\mathscr{U}$. So we indeed have $\lambda x.x$ in $\mathscr{U}_{A\to\star}\;a$, when checking that the constructor set is satisfied by the top type. We can easily prove, using similar reasoning as in Section 4 above, that $\lambda x.x$ has type $\forall A : \star.\forall a : A.\; a \overset{JK}{=}_A a$.

Any relation purported to be an equality relation in type theory should be substitutive, and indeed, given t of type $a \overset{JK}{=}_A b$, we may use the first part of its conjunctive type to transform any type containing a into one containing b, as expected. And, as expected for axiom J, to prove something about such a term t as a proof of $a \overset{JK}{=}_A b$, it suffices to reason just about $\lambda x.x$ as a proof of $a \overset{JK}{=}_A a$. But, we also have the second conjunct of the type $a \overset{JK}{=}_A b$, which allows us to prove any property of u of type $a \overset{JK}{=}_A a$ by proving it for $\lambda x.x$ of that type. This is axiom K.

5 Semantics of types

To define a semantics for types, we need a few preliminary definitions. We will work with set-theoretic partial functions for the semantics of higher kinded types. An application of such a function is undefined if the argument is not in the domain of the partial function. (As standard in set theory, such functions are themselves sets.) We consider any meta-level expressions, including formulas, which contain undefined subexpressions to be undefined themselves. In lemmas and theorems, if we affirm formulas involving possibly undefined expressions, we are implicitly affirming all those expressions are defined. We write $A \to B$ for the set of meta-level total functions from set A to set B; that is, total functional subsets of $A \times B$. We write $(x \in A \mapsto b)$ for the (meta-level) function mapping input x in the set A to b.

For our semantics, we prove results about closed terms only, though for the semantics of the lifting operation we will have to consider open terms. Let $\mathscr{L}$ be the set of closed lambda abstractions (i.e., terms of the form $\lambda x.t$ with no free variables), and let $\mathscr{N} \subseteq \mathscr{L}$ be the set of closed normal-form terms. We will write $\rightsquigarrow$ for (full) β-reduction. We also write $=_{c\beta}$ for standard β-equivalence restricted to closed terms, and $[t]_{c\beta}$ for the set $\{t' \mid t =_{c\beta} t'\}$. The latter operation is extended to sets S of terms by writing $[S]_{c\beta}$ for $\{[t]_{c\beta} \mid t \in S\}$. In a few places, we write $nf(t)$ for the (unique) normal form of term t; this is undefined if t has no normal form. We write Ω for an arbitrary term without normal form, like $(\lambda x.x\ x)\ (\lambda x.x\ x)$.

Definition 1 (*Reducibility candidates*)
$\mathscr{R} := \{[S]_{c\beta} \mid S \subseteq \mathscr{L}\}$.

A reducibility candidate (element of $\mathscr{R}$) is a set of β-equivalence classes of λ-abstractions. We will use this definition to develop as technically light a semantics as possible, while still being sufficient to show logical consistency (Corollary 14 below). Further adaptation would be necessary to show normalization, but this is not needed for our consistency proof. One exception is that we will need to reason about normalization for proving soundness of lifting. Throughout the development, we will make use of a *choicefunction* ζ. Given any set E of terms, ζ returns a λ-abstraction if E contains one, and is undefined otherwise.

Lemma 2
If $E = [\lambda x.t]_{c\beta}$, then $[\zeta(E)]_{c\beta} = E$.

Lemma 3 (*$\mathscr{R}$ is a complete lattice*)
The set $\mathscr{R}$ ordered by subset forms a complete lattice, with greatest element $[\mathscr{L}]_{c\beta}$ and greatest lower bound of a non-empty set of elements given by intersection.

Lemma 4
$[\mathscr{N}]_{c\beta} \in \mathscr{R}$, and $\emptyset \in \mathscr{R}$.

Figure 9 defines our semantics for types and kinds, by mutual structural recursion. The semantic functions take arguments σ and ρ, in addition to the type or kind to interpret. We require that σ maps term variables to terms, and ρ maps type variables to sets. The interpretations of types and kinds are then also sets. We will get more precise descriptions of the domains and codomains of the semantic functions later. The interpretation of ν-types uses the notation $F^n(a)$ for (meta-level) iteration of the function F n times on a: $F(F(\ldots F(a)))$. The operation $\cap_{\kappa,\sigma,\rho}$ used in the semantics of ν-types, and the value $\top_{\star,\sigma,\rho}$ used in the semantics of $\mathscr{U}$, are defined in Figure 10. The meaning of a type can be empty, and so in interpreting $\forall\, x : T.\, T'$ we must take the intersection using $\cap_{\star}$, which returns the top element of $\mathscr{R}$ if the interpretation of T is empty. The meaning of a kind cannot be empty, however, so we do not

$$
\begin{array}{lcl}
[\![X]\!]_{\sigma,\rho} & = & \rho(X) \\
[\![\mathscr{U}]\!]_{\sigma,\rho} & = & \top_\star \\
[\![\Pi x : T_1.T_2]\!]_{\sigma,\rho} & = & [\{\lambda x.t \mid \forall E \in [\![T_1]\!]_{\sigma,\rho}. \\
 & & \quad [[\zeta(E)/x]t]_{c\beta} \in [\![T_2]\!]_{\sigma[x\mapsto\zeta(E)],\rho}\}]_{c\beta} \\
[\![\forall X : \kappa.T]\!]_{\sigma,\rho} & = & \cap\{[\![T]\!]_{\sigma,\rho[X\mapsto S]} \mid S \in [\![\kappa]\!]_{\sigma,\rho}\} \\
[\![\forall x : T.T']\!]_{\sigma,\rho} & = & \cap_\star\{[\![T']\!]_{\sigma[x\mapsto\zeta(E)],\rho} \mid E \in [\![T]\!]_{\sigma,\rho}\} \\
[\![\iota x : T.T']\!]_{\sigma,\rho} & = & \{E \in [\![T]\!]_{\sigma,\rho} \mid E \in [\![T']\!]_{\sigma[x\mapsto\zeta(E)],\rho}\} \\
[\![\lambda X : \kappa.T]\!]_{\sigma,\rho} & = & (S \in [\![\kappa]\!]_{\sigma,\rho} \mapsto [\![T]\!]_{\sigma,\rho[X\mapsto S]}) \\
[\![\lambda x : T.T']\!]_{\sigma,\rho} & = & (E \in [\![T]\!]_{\sigma,\rho} \mapsto [\![T']\!]_{\sigma[x\mapsto\zeta(E)],\rho}) \\
[\![T\ T']\!]_{\sigma,\rho} & = & [\![T]\!]_{\sigma,\rho}([\![T']\!]_{\sigma,\rho}) \\
[\![T\ t]\!]_{\sigma,\rho} & = & [\![T]\!]_{\sigma,\rho}([(\sigma t)]_{c\beta}) \\
[\![\nu X : \kappa \mid \Theta.T]\!]_{\sigma,\rho} & = & q,\ \textit{where} \\
 & & q = \cap_{\kappa,\sigma,\rho}\{F^n(\top_{\kappa,\sigma,\rho}) \mid n \in \mathbb{N}\} \text{ and} \\
 & & F = (S \in [\![\kappa]\!]_{\sigma,\rho} \mapsto [\![T]\!]_{\sigma,\rho[X\mapsto S]})\}; \\
 & & \text{if } F(q) = q \\
[\![\uparrow_L(t)]\!]_{\sigma,\rho} & = & \langle\!\langle nf(t)\rangle\!\rangle^L_\emptyset \\
[\![\star]\!]_{\sigma,\rho} & = & \mathscr{R} \\
[\![\Pi x : T.\kappa]\!]_{\sigma,\rho} & = & (E \in [\![T]\!]_{\sigma,\rho} \to [\![\kappa]\!]_{\sigma[x\mapsto\zeta(E)],\rho}), \\
 & & \quad \text{if } [\![T]\!]_{\sigma,\rho} \in \mathscr{R} \\
[\![\Pi x : \kappa.\kappa']\!]_{\sigma,\rho} & = & (S \in [\![\kappa]\!]_{\sigma,\rho} \to [\![\kappa]\!]_{\sigma,\rho[X\mapsto S]})
\end{array}
$$

Fig. 9. Semantics for types and kinds (see also Figures 10 and 11).

need to worry about this situation when interpreting $\forall\, X : \kappa.\, T$. For the semantics of $\Pi x : T.\kappa$, if $[\![T]\!]_{\sigma,\rho} \notin \mathscr{R}$, then the meaning of the Π-kind is undefined.

An important principle in the definition of this semantics is that if the meaning of a type is defined, then it satisfies the semantic counterparts of the conversion rules in Figure 5. So loosely, if $T \triangleright T'$ and $[\![T]\!]$ is defined, then $[\![T]\!] = [\![T']\!]$ just based on the definition of $[\![T]\!]$ (not any auxiliary information). This greatly simplifies the semantic connection between conversion and typing for the proof of semantic soundness (Theorem 13 below).

Figure 11 defines a semantic lifting function to lift terms to semantic functions at the (set-theoretic) level where they are in the interpretations of kinds. We do not need to carry the valuations σ and ρ through the definition, since we have restricted lifting types to be simple types over $*$. A different kind of valuation θ is used, which maps term variables to pairs (L, S) where L is a lifting type and S is a set. If we included types $\Pi x : T.L$ as lifting types, then we would need to make use of σ and ρ in the definitions in Figure 11.

5.1 About the conversion relation

Most type theories are defined using a congruence relation on types, which is then shown to be algorithmic by proving its confluence and normalization. For CDLE – and, it seems likely, any system combining dependent and recursive types – the situation is somewhat more complicated, as indicated by the following theorem:

Theorem 5

There is no recursively enumerable convertibility relation between types in context which is sound and complete with respect to equality of interpretations.

$$
\begin{array}{lcl}
X \subseteq_{*,\sigma,\rho} Y & \Leftrightarrow & X \subseteq Y \\
X \subseteq_{\Pi x:T.\kappa,\sigma,\rho} Y & \Leftrightarrow & \forall E \in [\![T]\!]_{\sigma,\rho}. \\
 & & \quad X(E) \subseteq_{\kappa,\sigma[x\mapsto\zeta(E)],\rho} Y(E) \\
X \subseteq_{\Pi X:\kappa.\kappa'} Y & \Leftrightarrow & \forall S \in [\![\kappa]\!]_{\sigma,\rho}. \\
 & & \quad X(S) \subseteq_{\kappa',\sigma,\rho[X\mapsto S]} Y(S) \\
\\
\top_\star & = & [\mathscr{L}]_{c\beta} \\
\top_{\star,\sigma,\rho} & = & \top_\star \\
\top_{\Pi x:T.\kappa,\sigma,\rho} & = & (E \in [\![T]\!]_{\sigma,\rho} \mapsto \top_{\kappa,\sigma[x\mapsto\zeta(E)],\rho}) \\
\top_{\Pi X:\kappa.\kappa',\sigma,\rho} & = & (S \in [\![\kappa]\!]_{\sigma,\rho} \mapsto \top_{\kappa',\sigma,\rho[X\mapsto S]}) \\
\\
\cap_\star X & = & \begin{cases} \cap X, \text{ if } X \neq \emptyset \\ \top_\star, \text{ otherwise} \end{cases} \\
\cap_{\star,\sigma,\rho} X & = & \cap_\star X \\
\cap_{\Pi x:T.\kappa,\sigma,\rho} X & = & \begin{cases} (E \in [\![T]\!]_{\sigma,\rho} \mapsto \\ \quad \cap_{\kappa,\sigma[x\mapsto\zeta(E)],\rho}\{F(E) \mid F \in X\}), \\ \quad \text{if } X \neq \emptyset \\ \top_{\Pi x:T.\kappa,\sigma,\rho}, \text{ otherwise} \end{cases} \\
\cap_{\Pi X:\kappa.\kappa',\sigma,\rho} X & = & \begin{cases} (S \in [\![\kappa]\!]_{\sigma,\rho} \mapsto \\ \quad \cap_{\kappa',\sigma,\rho[X\mapsto S]}\{F(S) \mid F \in X\}), \\ \quad \text{if } X \neq \emptyset \\ \top_{\Pi x:\kappa.\kappa',\sigma,\rho}, \text{ otherwise} \end{cases}
\end{array}
$$

Fig. 10. Pointwise-extended lattice operations.

$$
\begin{array}{lcl}
\langle\!\langle \lambda x.t \rangle\!\rangle_\theta^{L\to L'} & = & S \in [\![\mathit{lift}(L)]\!]_{\emptyset,\emptyset} \mapsto \langle\!\langle t \rangle\!\rangle^{L'}_{\theta[x\mapsto(S,L)]} \\
\langle\!\langle x\,\bar{t} \rangle\!\rangle_\theta^{L} & = & S(\langle\!\langle t_1 \rangle\!\rangle_\theta^{L_1}) \cdots (\langle\!\langle t_n \rangle\!\rangle_\theta^{L_n}), \\
 & & \quad \text{if } \theta(x) = (S, \bar{L} \to L) \text{ with } |\bar{L}| = n = |\bar{t}|
\end{array}
$$

Fig. 11. Semantic lifting $\langle\!\langle\ \rangle\!\rangle$.

Proof

We can reduce extensional equivalence of primitive recursive numeric functions to this problem. That relation is not r.e., since if it were, it would be decidable (inequivalence is obviously r.e.), and it is known not to be so. Suppose f and g have type $\mathbb{N} \to \mathbb{N}$, and consider the following two types, where S denotes successor for Church-encoded numerals as above:

$$\forall\, P : \mathbb{N} \to \star.\ \nu\, X : \mathbb{N} \to \star.\ \lambda\, n : \mathbb{N}.\ P\ (f\ \ n) \to X(S\ \ n)$$
$$\forall\, P : \mathbb{N} \to \star.\ \nu\, X : \mathbb{N} \to \star.\ \lambda\, n : \mathbb{N}.\ P\ (g\ \ n) \to X(S\ \ n)$$

These types have the same interpretation (with empty functions for σ and ρ) iff f and g return the same values for all inputs $n : \mathbb{N}$. $\square$

So CDLE must be defined using a particular incomplete conversion relation. Further use of the theory will be required to see if further (semantically justified) principles need to be added for practical use. Additional analysis of this relation, such as studying decidability or complete formulations for subrelations, must remain to future work.

5.2 *Reasoning about lifting*

To prove soundness of the conversion and kinding rules for lift types $\uparrow_L t$, we need some intricate and interesting reasoning, summarized in the following lemmas. Several of these can be viewed as semantic lemmas about simple typing. To justify the main conversion axiom about lifting, we have the following lemma:

Lemma 6
Suppose $lift_{L,v}(t)$ is defined, and suppose that $\theta(x) = (S, L)$ holds for some $S \in [\![lift(L)]\!]_{\emptyset,\emptyset}$ iff $v(x) = L$. Suppose also that $nf(t)$ is defined, and $FV(t) \subseteq dom(\theta)$. Then, $\langle\!\langle t\rangle\!\rangle^L_{\emptyset,\emptyset,\theta} = [\![lift_{L,v}(t)]\!]_{\emptyset,\rho'}$, where $\rho'(x) = S$ iff $\theta(x) = (S, L')$ for some L'.

The main lemma needed to justify kinding of lift types is the following, where we first introduce a definition relating valuations θ used in semantic lifting (Figure 11) and the valuations σ mapping term variables to terms.

Definition 7 (*(θ, R)-constrained*)
Suppose θ is a given valuation of the sort used in Figure 11, and $R \in \mathscr{R}$ is also given. Then σ is called (θ, R)-constrained iff the following holds: $\sigma(x) \in [\![|L|_X]\!]_{\emptyset,[X\mapsto R]}$ iff $\theta(x) = (S, L)$.

Lemma 8 (*Main lifting lemma*)
Let t be a possibly open term in normal form, and assume a valuation θ with $dom(\theta) \supseteq FV(t)$, and such that for all $x \in dom(\theta)$, $\theta(x) = (S, L)$ iff $S \in [\![lift(L)]\!]_{\emptyset,\emptyset}$. Also, make the following main assumption about t and L: for all non-empty $R \in \mathscr{R}$, for all (θ, R)-constrained σ, we have $[\sigma t]_{c\beta} \in [\![|L|_X]\!]_{\emptyset,[X\mapsto R]}$. Then, $\langle\!\langle t\rangle\!\rangle^L_\theta \in [\![lift(L)]\!]_{\emptyset,\emptyset}$.

This main lemma uses what turns out to be a powerful semantic idea: since the kinding rule for $\uparrow_L t$ has premise $\Gamma, X : \star \vdash t : |L|_X$, we know that we have $\sigma t \in [\![|L|_X]\!]_{\emptyset,[X\mapsto R]}$, for any $R \in \mathscr{R}$. This additional quantification over R is crucial for getting the proof to go through, and leads to other interesting consequences. First, we get normalization, because we can instantiate R with $[\mathscr{N}]_{c\beta}$ (the set of closed normalizing terms).

Lemma 9
Suppose that for all $R \in \mathscr{R}$, we have $[t]_{c\beta} \in [\![|L|_X]\!]_{\emptyset,[X\mapsto R]}$. Then t is normalizing.

Next we have to note two lemmas, easily proved by induction on the lifting type L in question.

Lemma 10
Let $\rho = [X \mapsto R]$, where $R \in \mathscr{R}$ is non-empty. Then $[\![|L|_X]\!]_{\emptyset,\rho}$ is non-empty.

Lemma 11
Suppose $[t_1]_{c\beta} \notin [\![|L_1|_X]\!]_{\emptyset,\rho}$, where $\rho = [X \mapsto R]$ for some non-empty $R \in \mathscr{R}$. Then for any L_2 there exists a term of the form $\lambda y.t_2$ such that $[\lambda y.t_2]_{c\beta} \in [\![|L_1 \to L_2|]\!]_{\emptyset,\rho}$ but $[[t_1/y]t_2] \notin [\![|L_2|]\!]_{\emptyset,\rho}$.

With these, we can derive the following strong property about inclusion of interpretations, which is needed for Lemma 8. The proof is interesting enough that it is given here in full.

$$
\begin{array}{lcl}
(\sigma \uplus [x \mapsto t], \rho) \in [\![\Gamma, x : T]\!] & \Leftrightarrow & (\sigma, \rho) \in [\![\Gamma]\!] \wedge [\![T]\!]_{\sigma,\rho} \in \mathscr{R} \wedge [t]_{c\beta} \in [\![T]\!]_{\sigma,\rho} \\
(\sigma, \rho \uplus [X \mapsto S]) \in [\![\Gamma, X : \kappa]\!] & \Leftrightarrow & (\sigma, \rho) \in [\![\Gamma]\!] \wedge S \in [\![\kappa]\!]_{\sigma,\rho} \\
(\sigma, \rho) \in [\![\Gamma, t \in T]\!] & \Leftrightarrow & (\sigma, \rho) \in [\![\Gamma]\!] \wedge [\![T]\!]_{\sigma,\rho} \in \mathscr{R} \wedge [\sigma t]_{c\beta} \in [\![T]\!]_{\sigma,\rho} \\
(\emptyset, \emptyset) \in [\![\cdot]\!] & &
\end{array}
$$

Fig. 12. Semantics of typing contexts Γ.

Lemma 12 (Trivial semantic subtyping for simple types)
Suppose that for all non-empty $R \in \mathscr{R}$, $[\![|L|_X]\!]_{\emptyset,[X\mapsto R]} \subseteq [\![|L'|_X]\!]_{\emptyset,[X\mapsto R]}$. Then, $L = L'$.

Proof
The proof is by induction on the structure of L', considering several cases. We will refer to the assumption of the theorem as our *semantic subtyping assumption*. Let $\bar{L}$ and $\bar{L}'$ be sequences of lifting types with $|\bar{L}| = |\bar{L}'| = n$, for some n.

Case: Suppose L is $\bar{L} \to *$ and L' is $\bar{L}' \to L_a \to L_b$ for some L_a and L_b. Then, we can easily violate our semantic subtyping by instantiating R with $[\mathscr{N}]_{c\beta}$ and taking $[\lambda\bar{x}.\lambda y.\Omega]_{c\beta}$ as an element in $[\![|L|_X]\!]_{\emptyset,[X\mapsto[\mathscr{N}]_{c\beta}]}$ but not in $[\![|L'|_X]\!]_{\emptyset,[X\mapsto[\mathscr{N}]_{c\beta}]}$.

Case: Suppose L is $\bar{L} \to \bar{L}'' \to \star$, for some non-empty $\bar{L}''$, and L' is $\bar{L}' \to \star$. Instantiate R in our semantic subtyping assumption with $\{[\lambda x.x]_{c\beta}\}$. Now, we will have $\lambda\bar{x}.\lambda\bar{x}''.\lambda x.x \in [\![|\bar{L}' \to \bar{L}'' \to \star|_X]\!]_{\emptyset,[X\mapsto\{[\lambda x.x]_{c\beta}\}]}$, where $|\bar{x}''| = |\bar{L}''|$. But this term is not in $[\![|\bar{L}' \to \star|_X]\!]_{\emptyset,[X\mapsto\{[\lambda x.x]_{c\beta}\}]}$ (using the fact that the quantifications imposed by the semantics of function types are not vacuous, by Lemma 10).

Case: So, we are left with the case where L is $\bar{L} \to \star$ and L' is $\bar{L}' \to \star$ (and $|\bar{L}| = |\bar{L}'|$). Suppose some L_i differs from L'_i, and suppose that i is the greatest position at which this occurs. Now let L_a be $L_{i+1} \to \cdots \to L_n \to \star$. We can prove that L'_i must be a semantic subtype of L_i, by the following argument. Assume this is not the case. Then, there is some non-empty $R \in \mathscr{R}$ such that $E \in [\![|L'_i|_X]\!]_{\emptyset,[X\mapsto R]}$ but $E \notin [\![|L_i|_X]\!]_{\emptyset,[X\mapsto R]}$. But then by Lemma 11 there is a term $\lambda y.t'$ such that $[\lambda y.t']_{c\beta} \in [\![|L_i \to L_a|]\!]_{\emptyset,[X\mapsto R]}$ but $[[\zeta(E)/y]t']_{c\beta} \notin [\![|L_a|_X]\!]_{\emptyset,[X\mapsto R]}$. Consider the term $\lambda\bar{x}.\lambda y.t'$. We have $[\lambda\bar{x}.\lambda y.t']_{c\beta}$ in $[\![|\bar{L} \to \star|]\!]_{\emptyset,[X\mapsto R]}$, by a simple application of the semantics of function types. But we do not have $[\lambda\bar{x}.\lambda y.t']_{c\beta} \in [\![|\bar{L}' \to \star|]\!]_{\emptyset,[X\mapsto R]}$. This follows (using also Lemma 10 to instantiate the variables $\bar{x}$) because $E \in [\![|L'_i|_X]\!]_{\emptyset,[X\mapsto R]}$, but we deduced $[[\zeta(E)/y]t']_{c\beta} \notin [\![|L_a|_X]\!]_{\emptyset,[X\mapsto R]}$. So, we have L'_i as a semantic subtype of L_i, and we may then apply the IH to conclude that $L'_i = L_i$. This contradicts the assumption we made that those types are different. $\square$

6 Soundness for typing

Figure 12 defines a semantics for typing contexts, for purposes of the following main theorem. In that definition, we write $\sigma \uplus [x \mapsto t]$ to mean $\sigma[x \mapsto t]$ where $x \notin dom(\sigma)$ (and similarly for $\rho \uplus [X \mapsto S]$). Figure 13 defines $[\![\kappa \;|_X\; \Theta]\!]_{\sigma,\rho}$ to be the set of those elements of $[\![\kappa]\!]_{\sigma,\rho}$, which satisfy the constraints given by Θ for type variable X. These two helper notions are used in stating the main theorem below.

Theorem 13 (Soundness of typing and kinding)
If $(\sigma, \rho) \in [\![\Gamma]\!]$, then

$$\begin{array}{lcl}
\llbracket \kappa \,|_X\, \Theta \rrbracket_{\sigma,\rho} & = & \{S \in \llbracket \kappa \rrbracket_{\sigma,\rho} \mid \llbracket \Theta \rrbracket_{\sigma,\rho[X \mapsto S]}\} \\
\llbracket \cdot \rrbracket_{\sigma,\rho} & & \\
\llbracket t \in T, \Theta \rrbracket_{\sigma,\rho} & \Leftrightarrow & \llbracket T \rrbracket_{\sigma,\rho} \in \mathscr{R} \wedge [\sigma t]_{c\beta} \in \llbracket T \rrbracket_{\sigma,\rho} \wedge \llbracket \Theta \rrbracket_{\sigma,\rho}
\end{array}$$

Fig. 13. Definition of $\llbracket \kappa \,|_X\, \Theta \rrbracket_{\sigma,\rho}$, and semantics of constructor sets Θ.

1. If $\Gamma \vdash \kappa : \Box$, then $\llbracket \kappa \rrbracket_{\sigma,\rho}$ is defined.
2. If $\Gamma \vdash T : \kappa$, then $\llbracket T \rrbracket_{\sigma,\rho} \in \llbracket \kappa \rrbracket_{\sigma,\rho}$.
3. If $\Gamma \vdash t : T$, then $[\sigma t]_{c\beta} \in \llbracket T \rrbracket_{\sigma,\rho}$ and $\llbracket T \rrbracket_{\sigma,\rho} \in \mathscr{R}$.
4. If $\Gamma \vdash \Theta : \star$ and $\Theta = t_1 \in T_1, \ldots, t_n \in T_n$, then $\llbracket T_1 \rrbracket_{\sigma,\rho} \in \mathscr{R}, \ldots, \llbracket T_n \rrbracket_{\sigma,\rho} \in \mathscr{R}$.
5. If $\Gamma \vdash \Theta$, then $\llbracket \Theta \rrbracket_{\sigma,\rho}$.
6. If $\Gamma \vdash T \triangleright T'$ and $\llbracket T \rrbracket_{\sigma,\rho} \in \llbracket \kappa \rrbracket_{\sigma,\rho}$ for some kind κ, then $\llbracket T \rrbracket_{\sigma,\rho} = \llbracket T' \rrbracket_{\sigma,\rho}$.
7. Suppose $(X : \kappa) \in \Gamma$, and let $\sigma = \sigma_1 \uplus \sigma_2$ and $\rho = \rho_1 \uplus \rho_2[X \mapsto S]$. Suppose also that $S \subseteq_{\kappa,\sigma_1,\rho_1} S'$ and $A \subseteq \llbracket \kappa \rrbracket_{\sigma_1,\rho_1}$, with $A \neq \emptyset$.
   a. If $\Gamma \vdash T : \kappa'$, $\llbracket \kappa' \rrbracket_{\sigma_1,\rho_1}$ is defined, and $X \in^+ T$, then
      i. $\llbracket T \rrbracket_{\sigma,\rho[X \mapsto S]} \subseteq_{\kappa',\sigma_1,\rho_1} \llbracket T \rrbracket_{\sigma,\rho[X \mapsto S']}$
      ii. $\cap_{\kappa',\sigma_1,\rho_1} \{\llbracket T \rrbracket_{\sigma,\rho[X \mapsto S]} \mid S \in A\} \subseteq_{\kappa',\sigma_1,\rho_1}$
      $\llbracket T \rrbracket_{\sigma,\rho[X \mapsto \cap_{\kappa,\sigma_1,\rho_1} A]}$
   b. If $\Gamma \vdash T : \kappa'$ and $X \in^- T$, then $\llbracket T \rrbracket_{\sigma,\rho[X \mapsto S']} \subseteq_{\kappa',\sigma_1,\rho_1} \llbracket T \rrbracket_{\sigma,\rho[X \mapsto S]}$.
   c. If $\Gamma \vdash \kappa' : \Box$, $\llbracket \kappa' \rrbracket_{\sigma_1,\rho_1}$ is defined, and $X \in^+ \kappa'$, then
      i. $\llbracket \kappa' \rrbracket_{\sigma,\rho} \subseteq \llbracket \kappa' \rrbracket_{\sigma,\rho[X \mapsto S']}$
      ii. $\cap\{\llbracket \kappa' \rrbracket_{\sigma,\rho[X \mapsto S]} \mid S \in A\} \subseteq \llbracket \kappa' \rrbracket_{\sigma,\rho[X \mapsto \cap_{\kappa,\sigma_1,\rho_1} A]}$
   d. If $\Gamma \vdash \kappa' : \Box$ and $X \in^- \kappa'$, then $\llbracket \kappa' \rrbracket_{\sigma,\rho[X \mapsto S']} \subseteq \llbracket \kappa' \rrbracket_{\sigma,\rho}$.
8. If $\llbracket \kappa \rrbracket_{\sigma,\rho}$ is defined, $\Gamma, X : \kappa \vdash \Theta : \star$, $\llbracket \Theta \rrbracket_{\sigma,\rho[X \mapsto \top_\kappa]}$, and $\mathrm{Ctors}_X\ \Theta$, then $(\llbracket \kappa \,|_X\, \Theta \rrbracket_{\sigma,\rho}, \subseteq_{\kappa,\sigma,\rho}, \cap_{\kappa,\sigma,\rho})$ is a complete lattice.

These parts must be proved by mutual induction on the structure of the assumed derivation in each part. Parts (1), (2), and (3) of Theorem 13 are statements that the main judgements of CDLE – superkinding, kinding, and typing, respectively – are sound with respect to our semantics. Parts (4) and (5) express soundness of two helper judgements dealing with constructor sets Θ. Parts (5) and (6) express soundness of directed conversion. Parts (7) and (8) are critical for reasoning about ν-types. Parts (7ai) and (7ci) express monotonicity of the semantics for type variables occurring only positively, and parts (7b) and (7d) express antimonotonicity for type variables occurring only negatively. Parts (7aii) and (7cii) are expressing one part of continuity, which is used in establishing that the meaning of a ν-type is indeed a fixed-point of the interpretation of its body; the other ends up following from monotonicity. Part (8) embodies one of the central insights of constructor-constrained recursive types: if a constructor set Θ satisfies $\mathrm{Ctors}_X\ \Theta$, then it is preserved not just through the chain of iterates of the interpretation of the body, but also in the limit of that sequence, its greatest lower bound. Without preservation of Θ in the limit, we cannot show that the meaning of a ν-type is the appropriate fixed point.

Corollary 14 (*Logical consistency*)
There is no derivation of $\cdot \vdash t : \forall X : \star . X$, for any term t.

Proof
By Theorem 13 part (3) and the semantics of $\forall$-types, if $\cdot \vdash t : \forall X : \star . X$ is derivable, then $t \in \cap\mathscr{R}$. But $\cap\mathscr{R}$ is empty since $\emptyset \in \mathscr{R}$. $\square$

7 Cedille: an implementation of CDLE

I have implemented a system called Cedille based on CDLE. At first glance, this may seem difficult, because of typing rules like direct computation and the introduction rule for dependent intersections, which do not fit well into usual approaches to algorithmic typing. But one insight emerges that helps us resolve these difficulties. These troublesome features of CDLE are needed solely for kinding recursive types. Once recursive types are kinded, then it is a relatively simple matter to unfold them when their inhabitants are eliminated (i.e., applied to arguments). We need never introduce them, if we are content to use the constructors of the type (from the constructor sets) as the sole means of constructing inhabitants of recursive types. This rules out defining alternative versions of operations on lambda-encoded data, such as Rosser's alternative definitions of multiplication and exponentiation (though supporting these would require additional rules in CDLE, to allow typing of non-constructor terms with recursive types). But this is an acceptable loss to gain the power of higher order encodings. A final issue is the need to add typings $t \in T$ to contexts, due to the fact that constructor sets contain typings of arbitrary terms. This issue is resolved in Cedille by introducing names for the constructors, which are used in place of those arbitrary terms. Note that while the type-checking algorithm for annotated terms implemented by Cedille is based closely on the definition of CDLE above, formally defining this algorithm and proving the appropriate relation to CDLE must remain to future work.

Cedille supports top-level definition of recursive types with the following syntax:

$$\texttt{rec}\ X\ \mathit{params}\ :\ \mathit{indices}\ |\ \mathit{ctors}\ =\ T\ \texttt{with}\ \mathit{defs}$$

Here, *params* and *indices* are telescopes of bindings, the first for parameters fixed for the whole type definition, and the second for indices, which are inputs to the type constructor X, which may change in the body T of the definition. The *ctors* are declarations of constructors; this component of the definition is just like Θ, except that constraints are of the form $x : T$, where x is a constructor name, rather than $t \in T$. The definitions of the constructors named in *ctors*, using whichever lambda encoding is being applied, are given in the *defs*. For example, Figure 14 gives definitions of three standard datatypes: `Nat` is for Church-encoded natural numbers, `List` is for Parigot-encoded lists, and `Vector` is for Parigot-encoded vectors (lists indexed by their length). Cedille uses the notation $-t$ for an implicit (erased) argument, and Λ as a term-level binder for implicit inputs. Applications of terms or types to types are written with the $\cdot$ operator for parsing reasons. In datatype definitions only, the special variable `self` may be used as an implicitly ι-abstracted variable referring to the subject of the typing.

Let us consider how Cedille kinds the definition of `Nat` (Figure 14), for a representative example. Cedille uses Unicode, so Cedille code largely matches the

```
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.

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 a . λ v . Λ P . λ c . λ e . c a v (v · P c e),
    Nil = Λ P . λ c . λ n . n .

rec Vector (A : ⋆) : (n : Nat) |
 Cons : ∀ n : Nat . A → Vector n → Vector (S n) ,
 Nil : Vector Z =
  ∀ P : Π n : Nat . Vector n → ⋆ .
    (∀ n : Nat . Π a : A . Π v : Vector n .
     P n v → P (S n) (Cons -n a v)) →
    P Z Nil →
    P n self
  with
    Cons = Λ n . λ a . λ v . Λ P . λ c . λ e .
             c -n a v (v · P c e) ,
    Nil = Λ P . λ c . λ n . n .
```

Fig. 14. Cedille definitions of three standard datatypes.

mathematical syntax we have already considered. The constructor sets must first be typed, assuming the kinding `Nat : ⋆`. Next, the body is kinded, assuming that `self` has the recursive type (applied to any indices). So in this case, `self` is assumed to have type `Nat` when kinding the body. Finally, each constructor definition (the equations following the `with` keyword) must be typed. Cedille types a definition $c = t$ by checking that t has type T under the assumption that the recursively defined type is equal to its body, with the `self` variable explicitly ι-abstracted. There a variety of other small checks to perform as well (the conditions imposed by Ctors_X, the starting condition for kinding using the top type $\mathscr{U}_\kappa$, and a few others).

Cedille implements local type inference to cut down on the number of annotations required in terms (Pierce & Turner, 2000). We are either checking a term against a type or a type against a kind, or else trying to synthesize a type for a term or a kind for a type. Cedille seeks to instantiate ι-types introduced by recursive definitions either when checking against an introduction form (an implicit or explicit λ-abstraction), or when a type is synthesized for the head of an application. The former is intended just for typing constructor definitions, while the latter is for use there as well as when terms of recursive type are eliminated. This simple scheme appears sufficient so far to avoid any explicit reasoning about dependent intersections on the part of the user.

```
Eq ⇐ Π A : ⋆ . A → A → ⋆ =
  λ A : ⋆ . λ a : A . λ b : A . ∀ P : A → ⋆ . P a → P b .

refl ⇐ ∀ A : ⋆ . ∀ a : A . Eq · A a a =
  Λ A . Λ a . Λ P . λ u . u .
```

Fig. 15. Leibniz equality.

Cedille implements an algorithmic conversion relation based on normalizing term and type expressions. While this is not strictly speaking justified by Theorem 13 above, I conjecture that the proof may, with some effort, be adapted to show not just consistency but normalization. I have not invested this effort so far, for the following reason. Consistency is the crucial property for a type theory, as it tells us that we may safely avoid reducing some terms, and still know that they would reduce to canonical values. Normalization is nice in theory, but in practice the enormous computational complexity of functions, which can be written in type theory means that there are terms which will cause type checking to run so long as to be practically indistinguishable from non-termination. So any type theory that truly requires a bound on the time required to check terms will have to do more than just prove normalization (and such theories have, of course, been developed; e.g., Hofmann (2000)).

Cedille itself is coded in Agda. Agda is a dependently typed programming language under development (in its Agda 2 form) for around a decade. The main implementation was done by Ulf Norell, with subsequent additions from other researchers (Norell, 2007). Cedille makes use of the Iowa Agda Library, an alternative standard library I am developing, currently at a little under 5,000 lines of code. This library is the basis for my book on Agda (Stump, 2016). While I have not verified deep properties of the implementation using Agda's theorem-proving capabilities, I have expressed a number of simple program invariants using dependent types, and used type-level computation to simplify and condense some of the code.

8 Basic examples

Now, let us consider some examples demonstrating the features of CDLE, as implemented in Cedille.

8.1 Inductive reasoning about Church-encoded numbers

First, let us show that we can indeed perform dependent eliminations with Church-encoded numbers, by proving a basic inductive fact about addition. We can define Leibniz equality in the usual way, as shown in Figure 15. The statements shown in the figure are of the form $x \Leftarrow e = e'$, for checking expression e' against classifier (type or kind) e, and then adding a definition of x to equal e' to the global context. So here we define the type `Eq` for Leibniz equality in a standard way, and then give an inhabitant `refl` for reflexive equalities. As noted above, more complex forms of equality can also be defined using constructor-constrained recursive types, but this is sufficient here.

```
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 .

True ⇐ ⋆ = ∀ X : ⋆ . X → X .
triv ⇐ True = Λ X . λ x . x .

False ⇐ ⋆ = ∀ X : ⋆ . X .
```

Fig. 16. Booleans, and true and false types.

```
tt-not-equal-ff ⇐ Eq · Bool tt ff → False =
  λ u . u · (λ b . λ v .
              (↑ X . b · (λ b . X) : (∗ → ∗ → ∗)) · True · False)
       triv .
```

Fig. 17. Proof that boolean true is not equal to boolean false.

If we define addition in the standard way, we can then write the following very basic inductive proof about it, showing that $x + 0 = x$ for all x:

```
add ⇐ Nat → Nat → Nat =
  λ n . λ m . n · (λ n : Nat. Nat) (λ n . S) m .

addZ ⇐ Π x : Nat . Eq · Nat (add x Z) x =
  λ x . x · (λ n : Nat . Eq · Nat (add n Z) n)
          (λ n . λ u . Λ P . λ v . u · (λ x : Nat . P (S x)) v)
          (refl · Nat -Z) .
```

As is well known, this theorem does not hold simply by reducing `add x Z`, because `add` iterates on `x`. The term we have given as the definition for `addZ` has an induction on `x` matching this iteration. The induction is carried out by a dependent elimination, where `x` is applied, in the second line, to the predicate to be proved. The third line of the code gives the step case, where `u` is the proof of `P (add n Z)`, for an arbitrary predicate P postulated by Leibniz equality, and the return value is then the proof of `P n`. The fourth line gives the base case, which follows trivially using conversion.

8.2 True not equal to false

Figure 16 defines the `Bool` datatype using a constructor-constrained recursive type to support dependent eliminations on booleans. The figure also gives standard impredicative definitions for the types `True` and `False`.

Using these definitions, we may then write the proof in Figure 17 deriving a contradiction from an assumption that `tt` (boolean true) equals `ff`, where the notion of equality is again Leibniz equality. Note that this fact is not provable for Church-encoded booleans in Coq, for instance (Werner, 1992). Here, we instantiate the variable `P` from the Leibniz equality with a predicate which uses lifting (the ↑ expression) to compute the type `False` from boolean `ff` and `True` from `tt`. This

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

interp ⇐ tp → ⋆ =
  λ T . ↑ Y . (T · Y) : ((∗ → ∗ → ∗) → ((∗ → ∗) → ∗) → ∗) ·
                (λ A . λ B . A → B) ·
                (λ F . ∀ C : ⋆ . F · C) .
```

Fig. 18. Higher order encoding of System F types, and its interpretation.

allows us to cast `triv` from type `True` to type `False`. Using large eliminations is the standard way to prove this fact with primitive datatypes, but large eliminations are not available for lambda encodings in other theories. Lifting in CDLE makes this possible. In the Cedille implementation, we must explicitly introduce the type variable `X` (immediately following the ↑ sign), which the term being lifted will use to indicate the positions in the type which are to be lifted to the kind ⋆.

8.3 Higher order encoding of System F types

Let us see how CDLE allows large eliminations with higher order encodings of datatypes. We would like to represent the types of System F (constructed by universal quantification and function-space formation from type variables), using a higherorder encoding. So we do not want to encode the universally bound variables as de Bruijn indices, for example. Rather, we will use CDLE's variables to represent these System F type variables.

Figure 18 declares the type `tp`, of kind ⋆ to represent System F types. The type says that for all types `X`, a `tp` can take in a function of type `X → X → X` and also one of type `(X → X) → X`, and return a value of type `X`. The first function is the one to use if the `tp` is representing an arrow type (and then the values computed for the domain and range types will be supplied as the two arguments of type `X`). The second function takes in a `X → X` function and returns a value of type `X`. Here, we see the higher order aspect of the encoding. Due to the negative occurrence of `X` in the domain type `X → X` of this type, this would not be allowed as part of an inductive datatype definition in Coq or Agda, though it could be defined in the pure λ-calculus fragment of Coq.

But Coq does not have anything like the lifting operation of CDLE, and so one could not write the type-level function `interp-tp` of Figure 18, which interprets a `tp` as the corresponding actual type of CDLE. This definition lifts the `tp t` to the type level, and then applies it to functions, which compute either the arrow type or the universally quantified type. In the latter case, the higher order encoding presents us with `F` of type `⋆ → ⋆`, which maps any input type to the interpretation of the encoded body of the universal type. So, we just introduce a universally quantified `C` and apply `F` to that, to compute the interpretation.

For example, we may define the type of polymorphic identify functions as an inhabitant of `tp`:

```
polyid-t ⇐ tp = Λ X . λ arrow . λ forall .
                forall (λ x . arrow x x) .
```

If we interpret this value using our `interp` function, Cedille tells us we get

```
∀ C : ⋆ . (C → C)
```

To demonstrate the point that we can eliminate data at multiple levels of the type theory, let us also define a function to compute the size (as a natural number) of a `tp`:

```
size ⇐ tp → Nat =
  λ T : tp .
    T · Nat
      (λ m . λ n . S (add n m))
      (λ s . S (s one)) .
```

Cedille reports that normalizing `size polyid-t` results in Church-encoded four:

```
λ s . λ z . (s (s (s (s z))))
```

It is important to note that in this example, we are using lifting only. Constructor-constrained recursive types require positivity, which would not hold here. Even though we do not get a dependent elimination principle for a datatype like `tp`, we still gain extra expressive power in CDLE over other impredicative type theories like that of Coq, due to CDLE's lifting operation.

8.4 Strong Σ-types

Strong Σ-types can be defined in CDLE, using constructor-constrained recursive types as shown in Figure 19. As above, we define the type as its own induction principle. Defining first and second projections is then straightforward. For `fst`, we instantiate the predicate variable `P` with a trivial predicate that always returns `A` for any input. But for `snd`, we use a non-trivial predicate, so that the type which `λ a . λ b . b` must inhabit is

```
Π a : A . Π b : (B a) . (B (fst (mksigma a b)))
```

This type is convertible with just

```
Π a : A . Π b : (B a) . (B a)
```

which is inhabited, as required, by `λ a . λ b . b`. This is a nice example of how type refinement, as implemented in languages with dependent pattern matching (Coquand, 1992), is also available in CDLE.

One might be concerned that despite the claimed Theorem 13 above, definability of strong Σ-types could somehow put CDLE afoul of Coquand's result that the Calculus of Constructions with strong Σ-types is inconsistent (Coquand, 1986). But the system considered by Coquand allows the formation of large Σ-types $\Sigma X : \kappa.\kappa'$, which are crucially used in the proof of inconsistency. In contrast, the Σ-types defined in Figure 19 are small, so Coquand's result does not apply.

```
rec Sigma (A : ⋆)(B : A → ⋆)
| mksigma : Π a : A . B a → Sigma =
  ∀ P : Sigma → ⋆ .
    (Π a : A . Π b : B a . P (mksigma a b)) → P self
with
  mksigma = λ a . λ b . Λ P . λ c . c a b .

fst ⇐ ∀ A : ⋆ . ∀ B : A → ⋆ . Sigma · A · B → A
  = Λ A . Λ B . λ p .
      p · (λ _ : Sigma · A · B . A) (λ a . λ b . a) .

snd ⇐ ∀ A : ⋆ . ∀ B : A → ⋆ . Π p : Sigma · A · B .
         B (fst · A · B p)
  = Λ A . Λ B . λ p .
      p · (λ p : Sigma · A · B . B (fst · A · B p))
```

Fig. 19. Strong Σ-types.

```
tp ⇐ ⋆ = ∀ X : ⋆ . (X → X → X) → X → X .
arrow ⇐ tp → tp → tp =
  λ T1 . λ T2 . Λ X . λ a . λ b . a (T1 · X a b) (T2 · X a b) .
base ⇐ tp = Λ X . λ a . λ b . b.

trm ⇐ tp → ⋆ =
 λ T : tp.
 ∀ X : tp → ⋆ .
  (∀ T1 : tp . ∀ T2 : tp . X (arrow T1 T2) → X T1 → X T2) →
  (∀ T1 : tp . ∀ T2 : tp . (X T1 → X T2) → X (arrow T1 T2)) →
  X T.
```

Fig. 20. Typed higher order abstract syntax for STLC terms.

8.5 Statically typed higher order abstract syntax

The higher order encoding of System F types in Section 8.3 may leave some readers wondering if typed abstract syntax can be represented in a similar way. For System F types have only a trivial kinding structure, which the encoding of Section 8.3 thus did not have to take into account. Figure 20 gives a higher order encoding of the typed syntax for simply typed lambda calculus. The figure first defines `tp`, representing the simple types with a single base type. Constructors `arrow` and `base` for this type are then defined. Then `trm` is defined, of kind `tp → ⋆`. Then, `trm T` is the type for representations of simply typed terms t with type represented by `T`. The definition of `trm` first takes in the type `T` for this family of terms, and then a type `X` indexed by `tp`. The cases for application and λ-abstraction come next. Note that the λ-abstraction case is higher order: Given a function from `X T1` to `X T2`, the function supplied for this case must deliver a value of type `X (arrow T1 T2)`.

Figure 21 defines example terms `id` and `test`, of type `trm abb` and `trm test-tp`, respectively, where `abb` and `test-tp` represent $b \rightarrow b$ and $(b \rightarrow b) \rightarrow (b \rightarrow b)$, respectively. The `trm id` represents $\lambda x.x$, and `test` represents $\lambda s.\lambda z.s\ (id\ (s\ z))$. Because Cedille's type inference is currently just basic local type inference, quite a few erased arguments must be supplied. Improving this situation is future work.

```
abb ⇐ tp = arrow base base.
id ⇐ trm abb = Λ X . λ a . λ l . l -base -base (λ x . x) .

test-tp ⇐ tp = arrow abb abb .
test ⇐ trm test-tp =
  Λ X . λ a . λ l .
    l -abb -abb (λ s .
    l -base -base (λ z .
    a -base -base s (a -base -base (id · X a l) (a -base -base s z)))).
```

Fig. 21. Example terms and types in the encoding.

```
interp-tp ⇐ tp → ⋆ =
  λ T : tp . (↑ X . T · X : ((∗ → ∗ → ∗) → ∗ → ∗))
                · (λ T1 : ⋆ . λ T2 : ⋆ . T1 → T2)
                · True.

interp-trm ⇐ ∀ T : tp . trm T → interp-tp T =
  Λ T . λ t .
    t · (λ T : tp . interp-tp T)
      (Λ T1 . Λ T2 . λ f . λ a . f a)
      (Λ T1 . Λ T2 . λ r . λ x . r x).
```

Fig. 22. Interpreting encoded types and terms.

The erasure of `test`, however, is the following, which does encode the test term as expected:

`λ a . λ l . (l (λ s . (l (λ z . (a s (a (id a l) (a s z))))))`

Finally, Figure 22 defines a very simple interpreter for `trm`. This is `interp-trm`. To express its type, we first define `interp-tp` using lifting. This is a simpler version of the `interp-tp` function we saw in Figure 18 for System F types. The definition of `interp-trm` uses `interp-tp` (of Figure 22) and dependent types to express the idea that the interpretation of a `trm T` is a CDLE term of CDLE type `interp-tp T`. The interpretation is then completely direct: (typed) application is interpreted as application, and λ-abstraction is interpreted as λ-abstraction. Interpreting `test` results in `λ s . λ z . (s (s z))`, which as expected, has evaluated the term as part of interpreting it (cf. normalization by evaluation (Berger & Schwichtenberg, 1991)).

9 Formatted printing with local definitions

Let us now consider a more complex case of large eliminations with higher order encodings: adding local definitions to format specifiers for formatted printing as with `printf`. Typing `printf` is now a standard and quite appealing example of dependently typed programming, introduced by Augustsson (1998). Here, we will allow format specifiers – for which we will use a dedicated datatype, not a format string – to contain two types of `let`-declarations. `flet x y` will specify that the arguments required by `x` should be input to the call to `format`, and then the resulting string which is computed for `x` will be substituted into `y`. More dynamic is `fletd x y`, which just substitutes `x` into `y`, and thus could duplicate requirements

```
{-# OPTIONS --no-positivity-check #-}
module format-ilet where

open import lib

data formatti : Set where
  iarg : formatti
  inone : formatti
  iapp : formatti → formatti → formatti
  ilet : formatti →
          (formatti → formatti) → formatti

bitstr : Set
bitstr = 𝕃 𝔹

data formati : formatti → Set where
   farg : formati iarg
   fapp : {a b : formatti} →
            formati a → formati b →
            formati (iapp a b)
   flet : {a b : formatti} → formati a →
            (formati inone → formati b) →
            formati (iapp a b)
   fletd : {a : formatti}
           {b : formatti → formatti} →
              formati a →
              (formati a → formati (b a)) →
              formati (ilet a b)
   fbitstr : bitstr → formati inone

flit : 𝔹 → formati inone
flit b = fbitstr [ b ]
```

Fig. 23. Datatype definitions for `format` with local definitions, in Agda.

for arguments (leading to additional arguments to the call to `format`). We will print lists of booleans rather than lists of characters, to avoid dependence on a primitive type of characters.

9.1 Agda implementation

Figures 23 and 24 give Agda code for this example (based on the Iowa Agda Library). The approach used here is not the one which would typically be adopted in Agda, because it requires that we disable Agda's positivity checker to use a higher order encoding, thus sacrificing the termination property which Agda seeks to guarantee. The first point of showing this solution is explanatory: Hopefully, it will orient readers familiar with Agda or Haskell, for the subsequent Cedille implementation (Section 9.2). Second, though, it is meant to highlight that this implementation technique – which results in a reasonable solution for this novel problem – is not available in Agda without compromising termination. Of course, the example itself could be implemented using other methods, such as de Bruijn

```
format-th : formatti → Set → Set
format-th iarg r = 𝔹 → r
format-th inone r = r
format-th (iapp i i') r =
  format-th i (format-th i' r)
format-th (ilet i i') r = format-th (i' i) r

format-t : formatti → Set
format-t i = format-th i bitstr

formath : {i : formatti} → formati i →
          {A : Set} → (bitstr → A) → format-th i A
formath farg f x = f [ x ]
formath (fapp i i') f =
  formath i (λ s → formath i' λ s' → f (s ++ s'))
formath (flet i i') f =
  formath i (λ s → formath (i' (fbitstr s)) f)
formath (fletd i i') f = formath (i' i) f
formath (fbitstr s) f = f s

format : {i : formatti} → formati i → format-t i
format i = formath i (λ x → x)
```

Fig. 24. Formatted printing with local definitions, in Agda.

indices for representing bound variables. But with Cedille, the possibility of a higher order solution to this problem is available, without compromising logical soundness.

To turn to the code of Figure 23, we have a datatype `formatti` which will describe the argument requirements of format specifiers. A format specifier can require an argument (`iarg`), no argument (`inone`), or appended requirements (`iapp`), or requirements governed by a dynamic `let` (`ilet`). The type `formati` is the type for the actual format specifiers. The interesting cases are for `flet` and `fletd`, where we use higher order encoding. In the static case (`flet`), we have a function from inputs with argument requirement `inone` to outputs with requirement `b`, and in the dynamic case, the requirement goes from `a` to `b a`. The types of the inputs to these constructors use the `formati` in negative positions, and hence would be disallowed by Agda without the initial pragma disabling the positivity check.

The function `format-t` (Figure 24) computes the type for `format` from an argument requirement (of type `formatti`), while `format` itself (or rather, the helper function `formath`) is defined by recursion on the format specifier (of type `formati`). The `formath` function uses a continuation so that interpretation of the format directive can take place before any input arguments are required (by an `farg` format specifier).

For a test case, we can define

```
testi : formatti
testi = ilet (iapp iarg (iapp inone inone))
             (λ x → iapp x (iapp inone x))

test : formati testi
test = fletd (flet farg (λ j → fapp j j))
             (λ i → fapp i (fapp (flit tt) i))
```

The format specifier `test` says that we want to print a string consisting of `i` followed by a boolean literal `tt` (`flit tt`), and then `i` again, where `i` is dynamically defined to be the static definition `flet farg (λ j → fapp j j)`. This requests one argument to be named `j`, and then produces `j` appended to `j`. Agda's normalizer reports that as expected, `format test` normalizes to

```
λ x x₁ → x :: x :: tt :: x₁ :: x₁ :: []
```

9.2 *Cedille implementation*

Let us now implement this example in Cedille. It is worth emphasizing that no modification to CDLE is required (whereas we had to disable positivity checking for the example to type check in Agda). We should also note that similarly to the example of representing the types of System F (Section 8.3), we will use higher order encodings that prevent us from using constructor-constrained recursive types. Lifting, however, is still available, and is sufficient for this example. First, we must declare the type `formatti` for argument requirements. We break this into two parts: a type-level function `formatto`, and then the universal type `formatti`:

```
formatto ⇐ ★ → ★ =
  λ X : ★ . X → X → (X → X → X) → (X → (X → X) → X) → X .
formatti ⇐ ★ = ∀ X : ★ . formatto · X .
```

We can define abbreviations for the constructors of this type, the last of which is the most interesting, since it is there that higher order encoding shows up

```
iarg ⇐ formatti = Λ X . λ a . λ n . λ p . λ l . a .
inone ⇐ formatti = Λ X . λ a . λ n . λ p . λ l . n .
iapp ⇐ formatti → formatti → formatti =
  λ x . λ y .
  Λ X . λ a . λ n . λ p . λ l .
    p (x · X a n p l) (y · X a n p l).
ilet ⇐ formatti → (∀ X : ★ . X → formatto · X) → formatti =
  λ u . λ f .
  Λ X . λ a . λ n . λ p . λ l .
    l (u · X a n p l) (λ x . f · X x a n p l) .
```

The argument `f` to `ilet` takes in an `X` and returns a `formatto · X`, for any type `X`. This can be viewed as saying that `f` is a member of an extension of the datatype `formatti` with a new constructor (since `f` requires a value of type `X` for this constructor).

We elide a few easy definitions (Church-encoded booleans, an append operation on lists, and the `bsingleton` function for creating a singleton list from boolean input). Next comes the type `formati` for format specifiers. Again, we break it into two parts, shown in Figure 25.

```
formato ⇐ (formatti → ⋆) → formatti → ⋆ =
  λ X : formatti → ⋆ . λ i : formatti.
    X iarg →
    (∀ a : formatti . ∀ b : formatti.
       X a → X b → X (iapp a b)) →
    (∀ a : formatti . ∀ b : formatti.
       X a → (X inone → X b) → X (iapp a b)) →
    (∀ x : formatti . ∀ F : ∀ X : ⋆ . X → formatto · X.
       X x →
      (X x →
        X (Λ X . λ a . λ n . λ p . λ l . F · X (x · X a n p l)
                                          a n p l)) →
       X (ilet x F)) →
    (bitstr → X inone) →
    X i .
formati ⇐ formatti → ⋆ =
  λ i : formatti . ∀ X : formatti → ⋆ . formato · X i.
```

Fig. 25. The type `formati` for format strings.

```
k ⇐ □ = ⋆ → ⋆ .
Fa ⇐ k = λ r : ⋆ . (Bool → r) .
Fn ⇐ k = λ r : ⋆ . r .
Fp ⇐ k → k → k = λ f : k . λ g : k . λ r : ⋆ . f · (g · r) .
Fl ⇐ k → (k → k) → k = λ f : k. λ g : k → k . λ r : ⋆ . (g · f · r) .

format-th ⇐ formatti → ⋆ → ⋆ =
  λ i : formatti .
    ↑ X . i · (X → X) : ((∗ → ∗) →
                         (∗ → ∗) →
                         ((∗ → ∗) → (∗ → ∗) → (∗ → ∗)) →
                         ((∗ → ∗) → ((∗ → ∗) → (∗ → ∗)) → (∗ → ∗)) →
                         (∗ → ∗)) · Fa · Fn · Fp · Fl .
```

Fig. 26. Definition of the helper function computing the type for a call to `format` from a format string.

The type for the dynamic `let` (beginning on the eighth line in the figure) is the trickiest, since the argument requirement for the body of the let depends on the argument requirement `x` for the let's `definiens`. But our definition of `ilet` requires a `F` that can be extended with the value for a variable, which enables expression of this dependence. For space reasons, we must omit the definitions of constructors for this type, and turn to the definition of `format-th`. To make reasoning about this definition more manageable, we pre-define the type-level functions that will be used for the different cases of a `formati` term. The code is shown in Figure 26. The crucial point, of course, is to use lifting to define the type by higher order iteration on the input of type `formatti`.

It is convenient to break out the return type for `formath` as a separate definition (`formathr`), and then we have the code for `formath` itself, shown in Figure 27. Instead of recursive calls, the higher order iteration on `a` of type `formati` presents us with results `r` of recursive calls, in each case. As we are computing a higher order function (of type `formathr`), these results are themselves functions, which we call

```
formathr ⇐ formatti ➝ ⋆ =
  λ i : formatti . ∀ A : ⋆ . (bitstr ➝ A) ➝ format-th i · A .

formath ⇐ ∀ i : formatti . formati i ➝ formathr i =
  Λ i . λ x . x · formathr
                (Λ A . λ f . λ b . f (bsingleton b))
                (Λ a . Λ b . λ r . λ r2 .
                 Λ A . λ f . r · (format-th b · A)
                               (λ s . r2 · A
                                          (λ s' . f (append · Bool s s'))))
                (Λ a . Λ b . λ r . λ r2 .
                 Λ A . λ f . r · (format-th b · A)
                               (λ s . r2 (Λ A . λ f . f s) · A f))
                (Λ x . Λ F . λ r . λ r2 .
                 Λ A . λ f . r2 r · A f)
                (λ s . Λ A . λ f . f s) .
```

Fig. 27. Definition of the helper function for |format—.

```
format-t ⇐ formatti ➝ ⋆ = λ i : formatti . format-th i · (CList · Bool) .

format ⇐ ∀ i : formatti . formati i ➝ format-t i =
  Λ i : formatti . λ t : formati i .
    formath -i t · (CList · Bool) (toCList · Bool) .
```

Fig. 28. The definition of the `format` function and its return type.

with a continuation to obtain the printing function for the part of the format string from which the result was iteratively computed.

The final definition of the `format` function and its return type is then the following, where for the outermost continuation we use a function `CList` which converts Parigot-encoded to Church-encoded lists. This just makes the output produced by Cedille's interpreter more readable in this case. The code is in Figure 28. We can use Cedille's normalizer with the same test as we used for the Agda version, to obtain

```
(λ b' . λ b'' .
 λ c . λ e .
  (c b' (c b' (c (λ a' . λ b''' . a') (c b'' (c b'' e))))))
```

This is indeed a Church-encoded version of the answer we computed with the Agda implementation (at the end of Section 9.1).

In typing the `formath` term of Figure 27, several conversions dealing with lifting are required. These are the last two conversions shown in Figure 5 above. Let us see briefly how these arise. In typing the cases for `fapp` and `flet`, Cedille must check that the type `format-th (iapp a b) · A` is convertible with

```
formath-th a · (format-th b · A)
```

The latter type arises from the terms `r · (format-th b · A)` in both cases, while the former type is the one required by the elimination of the format specifier `x`. Since lifting introduces new lifting redexes for arguments to a head variable, normalizing the first type would, without the η-contraction lifting conversion of Figure 5 (the first

```
((↑ X . (λ a . λ n . λ p . λ l . (F (x a n p l) a n p l))
    : ((∗ → ∗) → (∗ → ∗) → ((∗ → ∗) → (∗ → ∗) → (∗ → ∗)) →
       ((∗ → ∗) → ((∗ → ∗) → (∗ → ∗)) → (∗ → ∗)) → (∗ → ∗)))
   · Fa · Fn · Fp · Fl · A)
```

Fig. 29. A lifting type arising in the `flet` case.

conversion in the last row of the figure), produce what is essentially an η-expanded version of a to be lifted.

The last conversion of Figure 5 is needed for the `fletd` case, where Cedille must check that `format-th (ilet x F) · A` is convertible with the type shown in Figure 29. Again, due to the way lifting produces new lifting redexes, normalization of the first type would result in a lifting of `F` being applied to a lifting of `x`. Those two uses of the lifting operation need to be consolidated at the top level of the term, in order to match the type of Figure 29. This is what the final conversion of Figure 5 does.

10 Related work

We compare the approach of CDLE with some recent works. In "The Gentle Art of Levitation" (*Levitation*), Chapman *et al.* (2010) present a closed type theory where inductive datatypes are implemented using a universe of datatype descriptions, which, cleverly, is itself given a datatype description. CDLE does not include a universe, although as a closed type theory it would make sense to consider extending it with one. *Levitation* is concerned with encoding universes of datatypes as datatypes themselves, but not with foundations of induction. Indeed, least fixed points of functors (polynomial, then strictly positive), and associated induction principles, are included as primitives of the theory. Some primitive datatypes are included as part of the type theory; indeed, *Levitation* affirms "We cannot dispose of data altogether!" (Section 4.1). In contrast, CDLE defines data as their own induction principles, and hence reduces induction to the underlying impredicative type theory; CDLE does not include any induction principle as primitive, nor any datatypes. One could imagine attempting to replace the primitive induction principle used in *Levitation*, with induction as derived in CDLE. But *Levitation*'s self-describing universe construction crucially relies on a predicative universe hierarchy, which we have omitted here in CDLE. *Levitation* also affirms, citing (Geuvers, 2001): "An impredicative Church-style encoding of datatypes is not adequate for dependently typed programming, as although such encodings present data as non-dependent eliminators, they do not support dependent induction." CDLE overturns the received wisdom that Geuvers's Theorem implies the inadequacy of lambda encodings for dependent type theory. The theorem only shows this for second-order dependent type theory, leaving open the possibility that extensions to that theory could be adequate – as we have seen with CDLE.

It is worth noting at this point that while universe polymorphism might appear to allow data to be eliminated across levels – thus solving the same problem as CDLE's proposed lifting operator – such quantifications are predicative (Harper

& Pollack, 1989). Indeed, a universe-polymorphic type cannot itself be in any of the universes whose levels it quantifies over, and in Agda is considered to be at ordinal level ω. The computational power of predicative polymorphism is quite limited compared with impredicative polymorphism, as analyzed by Leivant (1991). In contrast, CDLE avoids adding a hierarchy of universes by using a novel lifting operator to move expressions from the term level to the type level of the language.

Altenkirch *et al.* (2010) pursue a related goal to *Levitation*'s in "PiSigma: Dependent Types without the Sugar" (*PiSigma*): show how to define datatypes (inductive and co-inductive, indexed) and other central constructs of type theory in a minimalistic core language. Special care is paid to the control of reduction during type checking, for recursively defined types and terms, using *lifted types*, inhabited by suspended terms. These are different from CDLE's lifting types, which actually raise terms to the type level. *PiSigma* includes the $\star : \star$ principle, and so levels cannot be distinguished. Furthermore, general recursion is allowed, and questions both of termination and meta-theory in general are deferred to later work. This is in contrast with CDLE, which is proved logically sound, and in which a sound notion of induction is defined.

Let us consider several works more focused on semantics and induction. In "Internalizing Relational Parametricity in the Extensional Calculus of Constructions" (*Internalizing*), Krishnaswami and Dreyer (2013) develop a version of the Calculus of Constructions with a built-in equality type that enjoys equality reflection: if an equality is provable then it can be used definitionally – the central idea of extensional Martin-Löf type theory (Martin-Löf, 1984). They devise a relationally parametric realizability model, and show how this model validates various extensions of their syntactic theory, includes strong sum types. But these are true extensions: the type theory proposed by *Internalizing* does not actually allow typing strong sum types, for example. In contrast, we saw above (Section 8.4) that strong sigma types can be defined within CDLE (without any extensions). The same is true for natural-number induction, which again in *Internalizing* is shown consistent with their proposed syntactic theory, but has to be added as an extension to the theory. On the other hand, *Internalizing* gives examples of (semantically) relating extensionally equal terms, which the semantics for CDLE given in Section 5 above would distinguish (cf. Section 5.4 of *Internalizing*). Developing extensional models of CDLE, perhaps along the lines of *Internalizing* or perhaps following the "extensional collapse" approach of Tannen and Coquand (1988), remains to future work. A similar goal with some stronger results – notably that every indexed functor has an initial algebra – was achieved by Atkey *et al.* (2014).

The paper "Fibrational Induction Rules for Initial Algebras" of Ghani *et al.* (2010) proposes a general induction rule for arbitrary functors with initial algebras. The development is categorical, using the idea of a fibration to generalize the logical notion of predicate. The paper is focused on categorical semantics, and explicitly avoids impredicativity. In contrast, the present work on CDLE develops a new impredicative type theory, with a concrete realizability semantics. The deeper insights into the nature of induction arising from categorical study could provide more refined analysis of the forms of induction possible in CDLE, but this must remain to future work.

The lifting types of CDLE and its realizability semantics may put one in mind of "Realizability and Parametricity in Pure Type Systems" (*Realizability*), by Bernardy and Lasson (2011). In this elegant paper, the authors seek to shed light on the relationship between realizability and parametricity, by formally defining both as relations on terms in a first-level Pure Type System (PTS), using a second-level PTS viewed as a logic for the first one. Terms in the first-level PTS are lifted to the second-level PTS, which may then express statements about them. But in *Realizability*, lifting, like realizability and parametricity, are expressed as meta-level operations on PTS terms. In contrast, CDLE's lifting types are part of the type theory, which enables computation of types from terms, within the theory. For the PTS corresponding to CC, for example, this is not possible. The ideas of *Realizability* may, however, shed further light on CDLE's lifting types, as well as on the best approach to formalizing CDLE's meta-theory. Note, finally that while PTSs are expressed using a unified syntax for expressions (instead of syntactically different classes for terms, types, and kinds as in CDLE), some form of lifting is still required to lift a typing judgment. In *Realizability* this is at the meta-level, while with CDLE it is in the theory.

Next, let us compare the present approach to the works, already mentioned in Section 1.1, based on adding primitive inductive types to existing type theories (i.e., Coquand & Paulin (1988), Pfenning & Paulin-Mohring (1989), Werner (1994)). With primitive inductive types, one should determine some set of inductive types, which will be accepted by the type theory, if one wishes to be able to prove any general results about the addition (for an open theory) or declaration (for a closed one) of a new inductive type. For example, CIC restricts attention to inductive types generated by strictly positive functors (see Werner (1994), Definition 2.7). In CDLE, there are more options: if one needs induction and uses constructor-constrained recursive types, then we require only positivity. If induction is not needed, then there are no restrictions at all on the functors one may use, for Church-encoded datatypes (for the Parigot-encoding, of course, positive recursive types are needed). Describing a class of inductive datatypes is not a simple matter. Indeed, *Levitation* proposes an intricate solution to the problem. In Werner's dissertation, one finds quite long typing rules with lots of vector notation, to handle the variable-arity nature of both the inductive types and their constructors (and constructors' types). None of this is needed in CDLE. Finally, with primitive inductive types, one must augment the reduction relation, necessitating a new proof of confluence for reduction. With CDLE, the reduction relation is just standard β-reduction on untyped terms, so there is no new confluence theorem to prove.

Finally, let us compare with a few works on advanced representations of syntax, such as typed or higher order abstract syntax. The idea of using higher order representations in typed lambda calculus can be traced back to Church's Simple Theory of Types, where universal quantification, for example, is defined to be the application of a function $\Pi_{o(oa)}$ (expressing universality of a propositional function) to a lambda abstraction (Church, 1940). The term "higher order abstract syntax" was coined by Pfenning and Elliott (1988) for the idea of representing the syntax of various object languages using typed λ-abstractions. Many works have explored the idea of shallowly embedding object-language syntax into meta-language syntax. For one example, Mogensen (1992) proposed a shallow embedding of the syntax of pure lambda calculus in itself. For another, Carette *et al.* (2009) in "Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages" show how to define various meta-programs (interpreters, compilers, and more) by semantically embedding the syntactic structure of an object language into a meta-language. This gives a typed (shallowly embedded) syntax similar to the example of Section 8.5. As Chlipala (2008) points out, though, directly mapping object-language types to meta-language-language types makes it more complex to perform type-level operations. In the cited paper, Chlipala proposes parametric higher order abstract syntax (PHOAS) for achieving some of the benefits of higher order encodings in Coq (which due to positivity restrictions on datatypes cannot support HOAS directly). Fegaras and Sheard (1996) proposed a method for programming with higher order representations in functional programming languages with primitive inductive datatypes. In contrast to these works, CDLE supports impredicatively typed higher order syntactic representations.

There are many works proposing two-layer schemes, where object-language expressions are represented in a typed lambda calculus with a relatively weak function space, while meta-programs are written in a more powerful lambda calculus with pattern matching on higher order representations (Schürmann *et al.*, 2001; Pientka, 2008; Poswolsky & Schürmann, 2009). In CDLE, in contrast, there is just a single (typed) language for higher order representations and programs over these. One might be concerned that the stronger function space of CDLE will spoil adequacy of encodings. For developments, where it is critical to capture exactly the object-language syntax instead of over-approximate it, one could use techniques such as proposed by Crary (2010) for representing linear logic proof terms in a non-linear meta-language (see also Polakow (2015)), namely Twelf (Pfenning & Schürmann, 1999).

11 Conclusion and future work

This paper has demonstrated that lambda encodings can be the basis for a dependent type theory supporting both induction and large eliminations, via the system CDLE and its implementation Cedille. Induction is enabled by the novel constructor-constrained recursive types $\nu\, X : \kappa \,|\, \Theta.\, T$, where Θ is a set of typing constraints on pure lambda terms that must be shown to hold for a top type $\mathscr{U}_\kappa$ and then be preserved by the body T. Under some light restrictions on the use of X in the types in Θ, these typings hold not just for the elements of the infinite sequence of increasing dependent types one can associate with the ν-type, but also for the limit of that sequence, which our semantics defines the meaning of the type to be. Large eliminations are enabled by a lifting construct $\uparrow_L t$, which lifts simply typed lambda terms to the type level. We gave a rather simple semantics for types in terms of complete lattices, and proved the typing rules of CDLE sound with respect to this semantics. Logical consistency of the system is then a corollary.

CDLE does not use a datatype system, and hence one could hope would be less cumbersome for formal meta-theoretic analysis. The most exciting application of CDLE is for dependently typed programming with higher order encodings. We gave several examples, including the non-trivial one of formatted printing with local definitions.

Programming with higher order lambda-encodings is a delicate matter (cf. Washburn & Weirich (2003) for one illuminating example). Much more exploration of this area is required. It would be interesting, for example, to see how much formalized meta-theory one could do using higher order encodings in Cedille. CDLE has shown that one can have dependent typing for higher order encodings, via lifting. Induction for such encodings, however, is prevented currently by the positivity requirement for constructor-constrained recursive types. Thus, devising *inductive* higher order encodings is the most important next direction for future work.

Acknowledgments

I deeply thank Madeliene Stump for her support while I was writing this paper. AMDG.

References

Abel, A. & Matthes, R. (2004) Fixed points of type constructors and primitive recursion. In *Proceedings of 18th International Workshop Computer Science Logic (CSL)*, Marcinkowski, Jerzy, & Tarlecki, Andrzej (eds), Lecture Notes in Computer Science, vol. 3210. Springer, pp. 190–204.

Altenkirch, T., Danielsson, N. A., Löh, A. & Oury, N. (2010) PiSigma: Dependent types without the sugar. In *Proceedings of 10th International Symposium (Flops) Functional and Logic Programming*, Blume, M., Kobayashi, N. & Vidal, G. (eds), Lecture Notes in Computer Science, vol. 6009. Springer, Berlin, Heidelberg, pp. 40–55.

Atkey, R., Ghani, N. & Johann, P. (2014) A relationally parametric model of dependent type theory. *Sigplan Not.* **49**(1), 503–515.

Augustsson, L. (1998) Cayenne – a language with dependent types. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Felleisen, M., Hudak, P., & Queinnec, C. (eds). ACM, pp. 239–250.

Berger, U. & Schwichtenberg, H. (1991) An inverse of the evaluation functional for typed lambda-calculus. In *Proceedings of the 6th Annual Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, pp. 203–211.

Bernardy, J.-P. & Lasson, M. (2011) Realizability and parametricity in pure type systems. In *Proceedings of 14th International Conference Foundations of Software Science and Computational Structures*, Lecture Notes in Computer Science, vol. 6604. Springer, Berlin, Heidelberg, pp. 108–122.

Böhm, C. & Berarducci, A. (1985) Automatic synthesis of typed lambda-programs on term algebras. *Theor. Comput. Sci.* **39**, 135–154.

Carette, J., Kiselyov, O. & Shan, C.-C. (2009) Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.* **19**(5), 509–543.

Chapman, J., Dagand, P.-É., McBride, C. & Morris, P. (2010) The gentle art of levitation. In *Proceeding of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Hudak, P. & Weirich, S. (eds), pp. 3–14. ACM.

Chlipala, A. (2008) Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Hook, J. & Thiemann, P. (eds), pp. 143–156. ACM.

Church, A. (1940) A formulation of the simple theory of types. *J. Symb. Log.* **5**(2), 56–68.

Church, A. (1941) *The Calculi of Lambda Conversion*, *Annals of Mathematics Studies*, vol. 6. Princeton University Press.

Constable, R. L., Allen, S. F., Bromley, M., Cleaveland, R., Cremer, J. F., Harper, R. W., Howe, D. J., Knoblock, T. B., Mendler, N. P., Panangaden, P., Sasaki, J. T. & Smith, S. F. (1986) *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall.

Coquand, T. (1986) An analysis of Girard's paradox. In *Proceedings, Symposium on Logic in Computer Science (LICS)*, IEEE Computer Society, pp. 227–236.

Coquand, T. (1992) Pattern matching with dependent types. *Electronic Proceedings of the 3rd Annual Bra workshop on Logical Frameworks*, Nordström, B., Petersson, K. & Plotkin, G. (eds). Available from Coquand's home page.

Coquand, T. & Paulin, C. (1988) Inductively defined types. In *Colog-88, International Conference on Computer Logic*, Martin-Löf, P. & Mints, G. (eds), Lecture Notes in Computer Science, vol. 417. Springer, pp. 50–66.

Crary, K. (2010) Higher-order representation of substructural logics. In *Proceeding of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Hudak, P. & Weirich, S. (eds). ACM, pp. 131–142.

Fegaras, L. & Sheard, T. (1996) Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Proceedings of 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Boehm, H.-J. & Steele, Jr., G. L. (eds). ACM , pp. 284–294.

Fortune, S., Leivant, D. & O'Donnell, M. (1983) The expressiveness of simple and second-order type structures. *J. ACM* **30**(1), 151–185.

Fu, P. & Stump, A. (2014) Self types for dependently typed lambda encodings. In *Proceedings of 25th International Conference on Rewriting Techniques and Applications (RTA) joint with the 12th International Conference on Typed Lambda Calculi and Applications (TLCA)*, Dowek, G. (ed), Lecture Notes in Computer Science, vol. 8560. Springer, pp. 224–239.

Geuvers, H. (2001) Induction is not derivable in second order dependent type theory. In *Typed Lambda Calculi and Applications (TLCA)*, Abramsky, S. (ed), Lecture Notes in Computer Science, vol 2044. Springer, pp. 166–181.

Ghani, N., Johann, P. & Fumex, C. (2010) Fibrational induction rules for initial algebras. In *Computer Science Logic, 24th International Workshop (CSL)*, Dawar, A. & Veith, H. (eds), Lecture Notes in Computer Science, vol. 6247. Springer, pp. 336–350.

Girard, J.-Y., Taylor, P. & Lafont, Y. (1989) *Proofs and Types*. New York, USA: Cambridge University Press.

Goguen, H., McBride, C. & McKinna, J. (2006) Eliminating dependent pattern matching. In *Algebra, Meaning, and Computation, Essays Dedicated to Joseph A. Goguen on the Occasion of his 65th Birthday*, Futatsugi, K., Jouannaud, J.-P. & Meseguer, J. (eds), pp. 521–540.

Harper, R. & Pollack, R. (1989) Type checking, universe polymorphism, and typical ambiguity in the calculus of constructions (draft). In *Tapsoft'89: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Barcelona, Spain, March 13–17, 1989, Volume 2: Advanced Seminar on Foundations of Innovative Software Development II and Colloquium on Current Issues in Programming Languages (CCIPL)*, Díaz, J. & Orejas, F. (eds), Lecture Notes in Computer Science, vol. 352. Springer, pp. 241–256.

Hofmann, M. (2000) Safe recursion with higher types and bck-algebra. *Ann. Pure Appl. Log.* **104**(1–3), 113–166.

Hofmann, M. & Streicher, T. (1998) The groupoid interpretation of type theory. In *Twenty-Five Years of Constructive Type Theory*. Oxford Logic Guides, vol. 36. Oxford University Press, pp. 83–111.

Koopman, P., Plasmeijer, R. & Jansen, J. M. (2014) Church encoding of data types considered harmful for implementations. In *Proceedings of 26th Symposium on Implementation and Application of Functional Languages (IFL)*, Plasmeijer, R. & Tobin-Hochstadt, S. (eds). ACM, pp. 4:1–4:12.

Kopylov, A. (2003) Dependent intersection: A new way of defining records in type theory. In *Proceedings of 18th IEEE Symp. Log. Comput. Sci. (LICS)*, pp. 86–95.

Krishnaswami, N. R. & Dreyer, D. (2013) Internalizing relational parametricity in the extensional calculus of constructions. In *Computer Science Logic 2013 (CSL)*, Rocca, S. R. D. (ed), LIPIcs, vol. 23. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, pp. 432–451.

Leivant, D. (1983) Reasoning about functional programs and complexity classes associated with type disciplines. In *Proceedings of 24th Annual Symposium on Foundations of Computer Science, 1983*. IEEE Computer Society, pp. 460–469.

Leivant, D. (1991) Finitely stratified polymorphism. *Inf. Comput.* **93**(1), 93–113.

Martin-Löf, P. (1984) *Intuitionistic Type Theory*. Napoli: Bibliopolis.

Mendler, N. (1988) *Inductive Definition in Type Theory*. PhD Thesis, Cornell University.

Meyer, A. R. & Reinhold, M. B. (1986) “Type” is not a type. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*. New York, USA: ACM, pp. 287–295.

Miquel, A. (2001) The implicit calculus of constructions extending pure type systems with an intersection type binder and subtyping. In *Typed Lambda Calculi and Applications*, Abramsky, S. (ed), Lecture Notes in Computer Science, vol. 2044. Springer, pp. 344–359.

Mogensen, T. Æ. (1992) Efficient self-interpretations in lambda calculus. *J. Funct. Program.* **2**(3), 345–363.

Norell, U. (2007) *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD Thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

Parigot, M. (1988) Programming with proofs: A second order type theory. In *European Symposium On Programming (ESOP)*, Ganzinger, H. (ed), Lecture Notes in Computer Science, vol. 300. Springer, pp. 145–159.

Parigot, M. (1989) On the representation of data in lambda-calculus. In *Computer Science Logic (CSL)*, Börger, E., Büning, HansKleine & Richter, M. (eds), Lecture Notes in Computer Science, vol. 440. Springer, pp. 309–321.

Pfenning, F. & Elliott, C. (1988) Higher-order abstract syntax. *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI)*, Wexelblat, R. L. (ed). ACM, pp. 199–208.

Pfenning, F. & Paulin-Mohring, C. (1989) Inductively defined types in the calculus of constructions. In*Proceedings of 5th International Conference Mathematical Foundations of Programming Semantics*, Main, M. G., Melton, A., Mislove, M. W., & Schmidt, D. A. (eds), Lecture Notes in Computer Science, vol 442. Springer, pp. 209–228.

Pfenning, F. & Schürmann, C. (1999) System description: Twelf - a meta-logical framework for deductive systems. In *Proceedings of 16th International Conference on Automated Deduction Automated Deduction - Cade-16*, Ganzinger, H. (ed), Lecture Notes in Computer Science, vol. 1632. Springer, pp. 202–206.

Pientka, B. (2008) A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In*Proceedings of the 35th ACM SIGPLAN-SIGACT

Symposium on Principles of Programming Languages (POPL), Necula, G. C. & Wadler, P. (eds). ACM, pp. 371–382.

Pierce, B. C. & Turner, D. N. (2000) Local type inference. *ACM Trans. Program. Lang. Syst.* **22**(1), 1–44.

Polakow, J. (2015) Embedding a full linear lambda calculus in Haskell. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, pp. 177–188. ACM.

Poswolsky, A & Schürmann, C. (2009) System description: Delphin – a functional programming language for deductive systems. *Electr. Notes Theor. Comput. Sci.* **228**, 113–120.

Schürmann, C., Despeyroux, J. & Pfenning, F. (2001) Primitive recursion for higher-order abstract syntax. *Theor. Comput. Sci.* **266**(1–2), 1–57.

Stump, A. (2016) *Verified Functional Programming in Agda*. ACM Books.

Stump, A. & Fu, P. (2016) Efficiency of lambda-encodings in total type theory. *J. Funct. Program.* **26**(003).

Tannen, V. & Coquand, T. (1988) Extensional models for polymorphism. *Theor. Comput. Sci.* **59**, 85–114.

The Coq development team (2015) *The Coq Proof Assistant Reference Manual*. LogiCal Project. Version 8.4.

Univalent Foundations Program (2013) *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: `http://homotopytypetheory.org/book`. Last accessed April 21, 2017.

Washburn, G. & Weirich, S. (2003) Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, pp. 249–262.

Werner, B. (1992) A normalization proof for an impredicative type system with large elimination over integers. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*. Available from `http://www.cse.chalmers.se/research/group/logic/Types/previousevents.html`, last access April 23, 2017. pp. 341–357.

Werner, B. (1994) *Une Théorie des Constructions Inductives*. PhD Thesis. Paris VII: Université Paris-Diderot.

Appendix: Omitted rules

In this section are listed some straightforward rules omitted from the definition of CDLE in the main text.

A.1 Rules defining judgement $X \in^p T$

Rules defining judgement $X \in^p T$ are in Figure A1. We write $\hat{p}$ for the other polarity besides p.

B.2 Additional rules for directed conversion

Figure A2 gives additional rules for directed conversion. Computation rules were given in Figure 5. The additional rules include reflexivity, transitivity, and then congruence rules equating expressions where the corresponding subexpressions are equal. Passing under a binder extends the context, and passing into the body of a ν-type adds the constructor set to the context (just like the kinding rule for ν-types).

$$\frac{}{X \in^{+} X} \qquad \frac{X \neq Y}{X \in^{p} Y} \qquad \frac{X \in^{p} T \quad X \neq Y}{X \in^{p} \forall Y : \kappa . T}$$

$$\frac{X \in^{p} T' \quad X \in^{p} T \quad X \neq x}{X \in^{p} \iota x : T' . T} \qquad \frac{X \in^{\hat{p}} T_1 \quad X \in^{p} T_2}{X \in^{p} \Pi x : T_1 . T_2} \qquad \frac{X \in^{\hat{p}} T_1 \quad X \in^{p} T_2}{X \in^{p} \forall x : T_1 . T_2}$$

$$\frac{X \in^{p} T \quad X \notin FV(T')}{X \in^{p} T\ T'} \qquad \frac{X \in^{p} T}{X \in^{p} T\ t} \qquad \frac{X \in^{p} T'}{X \in^{p} \lambda x : T . T'}$$

$$\frac{X \in^{p} T' \quad X \neq Y}{X \in^{p} \lambda Y : \kappa . T'} \qquad \frac{X \in^{p} T \quad X \in^{p} \Theta \quad X \in^{p} T \quad X \neq Y}{X \in^{p} \nu Y \subseteq T : \kappa | \Theta . T}$$

$$\frac{}{X \in^{p} \mathscr{U}} \qquad \frac{}{X \in^{p} \cdot} \qquad \frac{X \in^{p} T \quad X \in^{p} \Theta}{X \in^{p} (t \in T, \Theta)}$$

$$\frac{}{X \in^{p} \star} \qquad \frac{X \in^{\hat{p}} T \quad X \in^{p} \kappa}{X \in^{p} \Pi x : T . \kappa} \qquad \frac{X \neq Y \quad X \in^{\hat{p}} \kappa \quad X \in^{p} \kappa'}{X \in^{p} \Pi Y : \kappa . \kappa'}$$

Fig. A1. Polarity of occurrences of type variables in types, constructor sets, and kinds.

$$\frac{}{\Gamma \vdash T \rhd T} \qquad \frac{\Gamma \vdash T_1 \rhd T_2 \quad \Gamma \vdash T_2 \rhd T_3}{\Gamma \vdash T_1 \rhd T_3}$$

$$\frac{\Gamma \vdash T \rhd T' \quad \Gamma \vdash t \rhd t'}{\Gamma \vdash T\ t \rhd T'\ t'} \qquad \frac{\Gamma \vdash T_1 \rhd T_1' \quad \Gamma \vdash T_2 \rhd T_2'}{\Gamma \vdash T_1\ T_2 \rhd T_1'\ T_2'} \qquad \frac{\Gamma \vdash T_1 \rhd T_1' \quad \Gamma \vdash, X : T_1 \vdash T_2 \rhd T_2'}{\Gamma \vdash \lambda X : T_1 . T_2 \rhd \lambda X : T_1' . T_2'}$$

$$\frac{\Gamma, X : \kappa \vdash T \rhd T'}{\Gamma \vdash \lambda X : \kappa . T \rhd \lambda X : \kappa . T'} \qquad \frac{\Gamma \vdash T_1 \rhd T_1' \quad \Gamma, X : T_1 \vdash T_2 \rhd T_2'}{\Gamma \vdash \Pi X : T_1 . T_2 \rhd \Pi X : T_1' . T_2} \qquad \frac{\Gamma \vdash T \rhd T' \quad \Gamma, X : T \vdash L \rhd L'}{\Gamma \vdash \Pi X : T . L \rhd \Pi X : T' . L}$$

$$\frac{\Gamma \vdash T_1 \rhd T_1' \quad \Gamma, X : T_1 \vdash T_2 \rhd T_2'}{\Gamma \vdash \forall X : T_1 . T_2 \rhd \forall X : T_1' . T_2} \qquad \frac{\Gamma, X : \kappa \vdash T \rhd T'}{\Gamma \vdash \forall X : \kappa . T \rhd \forall X : \kappa . T} \qquad \frac{\Gamma \vdash T_1 \rhd T_1' \quad \Gamma, X : T_1 \vdash T_2 \rhd T_2'}{\Gamma \vdash \iota X : T_1 . T_2 \rhd \iota X : T_1' . T_2}$$

$$\frac{\Gamma \vdash L \rhd L' \quad \Gamma \vdash t \rhd t'}{\Gamma \vdash \uparrow_{L} t \rhd \uparrow_{L'} t} \qquad \frac{\Gamma \vdash L_1 \rhd L_1' \quad \Gamma \vdash L_2 \rhd L_2'}{\Gamma \vdash L_1 \to L_2 \rhd L_1' \to L_2'} \qquad \frac{\Gamma, X : \kappa \vdash \Theta \rhd \Theta' \quad \Gamma, X : \kappa, \Theta \vdash T \rhd T'}{\Gamma \vdash \nu X : \kappa | \Theta . T \rhd \nu X : \kappa | \Theta' . T'}$$

$$\frac{}{\Gamma \vdash \cdot \rhd \cdot} \qquad \frac{\Gamma \vdash t \rhd t' \quad \Gamma \vdash T \rhd T' \quad \Gamma \vdash \Theta \rhd \Theta'}{\Gamma \vdash t \in T, \Theta \rhd t' \in T', \Theta'}$$

Fig. A2. Additional rules for conversion.