A Type Theory for Defining Logics and Proofs

Brigitte Pientka David Thibodeau Andreas Abel Francisco Ferreira Rebecca Zucchini
School of Computer Science McGill University
Dep. of Computer Science and Eng. Gothenburg University
Dept. of Computing Imperial College London
ENS Paris Saclay

***Abstract*—We describe a Martin-Löf-style dependent type theory, called COCON, that allows us to mix the intensional function space that is used to represent higher-order abstract syntax (HOAS) trees with the extensional function space that describes (recursive) computations. We mediate between HOAS representations and computations using contextual modal types. Our type theory also supports an infinite hierarchy of universes and hence supports type-level computation—thereby providing metaprogramming and (small-scale) reflection. Our main contribution is the development of a Kripke-style model for COCON that allows us to prove normalization. From the normalization proof, we derive subject reduction and consistency. Our work lays the foundation to incorporate the methodology of logical frameworks into systems such as Agda and bridges the longstanding gap between these two worlds.**

I. INTRODUCTION

Higher-order abstract syntax (HOAS) is an elegant and deceptively simple idea of encoding syntax and more generally formal systems given via axioms and inference rules. The basic idea is to map uniformly binding structures in our object language (OL) to the function space in a meta-language thereby inheriting α-renaming and capture-avoiding substitution. In the logical framework LF (Harper et al., 1993), for example, we encode a simple OL consisting of functions, function application, and let-expressions using a type `tm` as:

```
lam : (tm → tm) → tm.
app : tm → tm → tm.
letv: tm → (tm → tm) → tm.
```

The OL term $(\mathsf{lam}\ x.\mathsf{lam}\ y.\mathsf{let}\ w = x\ y\ \mathsf{in}\ w\ y)$ is then encoded as

```
lam λx.lam λy.letv (app x y) λw.app w y
```

using the LF abstractions to model binding. OL substitution is modelled through LF application; for instance, the fact that $((\mathsf{lam}\ x.M)\ N)$ reduces to $[N/x]M$ in our object language is expressed as `(app (lam M) N)` reducing to `(M N)`. This approach can offer substantial benefits: programmers do not need to build up the basic mathematical infrastructure, they can work at a higher-level of abstraction, encodings are more compact, and hence it is easier to mechanize formal systems together with their meta-theory.

However, this approach relies on the fact that we use an *intensional* function space that lacks recursion, case analysis, inductive types, and universes to adequately represent syntax. In LF, for example, we use the dependently-typed lambda calculus as a meta-language to represent formal systems. Under this view, intensional functions represent syntactic binding structures and functions are transparent. However, we cannot write recursive programs about such syntactic structures *within* LF, as we lack the power of recursion. In contrast, (recursive) computation relies on the *extensional* function space. Under this view, functions are opaque and programmers cannot compare two functions for equality nor can they use pattern matching on functions to inspect their bodies. Functions are treated as a black box.

To understand the fundamental difference between defining HOAS trees in LF vs. defining HOAS-style trees using inductive types, let us consider an inductive type `D` with one constructor `lam: (D → D) → D`. What is the problem with such a definition in type theory? – In functional ML-like languages, this is, of course, possible, and types like `D` can be explained using domain theory (Scott, 1976). However, the function argument to the constructor `lam` is opaque and we would not be able to pattern match deeper on the argument to inspect the shape and structure of the syntax tree that is described by it. We can only observe it by applying it to some argument. The resulting encoding also would not be adequate, i.e. there are terms of type `D` that are in normal form but do not uniquely correspond to a term in the object language we try to model. As a consequence, we may need to rule out such "exotic" representations (Despeyroux et al., 1995). But there is a more fundamental problem. In proof assistants based on type theory such as Coq or Agda, we cannot afford to work within an inconsistent system and we demand that all programs we write are terminating. The definition of a constructor `lam` as given previously would be forbidden, as it violates what is known as the positivity restriction.

It is worth pointing out that although we have extensional functions, we still may have an intensional type theory keeping the definitional equality (and hence typechecking) decidable. This notion of intensional equality should not be confused with the intensional function space that we attributed to LF and contrasted to the extensional function space that exists in type theories such as Martin Löf type theory.

This example begs two questions: How can we reason inductively about LF definitions, if they are seemingly not inductive? Do we have to simply give up on HOAS definitions to model syntactic structures within type theory to remain consistent?

Over the past two decades, we have made substantial progress in bringing the intensional and extensional views closer together. Despeyroux et al. (1997) made the key observation that we can mediate between the weak LF and the strong computation-level function space using a box-modality. The authors describe a simply typed lambda-calculus with iteration and case constructs which preserves the adequacy of HOAS encodings. The well-known paradoxes are avoided through the use of a modal box operator which obeys the laws of S4. In addition to being simply typed, all computation had to be on closed HOAS trees. Despeyroux and Leleu (1999) sketch an extension to dependent type theory – however it lacks a normalization proof.

BELUGA (Pientka and Dunfield, 2010; Pientka and Cave, 2015) took another important step towards writing inductive proofs about HOAS trees by generalizing the box-modality to a contextual modal type (Nanevski et al., 2008; Pientka, 2008). For example, we characterize the OL term let $w = x\ y$ in $w\ y$ as a contextual LF object $\lceil x, y \vdash \texttt{letv}\ (\texttt{app}\ x\ y)\ \lambda w.\texttt{app}\ w\ y\rceil$ pairing the LF term together with its LF context. Its contextual type is $\lceil x{:}\texttt{tm}, y{:}\texttt{tm} \vdash \texttt{tm}\rceil$. Here, $\lceil\ \ \rceil$ is a generalization of the box modality described in Despeyroux et al. (1997). This allows us to analyze contextual HOAS trees using case distinction and recursion. Exploiting the Curry-Howard isomorphism, inductive proofs about HOAS trees can be described using recursive functions. However, the gap to full dependent type theories with recursion and universes such as Martin-Löf type theory still remains. In particular, BELUGA cleanly separates representing syntax from reasoning about syntax. The resulting language is an indexed type system in the tradition of Zenger (1997) and Xi and Pfenning (1999) where the index language is contextual LF. This has the advantage that meta-theoretic proofs are modular and only hinge on the fact that equality in the index domain is decidable. However, this approach also is limited in its expressiveness: there is no support for type-level computation or higher-ranked polymorphism, and we lack the power to express properties of computations. This prevents us from fully exploiting the power of metaprogramming and reflection.

In this paper, we present the Martin-Löf style dependent type theory COCON where we mediate between intensional and extensional objects using contextual types. As in BELUGA, we can write recursive programs about contextual LF objects. However, in contrast to BELUGA, we also allow computations to be embedded within LF objects. For example, if a program t promises to compute a value of type $\lceil x{:}\texttt{tm}, y{:}\texttt{tm} \vdash \texttt{tm}\rceil$, then we can embed t directly into an LF object writing $\texttt{lam}\ \lambda x.\texttt{lam}\ \lambda y.\texttt{app}\ \lfloor t\rfloor\ x$, unboxing t. If helpful, one might think of boxing ($\lceil\ \rceil$) as quoting syntax and unboxing ($\lfloor\ \rfloor$) as unquoting computation and embedding its value within the syntax tree.

Allowing computations within LF objects, might seem like a small change syntactically, but it has far reaching consequences. To establish consistency of the type theory, we cannot consider normalization of LF separately from normalization of computations anymore, as it is done in Pientka and Abel (2015) and Jacob-Rao et al. (2018). Moreover, COCON is a predicative type theory and supports an infinite hierarchy of universes. This allows us to write type-level computation, i.e. we can compute types whose shape depends on a given value. Such recursively defined types are sometimes called large eliminations (Werner, 1992). Due to the presence of type-level computations dependencies cannot be erased from the model. As a consequence, the simpler proof technique of Harper and Pfenning (2005) which considers approximate shape of types and has been used to prove completeness of the equivalence algorithm for LF's type theory cannot be used in our setting. Instead, we follow recent work by Abel and Scherer (2012) and Abel et al. (2018) in defining a Kripke-style semantic model for computations that is defined recursively on its semantic type. Our model highlights the intensional character of the LF function space and the extensional character of computations. Our main contribution is the design of the Kripke-style model for the dependent type theory COCON that allows us to establish normalization. From the normalization proof, we derive type uniqueness, subject reduction, and consistency.

We believe COCON lays the foundation to incorporate the methodology of logical frameworks into systems such as Agda (Norell, 2007) or Coq (Bertot and Castéran, 2004). This finally allows us to combine the world of type theory and logical frameworks inheriting the best of both worlds.

II. MOTIVATION

To motivate from a practical point of view why we would like to combine the power of LF with a full dependent type theory, we sketch here the translation of the simply typed lambda calculus (STLC) into cartesian closed categories (CCC) using our framework. This has two goals: first, it illustrates how to use contextual type theory and second, it gives a glimpse of how contextual type theory is itself related to category theory.

To begin, we encode simple types in LF using the type family `obj`.

```
obj : type.
one : obj.
cross : obj → obj → obj.
arrow : obj → obj → obj.
```

We then encode STLC using the indexed type family `tm` to only capture well-typed terms. As before we use the intrinsic LF function space to encode STLC using HOAS.

```
tm : obj → type.
tUnit : tm one.
tPair : tm A → tm B → tm (cross A B).
tFst  : tm (cross A B) → tm A.
tSnd  : tm (cross A B) → tm B.
tLam  : (tm A → tm B) → tm (arrow A B).
tApp  : tm (arrow A B) → tm A → tm B.
```

As is common practice in implementations of LF, we treat free variables `A` and `B` as implicitly Π-quantified at the outside; they can typically be reconstructed. Our goal is to translate between well-typed terms in STLC and morphisms and also state some of the equivalence theorems. Morphisms are relations between objects. The standard morphisms in CCC can be encoded directly where we define the composition of morphisms using `@` as an infix operation for better readability.

```
mor   : obj → obj → type.
id    : mor A A.
```

```
@    : mor B C → mor A B → mor A C.
drop : mor A one.
fst  : mor (cross A B) A.
snd  : mor (cross A B) B.
pair : mor A B → mor A C → mor A (cross B C).
app  : mor (cross (arrow B C) B) C.
cur  : mor (cross A B) C → mor A (arrow B C).
```

To translate well-typed terms in STLC, we need to traverse terms under binders. Following BELUGA, we introduce a context schema, `ctx`, that classifies contexts containing declarations of type `tm A` for some object `A` (see also (Pientka and Dunfield, 2008)): **schema** `ctx = tm A`.

Before we can define a function `itm` that interprets STLC into CCC, we must describe how to interpret a context as an object in CCC. This is what the function `ictx` does. It has type `(γ : ctx) ⇒ ⌈ ⊢ obj⌉`. Here we write ⇒ for the extensional function space in contrast to → which we use for the intensional LF function space. For better readability, we write our function using pattern matching, although the core type theory we present subsequently uses recursors.

```
rec ictx :   (γ : ctx) ⇒ ⌈ ⊢ obj⌉ =
fn  ·                   = ⌈ ⊢ one⌉
  | γ, x:tm (⌊A⌋ with ·) = ⌈ ⊢ cross ⌊ictx γ⌋ ⌊A⌋⌉;
```

The function `ictx` takes as input a context γ which we analyze via pattern matching. Intuitively, γ is built like lists and we can pattern match on γ considering the empty context, written as ⌈·⌉, and the context that contains at least one declaration `x:tm (⌊A⌋ with ·)`. Both γ and A are pattern variables; they are bound on the computation level. This is in contrast to LF variables that occur inside a box and are bound by LF lambda-abstraction or by the LF context associated with an LF object. As `A` denotes a closed object and does not depend on γ, we unbox it together with the weakening substitution (written as ·) which moves `A` from the empty LF context to the LF context γ. In general, we omit the substitution associated with unboxing, if it is the identity.

The function `ictx` returns a closed object which we indicate by `⌈ ⊢ obj⌉`. Note that we do not simply return an LF object of type `obj`, as we mediate between LF objects that describe syntactic object and computations using box and unbox.

The ideas so far follow closely BELUGA, a programming environment that supports writing recursive programs about LF specifications. However, in contrast to BELUGA, we inline the recursive call `ictx γ` using unbox, written as ⌊ ⌋.

The real power of having a Martin Löf style type theory where we can embed computations within contextual types, becomes apparent, when we define the interpretation of STLC into CCC. The type of the interpretation function `itm` concisely specifies that it translates a well-typed lambda term `m` that has type `A` in the context γ to a morphism from `ictx γ` to `A`. Here we rely on the function `ictx` that translates a context γ to an object. Adopting Agda's approach, i.e. we use curly braces to indicate implicit arguments and round braces for explicit arguments. In a proof environment such as BELUGA that supports contextual types and first-class contexts, we would not be able to refer to the function `ictx` inside the type declaration of `itm`, as BELUGA makes a clear distinction between contextual LF types (and LF objects) and functions about them.

```
rec itm : {γ : ctx} ⇒ {A : ⌈ ⊢ obj⌉}
⇒ (m : ⌈γ ⊢ tm (⌊A⌋ with ·)⌉)
⇒ ⌈ ⊢ mor ⌊ictx γ⌋ ⌊A⌋⌉ =
fn (p : ⌈γ ⊢# tm (⌊A⌋ with ·)⌉ = ivar γ p
 | ⌈γ ⊢ tUnit⌉    = ⌈ ⊢ drop⌉
 | ⌈γ ⊢ tFst ⌊e⌋⌉ = ⌈ ⊢ fst @ ⌊itm e⌋ ⌉
 | ⌈γ ⊢ tSnd ⌊e⌋⌉ = ⌈ ⊢ snd @ ⌊itm e⌋ ⌉
 | ⌈γ ⊢ tPair ⌊e1⌋ ⌊e2⌋⌉ =
     ⌈ ⊢ pair ⌊itm e1⌋ ⌊itm e2⌋ ⌉
 | ⌈γ ⊢ tLam λx.⌊e⌋⌉ = ⌈ ⊢ cur ⌊itm e⌋ ⌉
 | ⌈γ ⊢ tApp ⌊e1⌋ ⌊e2⌋⌉ =
     ⌈ ⊢ app @ pair ⌊itm e1⌋ ⌊itm e2⌋ ⌉;
```

We implement the interpretation of STLC as morphisms by pattern matching on `m` considering all the constructors to build lambda-terms and the variable case, i.e. we have a variable from γ. In the latter case, we use the pattern variable `p` with contextual type `⌈γ ⊢# tm (⌊A⌋ with ·)⌉` which can only be instantiated with a variable from γ. We omit here the implementation of `ivar` due to space. It simply looks up a variable in the LF context γ and builds the corresponding projection. The most interesting case is `⌈γ ⊢ tLam x.⌊e⌋⌉`, where `e` has type `⌈γ, x:tm ⌊B⌋ ⊢ tm ⌊C⌋⌉`. The recursive call `itm e` returns a morphism from `⌊ictx (γ, x:tm ⌊B⌋)⌋` to `⌊C⌋` which matches what is expected, since `⌊ictx (γ, x:tm ⌊B⌋)⌋` evaluates to `(cross ⌊ictx γ⌋ ⌊B⌋)`.

Next we translate a morphism to a STL term. Given a morphism between `A` and `B`, we build a term of type `B` with one variable of type `A`. As our types are closed, we again employ the weakening substitution whenever we refer to `B` in a non-empty context.

```
rec imorph : {A : ⌈ ⊢ obj⌉} ⇒ {B : ⌈ ⊢ obj⌉}
⇒ (m : ⌈ ⊢ mor ⌊A⌋ ⌊B⌋⌉)
⇒ ⌈x:tm ⌊A⌋ ⊢ tm (⌊B⌋ with ·)⌉ =
fn ⌈ ⊢ id⌉ = ⌈x:tm _ ⊢ x⌉
 | ⌈ ⊢ ⌊f⌋ @ ⌊g⌋⌉ =
     ⌈x:tm _ ⊢ ⌊imorph f⌋ with ⌊imorph g⌋ ⌉
 | ⌈ ⊢ drop⌉ = ⌈x:tm _ ⊢ tUnit⌉
 | ⌈ ⊢ fst⌉  = ⌈x:tm _ ⊢ tFst x⌉
 | ⌈ ⊢ snd⌉  = ⌈x:tm _ ⊢ tSnd x⌉
 | ⌈ ⊢ pair ⌊f⌋ ⌊g⌋⌉ =
     ⌈x:tm _ ⊢ tPair ⌊imorph f⌋ ⌊imorph g⌋⌉
 | ⌈ ⊢ cur ⌊f⌋⌉ =
     ⌈x:tm _ ⊢ tLam λy.(⌊imorph f⌋ with tPair x y)⌉
 | ⌈ ⊢ app⌉ = ⌈x:tm _ ⊢ tApp (tFst x) (tSnd x)⌉;
```

The translation is mostly straight forward. The most interesting cases are the case for currying and concatenation. In the former, given `cur ⌊f⌋` of type `⌈ ⊢ mor ⌊A⌋ (arrow ⌊B⌋ ⌊C⌋)⌉`, we recursively translate `f:⌈ ⊢ mor (cross ⌊A⌋ ⌊B⌋) C⌉`. It yields a STL term of type `⌈x:tm (cross ⌊A⌋ ⌊B⌋)⊢ tm _⌉`. We now need to replace the LF variable `x` that occurs in the result of the recursive call `imorph f` with `tPair x y` to build a STL term `⌈x:tm ⌊A⌋ ⊢ tm (arrow ⌊B⌋ ⌊C⌋)⌉`. We therefore unbox the result of the recursive call with the substitution `tPair x y`. This is written as `⌊imorph f⌋ with tPair x y`. Here we do not write the domain of the substitution explicitly, however the type of `imorph f` tells us that the result of translating `f` contains one LF variable. In general, we write LF substitutions as lists whose domain is determined by the contextual object we unbox.

To translate a morphism ⌈ ⊢ ⌊f⌋ @ ⌊g⌋⌉, we recursively translate f and g where imorph f returns a STL term of type ⌈x:tm ⌊B⌋ ⊢ tm ⌊C⌋⌉ and imorph g returns a STL term of type ⌈x:tm ⌊A⌋ ⊢ tm ⌊B⌋⌉. To produce the desired STL term of type ⌈x:tm ⌊A⌋ ⊢ tm ⌊C⌋⌉, we replace the LF variable x in the translation of f with the result of the translation of g. This is simply done by ⌈x:tm ⌊A⌋ ⊢ ⌊imorph f⌋ **with** ⌊imorph g⌋⌉. We note that ⌊imorph g⌋ is associated with the identity substitution and hence the LF variable that occurs in the result of ⌊imorph g⌋ is implicitly renamed and bound by x.

Finally, we sketch the equivalence between STLC and CCC to illustrate what new possibilities COCON opens up. We do not show the concrete implementation, since this would go beyond this paper.

Assuming that we have defined convertibility of lambda-terms (conv) and equality (~) between morphism, we can now state the equivalence between STLC and CCC succinctly.

```
rec stlc2ccc : {γ : ctx} ⇒ {A : ⌈ ⊢ obj⌉} ⇒
{M : ⌈γ ⊢ tm ⌊A⌋ with ·⌉} ⇒
{N : ⌈γ ⊢ tm ⌊A⌋ with ·⌉} ⇒
(e : ⌈γ ⊢ conv ⌊M⌋ ⌊N⌋⌉)
⇒ ⌈ ⊢ ⌊itm M⌋ ~ ⌊itm N⌋ ⌉

rec ccc2tm : {A : ⌈ ⊢ obj⌉} ⇒ {B : ⌈ ⊢ obj⌉} ⇒
{f : ⌈ ⊢ mor ⌊A⌋ ⌊B⌋⌉} ⇒ {g : ⌈ ⊢ mor ⌊A⌋ ⌊B⌋⌉} ⇒
(m : ⌈ ⊢ ⌊f⌋ ~ ⌊g⌋⌉) ⇒
⌈x:tm A ⊢ conv ⌊imorph f⌋ ⌊imorph g⌋⌉
```

We hope this example provides a glimpse of what COCON has to offer. In the rest of the paper we develop the dependent type theory for COCON that supports both defining HOAS trees using the intensional function space of LF and defining (type-level) computations using the extensional function space.

III. A TYPE THEORY FOR DEFINING LOGICS AND PROOFS

COCON combines the logical framework LF with a full dependent type theory that supports recursion over HOAS objects and universes. For clarity, we split COCON's grammar into different syntactic categories (see Fig. 1).

LF Kinds	K	::=	$\mathsf{type} \mid \Pi x{:}A.K$
LF Types	A, B	::=	$\mathsf{a}\ M_1 \ldots M_n \mid \Pi x{:}A.B$
LF Terms	M, N	::=	$\lambda x.M \mid M\ N \mid x \mid \mathsf{c} \mid \lfloor t \rfloor_\sigma$
LF Contexts	Ψ, Φ	::=	$\cdot \mid \psi \mid \Psi, x{:}A$
LF Context (Erased)	$\hat{\Psi}, \hat{\Phi}$	::=	$\cdot \mid \psi \mid \hat{\Psi},\ x$
LF Substitutions	σ	::=	$\cdot \mid \mathsf{wk}_{\hat{\Psi}} \mid \sigma, M$
LF Signature	Σ	::=	$\cdot \mid \Sigma, \mathsf{a} : K \mid \Sigma, \mathsf{c} : A$
Contextual Types	T	::=	$\Psi \vdash A \mid \Psi \vdash_\# A$
Contextual Objects	C	::=	$\hat{\Psi} \vdash M$
Sorts	u	::=	U_k
Domain of Discourse	$\breve{\tau}$	::=	$\tau \mid \mathsf{ctx}$
Types and	$\tau, \mathcal{I},$	::=	$u \mid \lceil T \rceil \mid (y : \breve{\tau}_1) \Rightarrow \tau_2$
Terms	t, s		$\mid y \mid \lceil C \rceil \mid \mathsf{rec}^{\mathcal{I}}\ \vec{\mathcal{B}}\ \Psi\ \vec{t}$
			$\mid \mathsf{fn}\ y \Rightarrow t \mid t_1\ t_2$
Branches	$\mathcal{B}$	::=	$\Gamma \Rightarrow t$
Contexts	Γ	::=	$\cdot \mid \Gamma, y : \breve{\tau}$

Fig. 1. Syntax of COCON

A. Syntax

a) Logical Framework LF with Embedded Computations: As in LF we allow dependent kinds and types; LF terms can be defined by LF variables, constants, LF applications, and LF lambda-abstractions. In addition, we allow a computation t to be embedded into LF terms using a closure $\lfloor t \rfloor_\sigma$. Once computation of t produces a contextual object M in an LF context Ψ, we can embed the result by applying the substitution σ to M moving M from the LF context Ψ to the current context Φ. In the source level synax that we previously used in the code examples, this was written as ⌊t⌋ **with** σ.

We distinguish between computations that characterize a general LF term M of type A in a context Ψ using the contextual type $\Psi \vdash A$ and computations that are guaranteed to return a variable in a context Ψ of type A using the contextual type $\Psi \vdash_\# A$. This distinction is exploited in the definition of a recursor for contextual objects of type $\lceil \Psi \vdash A \rceil$ to characterize the base case where we consider a LF variable of LF type A. For simplicity and lack of space, we focus on $\Psi \vdash A$ in the subsequent development. Intuitively, $\Psi \vdash_\# A$ is a special case restricted to variables from Ψ inhabiting A.

b) LF contexts: LF contexts are either empty or are built by extending a context with a declaration $x{:}A$. We may also use a (context) variable ψ that stands for a context prefix and must be declared on the computation-level. In particular, we can write functions where we abstract over (context) variables. Consequently, we can pass LF contexts as arguments to functions. We classify LF contexts via schemas – for this paper, we pre-define the schema ctx. Such context schemas are similar to adding base types to computation-level types. We often do not need to carry the full LF context with the type annotations, but it suffices to simply consider the erased LF context. Erased LF contexts are simply a list of variables possibly with a context variable at the head. We sometimes abuse notation and write $\hat{\Psi}$ for the operation of erasing type information from a LF context Ψ.

c) LF Substitutions: LF substitutions allow us to move between LF contexts. The empty LF substitution provides a mapping from an empty LF context to a LF context Ψ and hence has weakening built-in. The weakening substitution, written as $\mathsf{wk}_{\hat{\Psi}}$, describes the weakening of the domain Ψ to $\Psi, \overrightarrow{x{:}A}$. The generality of weakening substitutions is necessary to, for example, express that we can weaken a LF context ψ. We simply write id, if $|\overrightarrow{x{:}A}| = 0$.

Weakening substitutions do not subsume the empty substitutions – only the empty substitution that maps the empty context to a concrete context $\cdot, x_n{:}A_n, \ldots, x_1{:}A_1$ can be expressed as $\mathsf{wk}_\cdot$ where we annotate the weakening substitution with the empty LF context. Our built-in weakening substitutions are also sometimes called *renamings*, as they only allow contexts to be extended to the right, but they do not support arbitrary weakening of a LF context where we would insert a declaration in the middle (i.e. given a context $x{:}A_1, y{:}A_3$ we can weaken it to $x{:}A_1, w{:}A_2, y{:}A_3$).

From a de Bruijn perspective, the weakening substitution

wk. which maps the empty context to $\cdot, x_n{:}A_n, \ldots, x_1{:}A_1$ can be viewed as a shift n. Further, as in the de Bruijn world, $\mathsf{wk}_{(\cdot, x_n:A_n, \ldots, x_1:A_1)}$ can be expanded and is equivalent to $\cdot, x_n, \ldots, x_1$. While our theory lends itself to an implementation with de Bruijn indices, we formulate our type theory using a named representation of variables. This not only simplifies our subsequent definitions of substitutions, but also leaves open how variables are realized in an implementation.

In addition to empty and weakening substitutions, LF substitutions can also be built by extending a LF substitution σ with a LF term M. Following Nanevski et al. (2008), we do not store the domain of a substitution, but simply write them as a list of terms. We resurrect the domain of the substitution before applying it by erasing types from a context.

d) Contextual Objects and Types: We mediate between LF and computations using contextual types. Here, we concentrate on contextual LF terms that have type $\Psi \vdash A$. However, others may be added (see (Cave and Pientka, 2013)).

e) Computations and their Types: Computations are formed by functions, written as $\mathsf{fn}\ y \Rightarrow t$, that are extensional, applications, written as $t_1\ t_2$, boxed contextual objects, written as $\lceil C \rceil$, and recursor, written as $\mathsf{rec}^{\mathcal{I}}\ \vec{\mathcal{B}}\ \Psi\ \vec{t}$ where $\vec{t} = t_n \ldots t_0$. We annotate the recursor with the typing invariant $\mathcal{I}$. We may either recurse over Ψ directly or we recurse over the values computed by the term t_0. The LF context Ψ describes the local LF-world in which the value computed by t_0 makes sense. The arguments $t_n \ldots t_1$ describe in general the implicit arguments t might depend on. Finally, $\mathcal{B}$ describes the different branches that we can take depending on the value computed by t_0. A covering set of branches can be generated generically following Pientka and Abel (2015). In this paper, we will subsequently work with a recursor for the LF type $\mathtt{tm}$ which we encountered in the introduction together with two LF constants $\mathtt{lam}{:}\Pi y{:}(\Pi x{:}\mathsf{tm}.\mathsf{tm}).\mathsf{tm}$ and $\mathtt{app}{:}\Pi x{:}\mathsf{tm}.\Pi y{:}\mathsf{tm}.\mathsf{tm}$ to keep the development compact.

Computation-level types consist of boxed contextual types, written as $\lceil T \rceil$, and dependent types, written as $(y : \check{\tau}_1) \Rightarrow \tau_2$. We overload the dependent function space and allow as domain of discourse both computation-level types and the schema ctx of LF context. To form both functions we use $\mathsf{fn}\ y \Rightarrow t$. We also overload function application $t\ s$ to eliminate dependent types $(y : \tau_1) \Rightarrow \tau_2$ and $(y : \mathsf{ctx}) \Rightarrow \tau_2$, although in the latter case s stands for a LF context. We separate LF contexts from contextual objects, as we do not allow functions that return an LF context.

COCON is a pure type system (PTS) with infinite hierarchy of predicative universes, written as U_k where $k \in \mathsf{Nat}$. The universes are not cumulative. We use sorts $u, k \in \mathcal{S}$, axioms $\mathfrak{A} = \{(\mathsf{U}_i,\ \mathsf{U}_{i+1} \mid i \in \mathsf{Nat}\}$, and rules $\mathfrak{R} = \{(\mathsf{U}_i,\ \mathsf{U}_j,\ \mathsf{U}_{\mathsf{max}(i,j)}) \mid i, j \in \mathsf{Nat}\}$.

B. LF Substitution Operation

Our type theory distinguishes between LF-variables and computation variables and we define substitution for both. We define LF substitutions uniformly using simultaneous substitution operation written as $[\sigma/\hat{\Psi}]M$. As LF substitutions are simply a list of terms, we need to resurrect the domain to lookup the instantiation for a LF variable x in σ. This is always possible.

$$\begin{array}{lll}
[\sigma/\hat{\Psi}](\lambda x.M) & = \lambda x.M' & \text{where } [\sigma, x/\hat{\Psi}, x](M) = M' \\
 & & \text{provided that } x \notin \mathsf{FV}(\sigma) \text{ and } x \notin \hat{\Psi} \\
[\sigma/\hat{\Psi}](M\ N) & = M'\ N' & \text{where } [\sigma/\hat{\Psi}](M) = M' \\
 & & \quad \text{and } [\sigma/\hat{\Psi}](N) = N' \\
[\sigma/\hat{\Psi}](\lfloor t \rfloor_{\sigma'}) & = \lfloor t \rfloor_{\sigma''} & \text{where } [\sigma/\hat{\Psi}](\sigma') = \sigma'' \\
[\sigma/\hat{\Psi}](x) & = M & \text{where lookup } x\ [\sigma/\hat{\Psi}] = M \\
[\sigma/\hat{\Psi}]c & = c & \\
 & & \\
[\sigma/\hat{\Psi}](\cdot) & = \cdot & \\
[\sigma/\hat{\Psi}](\mathsf{wk}_{\hat{\Phi}}) & = \sigma' & \text{where trunc}_{\Phi}\ (\sigma/\hat{\Psi}) = \sigma' \\
[\sigma/\hat{\Psi}](\sigma', M) & = \sigma'', M' & \text{where } [\sigma/\hat{\Psi}](\sigma') = \sigma'' \\
 & & \quad \text{and } [\sigma/\hat{\Psi}](M) = M'
\end{array}$$

When pushing the substitution through an application $M\ N$, we simply apply it to M and N respectively. When pushing the LF substitution through a λ-abstraction, we extend it. When applying σ to a LF variable x, we retrieve the corresponding instantiation from σ using the auxiliary function lookup which works mostly as expected. When applying the LF substitution σ to the LF closure $\lfloor t \rfloor_{\sigma'}$ we leave t untouched, since t cannot contain any free LF variables and compose σ and σ'. Composition of LF substitution is straightforward. When we apply σ to $\mathsf{wk}_{\hat{\Psi}}$, we truncate σ and only keep those entries corresponding to the LF context Ψ. Recall that $\mathsf{wk}_{\hat{\Psi}}$ provides a weakening substitution from a context Ψ to another context $\Psi, \overrightarrow{x{:}A}$ where $|\overrightarrow{x{:}A}| = n$. The simultaneous substitution σ provides mappings for all the variables in $\hat{\Psi}, \vec{x}$. The result of $[\sigma/\hat{\Psi}, \vec{x}]\mathsf{wk}_{\hat{\Psi}}$ then should only provide mappings for all the variables in Ψ. We use the operation trunc to remove irrelevant instantiations. Its definition is straightforward and is given in the long version (see Pientka et al. (2019)).

C. Computation-level Substitution Operation

The computation-level substitution operation $\{t/x\}t'$ traverses the computation t' and replaces any free occurrence of the computation-level variable x in t' with t. The interesting case is $\{t/x\}\lceil C \rceil$. Here we push the substitution into C and we will further apply it to objects in the LF layer. When we encounter a closure such as $\lfloor t'' \rfloor_\sigma$, we continue to push it inside σ and also into t''. When substituting a LF context Ψ for the variable ψ in a context Φ, we rename the declarations present in Φ. This is a convention. It would equally work to rename the variable declarations in Ψ. For example, in $\{(x{:}\mathsf{tm}, y{:}\mathsf{tm})/\psi\}(\hat{\psi}, x \vdash \mathtt{lam}\ \lambda y.\mathsf{app}\ x\ y\ )$, we rename the variable x in $\hat{\psi}, x$ and replace ψ with $(x{:}\mathsf{tm}, y{:}\mathsf{tm})$ in $(\hat{\psi}, w \vdash \mathtt{lam}\ \lambda y.\mathsf{app}\ w\ y)$. This results in $x, y, w \vdash \mathtt{lam}\ \lambda y.\mathsf{app}\ w\ y$. When type checking this term we will eventually also α-rename the λ-bound LF variable y.

Last, we define simultaneous computation-level substitution using the judgment $\boxed{\Gamma' \vdash \theta : \Gamma}$. For simplicity, we overload the typing judgment simply writing $\Gamma \vdash t : \check{\tau}$, although if $\check{\tau} = \mathsf{ctx}$, t stands for a LF context.

$$\frac{\vdash \Gamma'}{\Gamma' \vdash \cdot : \cdot} \quad \frac{\Gamma' \vdash \theta : \Gamma \quad \Gamma' \vdash t : \{\theta\}\breve{\tau}}{\Gamma' \vdash \theta, t/x : \Gamma, x : \breve{\tau}}$$

We distinguish between a substitution θ that provides instantiations for variables declared in the computation context Γ, and a renaming substitution ρ which maps variables in the computation context Γ to the same variables in the context Γ' where $\Gamma' = \Gamma, \overrightarrow{x{:}\breve{\tau}}$ and $\Gamma' \vdash \rho : \Gamma$. We write $\Gamma' \leq_\rho \Gamma$ for the latter. We note that the weakening and substitution properties for simultaneous substitutions also hold for renamings.

$\boxed{\Gamma; \Psi \vdash M : A}$ LF term M has LF type A in the LF context Ψ and context Γ

$$\frac{\Gamma \vdash \Psi : \mathsf{ctx} \quad x{:}A \in \Psi}{\Gamma; \Psi \vdash x : A} \quad \frac{\Gamma \vdash \Psi : \mathsf{ctx} \quad \mathsf{c}{:}A \in \Sigma}{\Gamma; \Psi \vdash \mathsf{c} : A}$$

$$\frac{\Gamma; \Psi \vdash M : \Pi x{:}A.B \quad \Gamma; \Psi \vdash N : A}{\Gamma; \Psi \vdash M\ N : [N/x]B} \quad \frac{\Gamma; \Psi, x{:}A \vdash M : B}{\Gamma; \Psi \vdash \lambda x.M : \Pi x{:}A.B}$$

$$\frac{\Gamma \vdash t : [\Phi \vdash A] \text{ or } \Gamma \vdash t : [\Phi \vdash_\# A] \quad \Gamma; \Psi \vdash \sigma : \Phi}{\Gamma; \Psi \vdash \lfloor t \rfloor_\sigma : [\sigma/\hat{\Phi}]A}$$

$$\frac{\Gamma; \Psi \vdash M : B \quad \Gamma; \Psi \vdash B \equiv A : \mathsf{type}}{\Gamma; \Psi \vdash M : A}$$

$\boxed{\Gamma; \Phi \vdash \sigma : \Psi}$ LF substitution σ provides a mapping from the LF context Ψ to Φ

$$\frac{\Gamma \vdash \Psi, \overrightarrow{x{:}A} : \mathsf{ctx}}{\Gamma; \Psi, \overrightarrow{x{:}A} \vdash \mathsf{wk}_{\hat{\Psi}} : \Psi} \quad \frac{\Gamma \vdash \Phi : \mathsf{ctx}}{\Gamma; \Phi \vdash \cdot : \cdot}$$

$$\frac{\Gamma; \Phi \vdash \sigma : \Psi \quad \Gamma; \Phi \vdash M : [\sigma/\hat{\Psi}]A}{\Gamma; \Phi \vdash \sigma, M : \Psi, x{:}A}$$

Fig. 2. Typing Rules for LF Terms and LF Substitutions

D. LF Typing

We concentrate here on the typing rules for LF terms, LF substitutions and LF contexts (see Fig. 2) and omit the rules for LF types and kinds. All of the typing rules have access to a LF signature Σ which we omit to keep the presentation compact. In typing rules for LF abstractions $\lambda x.M$ we simply extend the LF context and check the body M. When we encounter a LF variable, we look up its type in the LF context. The conversion rule is important and subtle. We only allow conversion of types – conversion of the LF context is not necessary, as we do not allow computations to return a LF context. Importantly, given a computation t that has type $\lceil \Psi \vdash A \rceil$ or $\lceil \Psi \vdash_\# A \rceil$, we can embed it into the current LF context Φ by forming the closure $\lfloor t \rfloor_\sigma$ where σ provides a mapping for the variables in Ψ. This formulation generalizes previous work which only allowed variables declared in Γ to be embedded in LF terms. This enforced a strict separation between computations and LF terms. The typing rules for LF substitutions are as expected.

The typing rules for LF contexts simply analyze the structure of a LF context. When we reach the head, we either encounter an empty LF context or a context variable y which must be declared in the computation-level context Γ. They can be found in the long version.

E. Definitional LF Equality

For LF terms, equality is $\beta\eta$. In addition, we can reduce $\lfloor \Psi \vdash M \rfloor_\sigma$ by simply applying σ to M. We omit the transitive closure rules as well as congruence rules, as they are straightforward.

For LF substitutions, we take into account that weakening substitutions are not unique. For example, the substitution $\mathsf{wk}_\cdot$ may stand for a mapping from the empty context to another LF context; so does the empty substitution $\cdot$. Similarly, $\mathsf{wk}_{x_1,\ldots x_n}$ is equivalent to $\mathsf{wk}_\cdot, x_1, \ldots, x_n$.

$\boxed{\Gamma; \Psi \vdash M \equiv N : A}$ LF Term M is definitionally equal to LF Term N at LF type A

$$\frac{\Gamma; \Psi \vdash M : \Pi x{:}A.B}{\Gamma; \Psi \vdash M \equiv \lambda x.M\ x : \Pi x{:}A.B}$$

$$\frac{\Gamma; \Psi, x{:}A \vdash M_1 : B \quad \Gamma; \Psi \vdash M_2 : A}{\Gamma; \Psi \vdash (\lambda x.M_1)\ M_2 \equiv [M_2/x]M_1 : [M_2/x]B}$$

$$\frac{\Gamma; \Phi \vdash N : A \quad \Gamma; \Psi \vdash \sigma : \Phi}{\Gamma; \Psi \vdash \lfloor \lceil \hat{\Phi} \vdash N \rceil \rfloor_\sigma \equiv [\sigma/\hat{\Phi}]N : [\sigma/\hat{\Phi}]A}$$

$\boxed{\Gamma; \Psi \vdash \sigma \equiv \sigma' : \Phi}$ LF Substitution σ is definitionally equal to LF Substitution σ'

$$\frac{\Gamma \vdash \Psi : \mathsf{ctx}}{\Gamma; \Psi \vdash \mathsf{wk}_\cdot \equiv \cdot : \cdot} \quad \frac{\Gamma \vdash \Phi, x{:}A, \overrightarrow{y{:}B} : \mathsf{ctx}}{\Gamma; \Phi, x{:}A, \overrightarrow{y{:}B} \vdash \mathsf{wk}_{\hat{\Phi},x} \equiv \mathsf{wk}_{\hat{\Phi}}, x : \Phi, x{:}A}$$

Fig. 3. Reduction and Expansion for LF Terms and LF Substitutions

F. Contextual LF Typing and Definitional Equivalence

We lift typing and definitional equality on LF terms to contextual objects. For example, two contextual objects $\hat{\Psi} \vdash M$ and $\hat{\Psi} \vdash N$ are equivalent at LF type $\lceil \Psi \vdash A \rceil$, if M and N are equivalent in Ψ.

G. Computation Typing

We describe well-typed computations in Fig. 4 using the typing judgment $\Gamma \vdash t : \tau$. Computations only have access to computation-level variables declared in the context Γ. We use the judgment $\vdash \Gamma$ to describe well-formed contexts where every declaration $x{:}\breve{\tau}$ in Γ is well-formed.

To avoid duplication of typing rules, we overload the typing judgment and write $\breve{\tau}$ instead of τ, if the same judgment is used to check that a given LF context is of schema ctx. For example, to ensure that $(y : \breve{\tau}_1) \Rightarrow \tau_2$ has kind u_3, we check that $\breve{\tau}_1$ is well-kinded. For compactness, we abuse notation writing $\Gamma \vdash \mathsf{ctx} : u$ although the schema ctx is not a proper type whose elements can be computed. In the typing rules for computation-level (extensional) functions, the input to the function which we also call domain of discourse may either be of type τ_1 or ctx. To eliminate a term t of type $(y : \tau_1) \Rightarrow \tau_2$,

$\boxed{\Gamma \vdash t : \tau}$ and $\boxed{\Gamma \vdash \tau : u}$ Typing and Kinding Judg. for Comp.

$$\frac{y : \breve{\tau} \in \Gamma \quad \vdash \Gamma}{\Gamma \vdash y : \breve{\tau}} \qquad \frac{\vdash \Gamma}{\Gamma \vdash u_1 : u_2}\ (u_1, u_2) \in \mathfrak{A}$$

$$\frac{\Gamma \vdash \breve{\tau}_1 : u_1 \quad \Gamma, y{:}\tau_1 \vdash \tau_2 : u_2}{\Gamma \vdash (y : \breve{\tau}_1) \Rightarrow \tau_2 : u_3}\ (u_1,\ u_2,\ u_3) \in \mathfrak{R}$$

$$\frac{\Gamma \vdash T}{\Gamma \vdash \lceil T \rceil : u} \qquad \frac{\Gamma \vdash t : (y : \breve{\tau}_1) \Rightarrow \tau_2 \quad \Gamma \vdash s : \breve{\tau}_1}{\Gamma \vdash t\ s : \{s/y\}\tau_2}$$

$$\frac{\Gamma, y : \breve{\tau}_1 \vdash t : \tau_2 \quad \Gamma \vdash (y : \breve{\tau}_1) \Rightarrow \tau_2 : u}{\Gamma \vdash \mathsf{fn}\ y \Rightarrow t : (y : \breve{\tau}_1) \Rightarrow \tau_2}$$

$$\frac{\Gamma \vdash C : T}{\Gamma \vdash \lceil C \rceil : \lceil T \rceil} \qquad \frac{\Gamma \vdash t : \tau' \quad \Gamma \vdash \tau' \equiv \tau : u}{\Gamma \vdash t : \tau}$$

Fig. 4. Typing Rules for Computations (without recursor)

we check that s is of type τ_1 and then return $\{s/y\}\tau_2$ as the type of $t\ s$. To eliminate a term of type $(y : \mathsf{ctx}) \Rightarrow \tau$, we overload application simply writing $t\ s$, although s stands for a LF context and check that s is of schema ctx. This distinction between the domains of discourse is important, as we only allow LF contexts to be built either by a context variable or a LF type declaration. We can embed contextual object C into computations by boxing it and transitioning to the typing rules for LF. We eliminate contextual types using a recursor. As mentioned earlier, we define an iterator over t of type $\lceil \Psi \vdash \mathsf{tm} \rceil$ to keep the exposition compact. For a more general generation of recursors for contextual objects of type $\Psi \vdash A$ and LF contexts, we refer the reader to Pientka and Abel (2015).

In general, the output type of the recursor may depend on the argument we are recursing over. We hence annotate the recursor itself with an invariant $\mathcal{I}$. We consider here the recursor for contextual LF terms where $\mathcal{I} = (\psi : \mathsf{ctx}) \Rightarrow (y : \lceil \psi \vdash \mathsf{tm} \rceil) \Rightarrow \tau$. To check that the recursor $\mathsf{rec}^{\mathcal{I}}\ \mathcal{B}\ \Psi\ t$ has type $\{\Psi/\psi, t/y\}\tau$, we check that each of the three branches has the specified type $\mathcal{I}$. In the base case, we may assume in addition to $\psi : \mathsf{ctx}$ that we have a variable $p : \lceil \psi \vdash_{\#} \mathsf{tm} \rceil$ and check that the body has the appropriate type. If we encounter a contextual LF object built with the LF constant app , then we choose the branch b_{app} . We assume $\psi{:}\mathsf{ctx}$, $m{:}\lceil \psi \vdash \mathsf{tm} \rceil$, $n{:}\lceil \psi \vdash \mathsf{tm} \rceil$, as well as f_n and f_m which stand for the recursive calls on m and n respectively. We then check that the body t_{app} is well-typed. If we encounter a LF object built with the LF constant $\mathtt{lam}$, then we choose the branch $b_{\mathtt{lam}}$. We assume $\psi{:}\mathsf{ctx}$ and $m{:}\lceil \psi, x{:}\mathsf{tm} \vdash \mathsf{tm} \rceil$ together with the recursive call f_m on m in the extended LF context $\psi, x{:}\mathsf{tm}$. We then check that the body $t_{\mathtt{lam}}$ is well-typed.

H. Definitional Equality for Computations

We now consider definitional equality for computations concentrating on the reduction rules. We omit the transitive closure and congruence rules, as they are as expected.

We consider two computations to be equal, if they evaluate to the same result. We propagate values through computations

Recursor over LF Terms $\mathcal{I} = (\psi : \mathsf{ctx}) \Rightarrow (y : \lceil \psi \vdash \mathsf{tm} \rceil) \Rightarrow \tau$

$$\frac{\begin{array}{ll} \Gamma \vdash t : \lceil \Psi \vdash \mathsf{tm} \rceil & \Gamma \vdash \mathcal{I} : u \\ \Gamma \vdash b_v : \mathcal{I} & \Gamma \vdash b_{\mathsf{app}} : \mathcal{I} \quad \Gamma \vdash b_{\mathsf{lam}} : \mathcal{I} \end{array}}{\Gamma \vdash \mathsf{rec}^{\mathcal{I}}(b_v \mid b_{\mathsf{app}} \mid b_{\mathtt{lam}}\ )\ \ \Psi\ t : \{\Psi/\psi,\ t/y\}\tau}$$

Branches where $\quad \mathcal{I} = (\psi : \mathsf{ctx}) \Rightarrow (y : \lceil \psi \vdash \mathsf{tm} \rceil) \Rightarrow \tau$

$$\frac{\Gamma, \psi : \mathsf{ctx}, p : \lceil\ \psi \vdash_{\#} \mathsf{tm} \rceil \vdash t_v : \{p/y\}\tau}{\Gamma \vdash (\psi, p \Rightarrow t_v) : \mathcal{I}}$$

$$\frac{\begin{array}{l} \Gamma, \psi : \mathsf{ctx}, \\ m{:}\lceil \psi \vdash \mathsf{tm} \rceil, n{:}\lceil \psi \vdash \mathsf{tm} \rceil \\ f_m{:}\{m/y\}\tau, f_n{:}\{n/y\}\tau\ \ \vdash t_{\mathsf{app}} : \{\lceil \psi \vdash \mathsf{app}\ \lfloor m \rfloor\ \ \lfloor n \rfloor \rceil / y\}\tau \end{array}}{\Gamma \vdash (\psi, m, n, f_n, f_m \Rightarrow t_{\mathsf{app}}) : \mathcal{I}}$$

$$\frac{\begin{array}{l} \Gamma, \phi : \mathsf{ctx}, \\ m{:}\lceil \phi, x{:}\mathsf{tm} \vdash \mathsf{tm} \rceil, \\ f_m{:}\{(\phi, x{:}\mathsf{tm})/\psi, m/y\}\tau \vdash t_{\mathtt{lam}}\ \ : \{\phi/\psi, \lceil \phi \vdash \mathtt{lam}\ \lambda x.\lfloor m \rfloor \rceil / y\}\tau \end{array}}{\Gamma \vdash \psi, m, f_m \Rightarrow t_{\mathtt{lam}}\ \ : \mathcal{I}}$$

Fig. 5. Typing Rules for Recursors

and types relying on the computation-level substitution operation. When we apply a term s to a computation $\mathsf{fn}\ y \Rightarrow t$, we β-reduce and replace y in the body t with s. We unfold the recursor depending on the value passed. If it is $\lceil \hat{\Psi} \vdash \mathtt{lam}\ \ \lambda x.M \rceil$, then we choose the branch $t_{\mathtt{lam}}$. If the value is $\lceil \hat{\Psi} \vdash \mathsf{app}\ \ M\ N \rceil$, we continue with the branch t_{app} . If it is $\lceil \hat{\Psi} \vdash x \rceil$, i.e. the variable case, we continue with t_v. Note that if Ψ is empty, then the case for variables is unreachable, since there is no LF variable of type tm in the empty LF context and hence the contextual type $\lceil \cdot \vdash_{\#} \mathsf{tm} \rceil$ is empty.

We also include the expansion of a computation t at type $\lceil \Psi \vdash A \rceil$; it is equivalent to unboxing t with the identity substitution and subsequently boxing it, i.e. t is equivalent to $\lceil \hat{\Psi} \vdash \lfloor t \rfloor_{\mathsf{wk}_{\hat{\Psi}}} \rceil$.

IV. Elementary Properties

For the LF level, we can establish well-formedness of LF context, LF substitution and weakening properties. In addition, we have LF context conversion and equality conversion for LF types. As usual, we can also prove directly functionality of LF type, injectivity of Pi-types.

Lemma IV.1 (Functionality of LF Typing)**.** Let $\Gamma; \Psi \vdash \sigma_1 : \Phi$ and $\Gamma; \Psi \vdash \sigma_2 : \Phi$, and $\Gamma; \Psi \vdash \sigma_1 \equiv \sigma_2 : \Phi$.

1) If $\Gamma; \Phi \vdash \sigma : \Phi'$ then $\Gamma; \Psi \vdash [\sigma_1/\hat{\Phi}]\sigma \equiv [\sigma_2/\hat{\Phi}]\sigma : \Phi'$.
2) If $\Gamma; \Phi \vdash M : A$ then
   $\Gamma; \Psi \vdash [\sigma_1/\hat{\Phi}]M \equiv [\sigma_2/\hat{\Phi}]M : [\sigma_1/\hat{\Phi}]A$.

Proof. We prove these statements by induction on $\Gamma; \Phi \vdash M : A$ (resp. $\Gamma; \Phi \vdash \sigma : \Phi'$) and followed by another inner induction on $\Gamma; \Psi \vdash \sigma_1 \equiv \sigma_2 : \Phi$ to prove (1). □

Lemma IV.2 (Injectivity of LF Pi-Types)**.** If $\Gamma; \Psi \vdash \Pi x{:}A.B \equiv \Pi x{:}A'.B' : \mathsf{type}$ then $\Gamma; \Psi \vdash A \equiv A' : \mathsf{type}$ and $\Gamma; \Psi, x{:}A \vdash B \equiv B' : \mathsf{type}$.

Proof. By equality inversion . □

$$\frac{\Gamma \vdash \mathsf{fn}\ y \Rightarrow t : (y{:}\check{\tau}_1) \Rightarrow \tau_2 \quad \Gamma \vdash s : \check{\tau}_1}{\Gamma \vdash (\mathsf{fn}\ y \Rightarrow t)\ s \equiv \{s/y\}t : \{s/y\}\tau_2} \qquad \frac{\Gamma \vdash t : \lceil \Psi \vdash A \rceil}{\Gamma \vdash \lceil \hat{\Psi} \vdash \lfloor t \rfloor_{\mathsf{wk}_{\hat{\Psi}}} \rceil \equiv t : \lceil \Psi \vdash A \rceil}$$

let $\mathcal{B} = (\psi, p \Rightarrow t_p \mid \psi, m, n, f_m, f_n \Rightarrow t_{\mathsf{app}} \mid \psi, m, f_m \Rightarrow t_{\mathtt{lam}}\ )$ and $\mathcal{I} = (\psi : \mathsf{ctx}) \Rightarrow (y : \lceil \psi \vdash \mathsf{tm} \rceil) \Rightarrow \tau$

$$\frac{\Gamma \vdash \Psi : \mathsf{ctx} \qquad \Gamma; \Psi, x{:}\mathsf{tm} \vdash M : \mathsf{tm} \qquad \Gamma \vdash \mathcal{I} : u}{\Gamma \vdash \mathsf{rec}^{\mathcal{I}}\ \mathcal{B}\ \ \Psi\ \lceil \hat{\Psi} \vdash \mathtt{lam}\ \lambda x.M \rceil \equiv \{\theta\}t_{\mathtt{lam}}\ : \{\Psi/\psi, \lceil \hat{\Psi} \vdash \mathtt{lam}\ \lambda x.M \rceil/y\}\tau}$$

where $\theta = \Psi/\psi,\ \lceil \hat{\Psi}, x \vdash M \rceil/m,\ \mathsf{rec}^{\mathcal{I}}\ \mathcal{B}\ \ (\Psi, x{:}\mathsf{tm})\ \lceil \hat{\Psi}, x \vdash M \rceil/f$

$$\frac{\Gamma \vdash \Psi : \mathsf{ctx} \qquad \Gamma; \Psi \vdash M : \mathsf{tm} \qquad \Gamma; \Psi \vdash N : \mathsf{tm} \qquad \Gamma \vdash \mathcal{I} : u}{\Gamma \vdash \mathsf{rec}^{\mathcal{I}}\ \mathcal{B}\ \ \Psi \lceil \hat{\Psi} \vdash \mathsf{app}\ M\ N \rceil \equiv \{\theta\}t_{\mathsf{app}}\ : \{\Psi/\psi, \lceil \hat{\Psi} \vdash \mathsf{app}\ M\ N \rceil/y\}\tau}$$

where $\theta = \Psi/\psi,\ \lceil \hat{\Psi} \vdash M \rceil/m,\ \lceil \hat{\Psi} \vdash N \rceil/n,\ \mathsf{rec}^{\mathcal{I}}\ \mathcal{B}\ \ \Psi\ \lceil \hat{\Psi} \vdash M \rceil/f_m,\ \mathsf{rec}^{\mathcal{I}}\ \mathcal{B}\ \ \Psi\ \lceil \hat{\Psi} \vdash N \rceil/f_n$

$$\frac{x{:}\mathsf{tm} \in \Psi \quad \Gamma \vdash \Psi : \mathsf{ctx} \qquad \Gamma \vdash \mathcal{I} : u}{\Gamma \vdash \mathsf{rec}^{\mathcal{I}}\ \mathcal{B}\ \ \Psi\ \lceil \hat{\Psi} \vdash x \rceil \equiv \{\Psi/\psi,\ \lceil \hat{\Psi} \vdash x \rceil/p\}t_p : \{\Psi/\psi, \lceil \Psi \vdash x \rceil/y\}\tau}$$

Fig. 6. Definitional Equality for Computations

For the computation level, we also know that computation context Γ is well-formed.; in addition, weakening and substitution properties hold. However, proving functionality of typing and injectivity of Pi-types on the computation-level must be postponed.

V. Weak head reduction

The operational semantics of COCON uses weak head reduction and mirrors declarative equality. It proceeds lazily. We characterize weak head normal forms (whnf) for both, (contextual) LF and computations . They are mutually defined.

Definition V.1 (Whnf of LF)**.**

- *A LF term M is in whnf,* $\mathsf{whnf}\ M$*, iff $M = \lambda x.N$, or M is neutral, i.e.* $\mathsf{wne}\ M$*, or $M = \lfloor t \rfloor_\sigma$ and t is neutral (i.e.* $\mathsf{wne}\ t$*). .*
- *A LF term M is neutral,* $\mathsf{wne}\ M$*, iff M is of the form $h\ \ M_1 \ldots M_n$ where h is either a LF variable or a constant* c*.*

LF substitutions are already in whnf. LF types are also always considered to be in whnf, as computation may only produce a contextual LF term, but not a contextual LF type. Last, (erased) LF contexts are in whnf, as we do not allow computations to return a LF context.

Computation-level expressions are in whnf, if they do not trigger any further computation-level reductions.

Definition V.2 (Whnf of Computations)**.**

- *A term t is in whnf,* $\mathsf{whnf}\ t$*, if t is a $(\mathsf{fn}\ y \Rightarrow s)$ or $(y : \tau_1) \Rightarrow \tau_2$ or u, t is $\lceil C \rceil$ or $\lceil T \rceil$, or t is neutral*
- *A term t is neutral,* $\mathsf{wne}\ t$*, if t is a variable, $t = s_1\ s_2$ where* $\mathsf{wne}\ s_1$*, $t = (\mathsf{rec}^{\mathcal{I}}\ \mathcal{B}\ \ \Psi\ s)$ where either* $\mathsf{wne}\ s$ *or $s = \lceil \hat{\Psi} \vdash \lfloor t \rfloor_\sigma \rceil$ and* $\mathsf{wne}\ t$*.*

We consider boxed objects, i.e. $\lceil C \rceil$, and boxed types, i.e. $\lceil T \rceil$, in whnf, as the contextual object C will be further reduced when we use them and have to unbox them. The remaining definition of whnf characterizes terms that do not trigger any further reductions. We note that weakening preserves whnfs.

We now define weak head reductions (Fig. 7 and Fig. 8). If an LF term is not already in whnf , we have two cases: either we encounter an LF application $M\ N$ and we may need to beta-reduce or M reduces to $\lfloor t \rfloor_\sigma$. If t is neutral, then we are done; otherwise t reduces to a contextual object $\lceil \hat{\Psi} \vdash M \rceil$, and we continue to reduce $[\sigma/\hat{\Psi}]M$.

$\boxed{M \searrow_{\mathsf{LF}} N}$: LF Term M weak head reduces to N s.t. $\mathsf{whnf}\ N$

$$\frac{M \searrow_{\mathsf{LF}} \lambda x.M' \quad [N/x]M' \searrow_{\mathsf{LF}} R}{M\ N \searrow_{\mathsf{LF}} R} \qquad \frac{M \searrow_{\mathsf{LF}} R \quad \mathsf{wne}\ R}{M\ N \searrow_{\mathsf{LF}} R\ N}$$

$$\frac{\mathsf{whnf}\ M}{M \searrow_{\mathsf{LF}} M} \quad \frac{t \searrow \lceil \hat{\Psi} \vdash M \rceil \quad [\sigma/\hat{\Psi}]M \searrow_{\mathsf{LF}} N}{\lfloor t \rfloor_\sigma \searrow_{\mathsf{LF}} N} \quad \frac{t \searrow n \quad \mathsf{wne}\ n}{\lfloor t \rfloor_\sigma \searrow_{\mathsf{LF}} \lfloor n \rfloor_\sigma}$$

$\boxed{\sigma \searrow_{\mathsf{LF}} \sigma'}$: LF Subst. σ weak head reduces to σ' s.t. $\mathsf{whnf}\ \sigma'$

$$\frac{\mathsf{whnf}\ \sigma}{\sigma \searrow_{\mathsf{LF}} \sigma} \quad \frac{}{\mathsf{wk}_{\cdot} \searrow_{\mathsf{LF}} \cdot} \qquad \frac{}{\mathsf{wk}_{(\hat{\Psi}, x)} \searrow_{\mathsf{LF}} \mathsf{wk}_{\hat{\Psi}}, x}$$

Fig. 7. Weak Head Reductions for LF Terms and LF Substitutions

If a computation-level term t is not already in whnf , we have either an application $t_1\ t_2$ or a recursor. For an application $t_1\ t_2$, we reduce t_1. If it reduces to a function, we continue to beta-reduce otherwise we build a neutral application. For the recursor $\mathsf{rec}^{\mathcal{I}}\ \vec{\mathcal{B}}\ \Psi\ t$ we consider different cases: 1) if t reduces to a neutral term, then we cannot proceed; 2) if t reduces to $\lceil \hat{\Psi}' \vdash M \rceil$, and then proceed to further reduce M. If the result is $\lfloor t' \rfloor_\sigma$, where t is neutral, then we cannot proceed; if the result is N where N is neutral, then we proceed and choose the appropriate branch in $\mathcal{B}$. We note that weak head reduction for LF and computation is deterministic and stable under weakening and LF substitutions.

To facilitate our further development we introduce the following notational abbreviations for well-typed whnfs (see Fig. 9)

$\boxed{t \searrow r}$: Term t weak head reduces to r s.t. $\mathsf{whnf}\ r$

$$\frac{\mathsf{whnf}\ t}{t \searrow t} \quad \frac{t_1 \searrow \mathsf{fn}\ y \Rightarrow t \quad \{t_2/y\}t \searrow v}{t_1\ t_2 \searrow v} \quad \frac{t_1 \searrow w \quad \mathsf{wne}\ w}{t_1\ t_2 \searrow w\ t_2}$$

$$\frac{t \searrow s \quad \mathsf{wne}\ s}{\mathsf{rec}^{\mathcal{I}}\ \mathcal{B}\ \ \Psi\ t \searrow \mathsf{rec}^{\mathcal{I}}\ \mathcal{B}\ \ \Psi\ s}$$

$$\frac{t \searrow \lceil \hat{\Psi} \vdash M \rceil \quad M \searrow_{\mathsf{LF}} \lfloor t' \rfloor_\sigma \quad \mathsf{wne}\ t'}{\mathsf{rec}^{\mathcal{I}}\ \mathcal{B}\ \ \Psi\ t \searrow \mathsf{rec}^{\mathcal{I}}\ \mathcal{B}\ \Psi\ (\hat{\Psi} \vdash \lfloor t' \rfloor_\sigma)}$$

$$\frac{t \searrow \lceil \hat{\Psi} \vdash M \rceil \quad M \searrow_{\mathsf{LF}} N \quad \mathsf{wne}\ N \quad \mathcal{B} \ll (\Psi)\ (\hat{\Psi} \vdash N) \searrow v}{\mathsf{rec}^{\mathcal{I}}\ \mathcal{B}\ \ \Psi\ t \searrow v}$$

let $\mathcal{B} = (\psi, p \Rightarrow t_v \mid \psi, m, n, f_m, f_n \Rightarrow t_{\mathsf{app}} \mid \psi, m, f_m \Rightarrow t_{\mathtt{lam}}\ )$

$$\frac{\begin{array}{l}\{\Psi/\psi,\ \lceil \hat{\Psi} \vdash M \rceil/m,\ \lceil \hat{\Psi} \vdash N \rceil/n,\\ \ \mathsf{rec}^{\mathcal{I}}\ \mathcal{B}\ \ \Psi\ \lceil \hat{\Psi} \vdash M \rceil/f_m,\ \mathsf{rec}^{\mathcal{I}}\ \mathcal{B}\ \ \Psi\ \lceil \hat{\Psi} \vdash N \rceil/f_n\ \}t_{\mathsf{app}} \searrow v\end{array}}{\mathcal{B} \ll (\Psi)\ (\hat{\Psi} \vdash \mathtt{app}\ M\ N) \searrow v}$$

$$\frac{\begin{array}{l}\{\Psi/\psi,\ \ \lceil \hat{\Psi}, x \vdash M \rceil/m,\\ \ \mathsf{rec}^{\mathcal{I}}\ \mathcal{B}\ (\Psi, x{:}\mathsf{tm})\ \lceil \hat{\Psi}, x \vdash M \rceil/f_m\}t_{\mathtt{lam}}\ \ \searrow v\end{array}}{\mathcal{B} \ll (\Psi)\ (\hat{\Psi} \vdash \mathtt{lam}\ \lambda x.M) \searrow v}$$

$$\frac{\{\Psi/\psi,\ \lceil \hat{\Psi} \vdash x \rceil/p\}t_v \searrow v}{\mathcal{B} \ll (\Psi)\ (\hat{\Psi} \vdash x) \searrow v}$$

Fig. 8. Weak Head Reductions for Computations

$$\frac{\begin{array}{l}\Gamma;\Psi \vdash M : A\\ \Gamma;\Psi \vdash N : A \quad \Gamma;\Psi \vdash M \equiv N : A \quad M \searrow_{\mathsf{LF}} N\end{array}}{\Gamma;\Psi \vdash M \searrow_{\mathsf{LF}} N : A}$$

$$\frac{\begin{array}{l}\Gamma;\Psi \vdash \sigma_1 : \Phi\\ \Gamma;\Psi \vdash \sigma_2 : \Phi \quad \Gamma;\Psi \vdash \sigma_1 \equiv \sigma_2 : \Phi \quad \sigma_1 \searrow_{\mathsf{LF}} \sigma_2\end{array}}{\Gamma;\Psi \vdash \sigma_1 \searrow_{\mathsf{LF}} \sigma_2 : \Phi}$$

$$\frac{\Gamma \vdash t : \tau \quad \Gamma \vdash t' : \tau \quad \Gamma \vdash t \equiv t' : \tau \quad t \searrow t'}{\Gamma \vdash t \searrow t' : \tau}$$

Fig. 9. Well-Typed Whnf

VI. Kripke-style Logical Relation

We construct a Kripke-logical relation that is defined on well-typed terms to prove weak head normalization. Our semantic definitions for computations follow closely Abel and Scherer (2012) to accommodate type-level computation.

We start by defining semantic equality for LF terms of type $\mathtt{tm}$ (Fig. 10), as these are the terms the recursor eliminates and it illustrates the fact that we are working with syntax trees. To define semantic equality for LF terms M and N, we consider different cases depending on their whnf: 1) if they reduce to $\mathtt{app}\ M_1\ M_2$ and $\mathtt{app}\ \ N_1\ N_2$ respectively, then M_i must be semantically equal to N_i; 2) if they reduce to $\mathtt{lam}\ M'$ and $\mathtt{lam}\ N'$ respectively, then the bodies of M' and N' must be equal. To compare their bodies, we apply both M' and N' to an LF variable x and consider $M'\ x$ and $N'\ x$ in the extended LF context $\Psi, x{:}\mathsf{tm}$. This has the effect

$$\frac{\begin{array}{l}\Gamma;\Psi \vdash M \searrow_{\mathsf{LF}} \lfloor t_1 \rfloor_{\sigma_1} : \mathsf{tm} \quad \mathsf{typeof}(\Gamma \vdash t_1) = \lceil \Phi_1 \vdash \mathsf{tm} \rceil\\ \Gamma;\Psi \vdash N \searrow_{\mathsf{LF}} \lfloor t_2 \rfloor_{\sigma_2} : \mathsf{tm} \quad \mathsf{typeof}(\Gamma \vdash t_2) = \lceil \Phi_2 \vdash \mathsf{tm} \rceil\\ \Gamma \vdash t_1 \equiv t_2 : \lceil \Phi_1 \vdash \mathsf{tm} \rceil \quad \Gamma;\Psi \Vdash \sigma_1 = \sigma_2 : \Phi_1 \quad \Gamma \vdash \Phi_1 \equiv \Phi_2 : \mathsf{ctx}\end{array}}{\Gamma;\Psi \Vdash M = N : \mathsf{tm}}$$

$$\frac{\begin{array}{l}\Gamma;\Psi \vdash M \searrow_{\mathsf{LF}} \mathtt{lam}\ M' : \mathsf{tm}\\ \Gamma;\Psi \vdash N \searrow_{\mathsf{LF}} \mathtt{lam}\ N' : \mathsf{tm} \quad \Gamma;\Psi, x{:}\mathsf{tm} \Vdash M'\ x = N'\ x : \mathsf{tm}\end{array}}{\Gamma;\Psi \Vdash M = N : \mathsf{tm}}$$

$$\frac{\begin{array}{l}\Gamma;\Psi \vdash M \searrow_{\mathsf{LF}} \mathtt{app}\ M_1\ M_2 : \mathsf{tm} \quad \Gamma;\Psi \Vdash M_1 = N_1 : \mathsf{tm}\\ \Gamma;\Psi \vdash N \searrow_{\mathsf{LF}} \mathtt{app}\ N_1\ N_2 : \mathsf{tm} \quad \Gamma;\Psi \Vdash M_2 = N_2 : \mathsf{tm}\end{array}}{\Gamma;\Psi \Vdash M = N : \mathsf{tm}}$$

$$\frac{\Gamma;\Psi \vdash M \searrow_{\mathsf{LF}} x : \mathsf{tm} \quad \Gamma;\Psi \vdash N \searrow_{\mathsf{LF}} x : \mathsf{tm}}{\Gamma;\Psi \Vdash M = N : \mathsf{tm}}$$

Fig. 10. Semantic Equality for LF Terms: $\boxed{\Gamma;\Psi \Vdash M = N : A}$

of opening up the body and replacing the bound LF variable with a fresh one. This highlights the difference between the intensional LF function space and the extensional nature of the computation-level functions. In the former, we can concentrate on LF variables and continue to analyze the LF function body; in the latter, we consider all possible inputs, not just variables; 3) if the LF terms M and N may reduce to the same LF variable in Ψ, then they are obviously also semantically equal; 4) last, if M and N reduce to $\lfloor t_i \rfloor_{\sigma_i}$ respectively. In this case t_i is neutral and we only need to semantically compare the LF substitutions σ_i and check whether the terms t_i are definitional equal. However, what type should we choose? – As the computation t_i is neutral, we can infer a unique type $\lceil \Phi \vdash \mathsf{tm} \rceil$ which we can use. This is defined as follows:

Type inference for Neutral Computations t : $\mathsf{typeof}(\Gamma \vdash t) = \tau$

$$\frac{\mathsf{typeof}(\Gamma \vdash t) = \tau \quad \tau \searrow (y{:}\tau_1) \Rightarrow \tau_2 \quad \Gamma \vdash s : \tau_1}{\mathsf{typeof}(\Gamma \vdash t\ s) = \{s/y\}\tau_2}$$

$$\frac{x{:}\tau \in \Gamma}{\mathsf{typeof}(\Gamma \vdash x) = \tau} \quad \frac{\mathcal{I} = (\psi : \mathsf{ctx}) \Rightarrow (y : \lceil \psi \vdash \mathsf{tm} \rceil) \Rightarrow \tau}{\mathsf{typeof}(\Gamma \vdash \mathsf{rec}^{\mathcal{I}}\ \mathcal{B}\ \Psi\ t) = \{\Psi/\psi, t/y\}\tau}$$

Semantic equality for LF substitutions is also defined by considering different whnfs (Fig. 11). As we only work with well-typed LF objects, there is only one inhabitant for an empty context. Moreover, given a LF substitution with domain $\Phi, x{:}A$, we can weak head reduce the LF substitutions σ and σ' and continue to recursively compare them. A LF substitutions with domain ψ, a context variable, reduces to wk_ψ.

Defining semantic kinding and semantic equality is intricate, as they depend on each other and we need to ensure our definitions are well-founded. Following Abel et al. (2018), we first define semantic kinding, i.e. $\Gamma \Vdash \breve{\tau} : u$ (Fig. 12) which technically falls into two parts: $\Gamma \Vdash \tau : u$ and $\Gamma \Vdash \mathsf{ctx} : u$ where the latter is simply notation, as ctx is not a computation-level type. Function types $(y : \breve{\tau}_1) \Rightarrow \tau_2$ are semantically well-kinded, if $\breve{\tau}_1$ is semantically well-kinded in Γ', a possible extension of Γ and for any term s that has semantic type

$$\frac{\Gamma;\Psi \vdash \sigma \searrow_{\mathsf{LF}} \cdot : \cdot \quad \Gamma;\Psi \vdash \sigma' \searrow_{\mathsf{LF}} \cdot : \cdot}{\Gamma;\Psi \Vdash \sigma = \sigma' : \cdot}$$

$$\frac{\Gamma;\psi, \overrightarrow{x{:}A} \vdash \sigma \searrow_{\mathsf{LF}} \mathsf{wk}_\psi : \psi \quad \Gamma;\psi, \overrightarrow{x{:}A} \vdash \sigma' \searrow_{\mathsf{LF}} \mathsf{wk}_\psi : \psi}{\Gamma;\psi, \overrightarrow{x{:}A} \Vdash \sigma = \sigma' : \psi}$$

$$\frac{\begin{array}{cc}\Gamma;\Psi \vdash \sigma \searrow_{\mathsf{LF}} \sigma_1, M : \Phi, x{:}A & \Gamma;\Psi \Vdash \sigma_1 = \sigma_2 : \Phi \\ \Gamma;\Psi \vdash \sigma' \searrow_{\mathsf{LF}} \sigma_2, N : \Phi, x{:}A & \Gamma;\Psi \Vdash M = N : [\sigma_1/\hat{\Phi}]A\end{array}}{\Gamma;\Psi \Vdash \sigma = \sigma' : \Phi, x{:}A}$$

Fig. 11. Semantic Equality for LF Substitutions: $\boxed{\Gamma;\Psi \Vdash \sigma = \sigma' : \Phi}$

$\breve{\tau}_1$, $\{s/y\}\tau_2$ is well-kinded. In our definition, we make the weakening (renaming) substitution that allows us to move from Γ to Γ' explicit. The definition of semantic kinding is inductively defined on $\breve{\tau}$.

$$\frac{\Gamma \vdash \tau \searrow \lceil T \rceil : u \quad \Gamma \vdash T \equiv T}{\Gamma \Vdash \tau : u} \qquad \frac{\Gamma \vdash \tau \searrow u' : u \quad u' < u}{\Gamma \Vdash \tau : u}$$

$$\frac{\Gamma \vdash \tau \searrow x\ \vec{t} : u \qquad \mathsf{wne}\ (x\ \vec{t})}{\Gamma \Vdash \tau : u} \qquad \frac{\vdash \Gamma}{\Gamma \Vdash \mathsf{ctx} : u}$$

$$\frac{\begin{array}{c}\Gamma \vdash \tau \searrow (y : \breve{\tau}_1) \Rightarrow \tau_2 : u \quad \forall \Gamma' \leq_\rho \Gamma.\ \Gamma' \Vdash \{\rho\}\breve{\tau}_1 : u_1 \\ \forall \Gamma' \leq_\rho \Gamma.\ \Gamma' \Vdash s = s\ : \{\rho\}\breve{\tau}_1 \Longrightarrow \Gamma' \Vdash \{\rho, s/y\}\tau_2 : u\end{array}}{\Gamma \Vdash \tau : u}$$
$$\text{where } (u_1,\ u_2,\ u) \in \mathfrak{R}$$

Fig. 12. Semantic Kinding for Types $\boxed{\Gamma \Vdash \breve{\tau} : u}$ (inductive)

Semantic kinding (Fig. 12) is used as a measure to define the semantic typing for computations. In particular, we define $\Gamma \Vdash \breve{\tau} = \breve{\tau}' : u$ and $\Gamma \Vdash t = t' : \breve{\tau}$ recursively on the semantic kinding of $\breve{\tau}$. i.e. $\Gamma \Vdash \breve{\tau} : u$. For better readability, we simply write for example $\Gamma \Vdash t = t' : \lceil T \rceil$ instead of $\Gamma \Vdash t = t' : \tau$ where $\tau \searrow \lceil T \rceil$, and $\Gamma \vdash T \equiv T$ in proofs. The extensional character of function types is apparent in the semantic equality for terms at function type. Semantic equality at type $\lceil \Psi \vdash A \rceil$ falls back to semantic equality on LF terms at type A in the LF context Ψ.

VII. Semantic Properties

A. Semantic Properties of LF

If an LF term is semantically well-typed, then it is also syntactically well-typed. Furthermore, our definition of semantic LF typing is stable under renaming and semantic LF equality is preserved under LF substitution and allows for context conversion.

Lemma VII.1 (Backwards Closure for LF terms)**.**
If $\Gamma;\Psi \Vdash Q = N : A$ (or $\Gamma;\Psi \Vdash N = Q : A$) and $\Gamma;\Psi \vdash M \searrow_{\mathsf{LF}} Q : A$ then $\Gamma \Vdash M = N : A$

Proof. By case analysis on $\Gamma;\Psi \Vdash Q = N : A$ and the fact that Q is in whnf . □

Our semantic definitions are reflexive, symmetric, and transitive. Further they are stable under type conversions. We state the lemma below only for terms, but it must in fact be proven mutually with the corresponding property for LF substitutions. We first establish these properties for LF and subsequently for computations. Establishing these properties is tricky and intricate. All proofs can be found in the long version.

Lemma VII.2 (Reflexivity, Symmetry, Transitivity, and Conversion of Semantic Equality for LF)**.**

1) $\Gamma;\Psi \Vdash M = M : A$.
2) If $\Gamma;\Psi \Vdash M = N : A$ then $\Gamma;\Psi \Vdash N = M : A$.
3) If $\Gamma;\Psi \Vdash M_1 = M_2 : A$ and $\Gamma;\Psi \Vdash M_2 = M_3 : A$ then $\Gamma;\Psi \Vdash M_1 = M_3 : A$.
4) and $\Gamma;\Psi \Vdash M = N : A$ then $\Gamma;\Psi \Vdash M = N : A'$.

Proof. Reflexivity follows directly from symmetry and transitivity. For LF terms (and LF substitutions), we prove symmetry and conversion by induction on the derivation $\Gamma;\Psi \Vdash M = N : A$ and $\Gamma;\Psi \Vdash \sigma = \sigma' : \Phi$ respectively. For transitivity, we use lexicographic induction. The proofs relies on symmetry of decl. equivalence ($\equiv$), determinacy of weak head reductions, and crucially relies on well-formedness of semantic equality and functionality of LF typing (Lemma IV.1). □

B. Semantic Properties of Computations

If a term is semantically well-typed, then it is also syntactically well-typed. Furthermore, our definition of semantic typing is stable under weakening.

Our semantic equality definition is symmetric and transitive. It is also reflexive – however, note that we prove a weaker reflexivity statement which says that if t_1 is semantically equivalent to another term t_2 then it is also equivalent to itself. This suffices for our proofs. We also note that our semantic equality takes into account extensionality for terms at function types and contextual types; this is in fact baked into our semantic equality definition.

Lemma VII.3 (Symmetry, Transitivity, and Conversion of Semantic Equality)**.** Let $\Gamma \Vdash \breve{\tau} : u$ and $\Gamma \Vdash \breve{\tau}' : u$ and $\Gamma \Vdash \breve{\tau} = \breve{\tau}' : u$ and $\Gamma \Vdash t_1 = t_2 : \breve{\tau}$. Then:

1) (Reflexivity) $\Gamma \Vdash t_1 = t_1 : \breve{\tau}$.
2) (Symmetry) $\Gamma \Vdash t_2 = t_1 : \breve{\tau}$.
3) (Transitivity) If $\Gamma \Vdash t_2 = t_3 : \breve{\tau}$ then $\Gamma \Vdash t_1 = t_3 : \breve{\tau}$.
4) (Conversion:) $\Gamma \Vdash t_1 = t_2 : \breve{\tau}'$.

Proof. Reflexivity follows directly from symmetry and transitivity. We prove symmetry and transitivity for terms using a lexicographic induction on u and $\Gamma \Vdash \tau : u$; we appeal to the induction hypothesis and use the corresponding properties on types if the universe is smaller; if the universe stays the same, then we may appeal to the property for terms if $\Gamma \Vdash \tau : u$ is smaller; to prove conversion and symmetry for types, we may also appeal to the induction hypothesis if $\Gamma \Vdash \tau' : u$ is smaller. □

Finally we establish various elementary properties about our semantic definition that play a key role in the fundamental lemma which we prove later.

Semantic Equality for Types: $\boxed{\Gamma \Vdash \breve{\tau} = \breve{\tau}' : u}$ recursively defined by recursion on $\Gamma \Vdash \tau : u$

Semantic Equality for Terms: $\boxed{\Gamma \Vdash t = t' : \breve{\tau}}$ by recursion on $\Gamma \Vdash \breve{\tau} : u$

$$\frac{}{\Gamma \Vdash \mathsf{ctx} = \mathsf{ctx} : u} \quad \frac{\Gamma \vdash \tau' \searrow u' : u}{\Gamma \Vdash u' = \tau' : u} \quad \frac{\Gamma \vdash \tau' \searrow \lceil T' \rceil : u \qquad \Gamma \vdash T \equiv T'}{\Gamma \Vdash \lceil T \rceil = \tau' : u} \quad \frac{\Gamma \vdash \tau' \searrow x\ \vec{s} : u \quad \Gamma \vdash x\ \vec{t} \equiv x\ \vec{s} : u}{\Gamma \Vdash x\ \vec{t} = \tau' : u}$$

$$\frac{\begin{array}{c}\Gamma \vdash \tau' \searrow (y' : \breve{\tau}_1') \Rightarrow \tau_2' : u \qquad \forall \Gamma' \leq_\rho \Gamma.\ \Gamma' \Vdash \{\rho\}\breve{\tau}_1 = \{\rho\}\breve{\tau}_1' : u_1 \\ \forall \Gamma' \leq_\rho \Gamma.\ \Gamma' \Vdash s = s' : \{\rho\}\breve{\tau}_1 \Longrightarrow \Gamma' \Vdash \{\rho, s/y\}\tau_2 = \{\rho, s'/y'\}\tau_2' : u_2\end{array}}{\Gamma \Vdash (y : \breve{\tau}_1) \Rightarrow \tau_2 = \tau' : u}\ (u_1, u_2, u) \in \mathfrak{R}$$

$$\frac{\Gamma \vdash \Psi \equiv \Psi' : \mathsf{ctx}}{\Gamma \Vdash \Psi = \Psi' : \mathsf{ctx}} \quad \frac{\Gamma \vdash t \searrow w : \lceil \Psi \vdash A \rceil \quad \Gamma \vdash t' \searrow w' : \lceil \Psi \vdash A \rceil \quad \Gamma ; \Psi \Vdash \lfloor w \rfloor_{\mathsf{id}} = \lfloor w' \rfloor_{\mathsf{id}} : A}{\Gamma \Vdash t = t' : \lceil \Psi \vdash A \rceil}$$

$$\frac{\Gamma \vdash t \searrow n : x\ \vec{s} \quad \Gamma \vdash t' \searrow n' : x\ \vec{s} \quad \mathsf{wne}\ n, n' \quad \Gamma \vdash n \equiv n' : x\ \vec{s}}{\Gamma \Vdash t = t' : x\ \vec{s}}$$

$$\frac{\Gamma \vdash t \searrow w : (y : \breve{\tau}_1) \Rightarrow \tau_2 \quad \Gamma \vdash t' \searrow w' : (y : \breve{\tau}_1) \Rightarrow \tau_2 \quad \forall \Gamma' \leq_\rho \Gamma.\ \Gamma' \Vdash s = s' : \{\rho\}\breve{\tau}_1 \Longrightarrow \Gamma' \Vdash \{\rho\}w\ s = \{\rho\}w'\ s' : \{\rho, s/y\}\tau_2}{\Gamma \Vdash t = t' : (y : \breve{\tau}_1) \Rightarrow \tau_2}$$

Fig. 13. Semantic Equality for Computations

Lemma VII.4 (Neutral Soundness)**.**
If $\Gamma \Vdash \breve{\tau} : u$ and $\Gamma \vdash t : \breve{\tau}$ and $\Gamma \vdash t' : \breve{\tau}$ and $\Gamma \vdash t \equiv t' : \breve{\tau}$ and $\mathsf{wne}\ t, t'$ then $\Gamma \Vdash t = t' : \breve{\tau}$.

Proof. By induction on $\Gamma \Vdash \tau : u$. ☐

Lemma VII.5 (Backwards Closure for Computations)**.** If $\Gamma \Vdash t_1 = t_2 : \breve{\tau}$ and $\Gamma \vdash t_1 \searrow w : \breve{\tau}$ and $\Gamma \vdash t_1' \searrow w : \breve{\tau}$ then $\Gamma \Vdash t_1' = t_2 : \breve{\tau}$.

Proof. We analyse the definition of $\Gamma \Vdash t_1 = t_2 : \breve{\tau}$ considering different cases of $\Gamma \Vdash \breve{\tau} : u$. ☐

Lemma VII.6 (Typed Whnf is Backwards Closed)**.** If $\Gamma \vdash t \searrow w : (y : \breve{\tau}_1) \Rightarrow \tau_2$ and $\Gamma \vdash s : \breve{\tau}_1$ and $\Gamma \vdash w\ s \searrow v : \{s/y\}\tau_2$ then $\Gamma \vdash t\ s \searrow v : \{s/y\}\tau_2$.

Proof. By unfolding the definitions and considering different cases for w. ☐

Lemma VII.7 (Semantic Application)**.** If $\Gamma \Vdash t = t' : (y : \breve{\tau}_1) \Rightarrow \tau_2$ and $\Gamma \Vdash s = s' : \breve{\tau}_1$ then $\Gamma \Vdash t\ s = t'\ s' : \{s/y\}\tau_2$

Proof. Using well-formedness of semantic equality, Backwards closed properties (Lemma VII.6 and VII.5), and Symmetry of semantic equality (Lemma Prop. 2). ☐

VIII. Validity in the Model

For normalization, we need to establish that well-typed terms are logically related. However, as we traverse well-typed terms, they do not remain closed. As is customary, we now extend our logical relation to substitutions defining semantic substitutions $\boxed{\Gamma \Vdash \theta = \theta' : \Gamma}$:

$$\frac{\vdash \Gamma'}{\Gamma' \Vdash \cdot = \cdot : \cdot} \qquad \frac{\begin{array}{ll}\Gamma' \Vdash \theta = \theta' : \Gamma & \Gamma' \Vdash \{\theta\}\breve{\tau} = \{\theta'\}\breve{\tau} : u \\ \Gamma' \Vdash \{\theta\}\breve{\tau} : u & \Gamma' \Vdash t = t' : \{\theta\}\breve{\tau}\end{array}}{\Gamma' \Vdash \theta, t/x = \theta', t'/x : \Gamma, x{:}\breve{\tau}}$$

Semantic substitutions are well-formed (i.e. they imply that substitutions are well-typed), stable under weakening and preserve equivalences. They are also reflexive, symmetric, and transitive. Further, given a valid context where each of the declarations is valid, we can always generate $\Gamma \Vdash \mathsf{id}(\Gamma) = \mathsf{id}(\Gamma) : \Gamma$, where id is the identity substitution.

Last, we define validity of computation-level types and terms referring to their semantic definitions (Fig. 14). This allows us to define compactly the fundamental lemma which states that well typed terms correspond to valid terms in our model. Validity here is defined in terms of the semantic definition (Fig. 13). In particular, we say that two terms M and N are equal in our model, if for all computation-level instantiations θ and θ' which are considered semantically equal, we have that $\{\theta\}M$ and $\{\theta'\}N$ are semantically equal. Our definition is symmetric and transitive.

Lemma VIII.1 (Function Type Injectivity is valid.)**.** If $\Gamma \models (y : \breve{\tau}_1) \Rightarrow \tau_2 = (y : \breve{\tau}_1') \Rightarrow \tau_2' : u_3$ then $\Gamma \models \breve{\tau}_1 = \breve{\tau}_1' : u_1$ and $\Gamma, y{:}\breve{\tau}_1 \models \tau_2 = \tau_2' : u_2$ and $(u_1,\ u_2,\ u_3) \in \mathfrak{R}$

Proof. Proof by unfolding the semantic definitions. ☐

The fundamental lemma (Lemma VIII.1) states that well-typed terms are valid. The proof proceeds by mutual induction on the typing derivation for LF-objects and computations. It relies on the validity of type conversion, computation-level functions, applications, and recursion. To establish these properties, we require symmetry, transitivity of semantic equality, and semantic type conversion (Lemma VII.3).

Theorem VIII.1 (Fundamental Theorem)**.**

1) If $\vdash \Gamma$ then $\models \Gamma$.
2) If $\Gamma; \Psi \vdash M : A$ then $\Gamma; \Psi \models M = M : A$.
3) If $\Gamma; \Psi \vdash \sigma : \Phi$ then $\Gamma; \Psi \models \sigma = \sigma : \Phi$.
4) If $\Gamma; \Psi \vdash M \equiv N : A$ then $\Gamma; \Psi \models M = N : A$.
5) If $\Gamma; \Psi \vdash \sigma \equiv \sigma' : \Phi$ then $\Gamma; \Psi \models \sigma = \sigma' : \Phi$.

Validity of LF Objects : $\boxed{\Gamma; \Psi \models M = N : A}$ where $\models \Gamma$

$$\frac{\begin{array}{l}\forall \Gamma', \theta, \theta'. \Gamma' \Vdash \theta = \theta' : \Gamma \\ \qquad \Longrightarrow \Gamma'; \{\theta\}\Psi \Vdash \{\theta\}M = \{\theta'\}N : \{\theta\}A\end{array}}{\Gamma; \Psi \models M = N : A}$$

Validity of LF Substitutions : $\boxed{\Gamma; \Psi \models \sigma = \sigma' : \Phi}$ where $\models \Gamma$

$$\frac{\begin{array}{l}\forall \Gamma',\ \theta,\ \theta'. \Gamma' \Vdash \theta = \theta' : \Gamma \\ \qquad \Longrightarrow \Gamma'; \{\theta\}\Psi \Vdash \{\theta\}\sigma_1 = \{\theta'\}\sigma' : \{\theta\}\Phi\end{array}}{\Gamma; \Psi \models \sigma = \sigma' : \Phi}$$

Validity of Types : $\boxed{\Gamma \models \breve{\tau} = \breve{\tau}' : u}$ and $\boxed{\Gamma \models \breve{\tau} : u}$

$$\frac{\begin{array}{l}\forall \Gamma',\ \theta,\ \theta'. \Gamma' \Vdash \theta = \theta' : \Gamma \\ \qquad \Longrightarrow \Gamma' \Vdash \{\theta\}\breve{\tau} = \{\theta'\}\breve{\tau}' : u\end{array}}{\Gamma \models \breve{\tau} = \breve{\tau}' : u} \qquad \frac{\Gamma \models \breve{\tau} = \breve{\tau} : u}{\Gamma \models \breve{\tau} : u}$$

Validity of Terms : $\boxed{\Gamma \models t = t' : \breve{\tau}}$ and $\boxed{\Gamma \models t : \breve{\tau}}$

$$\frac{\begin{array}{ll}\models \Gamma & \forall \Gamma', \theta, \theta'. \Gamma' \Vdash \theta = \theta' : \Gamma \\ \Gamma \models \breve{\tau} : u & \quad \Longrightarrow \Gamma' \Vdash \{\theta\}t = \{\theta'\}t' : \{\theta\}\breve{\tau}\end{array}}{\Gamma \models t = t' : \breve{\tau}} \qquad \frac{\Gamma \models t = t : \breve{\tau}}{\Gamma \models t : \breve{\tau}}$$

Fig. 14. Validity Definition

6) If $\Gamma \vdash t : \tau$ then $\Gamma \models t : \tau$.
7) If $\Gamma \vdash t \equiv t' : \tau$ then $\Gamma \models t = t' : \tau$.

Proof. By induction on the first derivation using validity of application, functions, recursion, and type conversion, Backwards Closed (VII.5), Well-formedness of Semantic Typing , Semantic weakening. □

Theorem VIII.2 (Normalization and Subject Reduction)**.** If $\Gamma \vdash t : \tau$ then $t \searrow w$ and $\Gamma \vdash t \equiv w : \tau$

Proof. By the Fundamental theorem (Lemma VIII.1), we have $\Gamma \Vdash t = t : \tau$ (choosing the identity substitution for θ and θ'). This includes a definition $t \searrow w$. Since w is in whnf (i.e. $\mathsf{whnf}\ w$), we have $w \searrow w$. Therefore, we can easily show that also $\Gamma \Vdash t = w : \tau$. By well-formedness , we also have that $\Gamma \vdash t \equiv w : \tau$ and more specifically, $\Gamma \vdash w : \tau$. □

Using the fundamental lemma, we can also show that every term has a unique type. This requires first showing some standard inversion lemmas and then showing function type injectivity.

Lemma VIII.2 (Injectivity of Function Type)**.** If $\Gamma \vdash (y : \breve{\tau}_1) \Rightarrow \tau_2 \equiv (y : \breve{\tau}_1') \Rightarrow \tau_2' : u$ then $\Gamma \vdash \breve{\tau}_1 \equiv \breve{\tau}_1' : u_1$ and $\Gamma, y : \breve{\tau}_1 \vdash \tau_2 \equiv \tau_2 : u_2$ and $(u_1, u_2, u) \in \mathfrak{R}$.

Proof. By the fundamental theorem (Lemma VIII.1) $\Gamma \Vdash (y : \breve{\tau}_1) \Rightarrow \tau_2 \equiv (y : \breve{\tau}_1') \Rightarrow \tau_2' : u$ (choosing the identity substitution for θ and θ'). By the sem. equality def., we have $\Gamma \Vdash \breve{\tau}_1 = \breve{\tau}_1' : u_1$ and $\Gamma, y : \breve{\tau}_1 \Vdash \tau_2 = \tau_2' : u_2$ and $(u_1, u_2, u) \in \mathfrak{R}$. By well-formedness of semantic typing , we have $\Gamma \vdash \breve{\tau}_1 \equiv \breve{\tau}_1' : u_1$ and and $\Gamma, y : \breve{\tau}_1 \vdash \tau_2 \equiv \tau_2' : u_2$ □

Theorem VIII.3 (Type Uniqueness)**.**

1) If $\Gamma; \Psi \vdash M : A$ and $\Gamma; \Psi \vdash M : B$ then $\Gamma \vdash A \equiv B :$ type.
2) If $\Gamma \vdash t : \breve{\tau}$ and $\Gamma \vdash t : \breve{\tau}'$ then $\Gamma \vdash \breve{\tau} \equiv \breve{\tau}' : u$.

Proof. By mutual induction on the typing derivation exploiting typing inversion lemmas. □

Last but not least, the fundamental lemma allows us to show that not every type is inhabited and thus COCON can be used as a logic. To establish this stronger notion of consistency, we first prove that we can discriminate type constructors.

Lemma VIII.3 (Type Constructor Discrimination)**.** Neutral types, sorts, and function types are can be discriminated.

Proof. Proof by contradiction. To show for example that $\Gamma \vdash x\ \vec{t} \neq (y : \breve{\tau}_1) \Rightarrow \tau_2$, we assume $\Gamma \vdash x\ \vec{t} \equiv (y : \breve{\tau}_1) \Rightarrow \tau_2 : u$. By the fundamental lemma (Lemma VIII.1), we have $\Gamma \Vdash x\ \vec{t} \equiv (y : \breve{\tau}_1) \Rightarrow \tau_2 : u$ (choosing the identity substitution for θ and θ'); but this is impossible given the semantic equality definition. □

Theorem VIII.4 (Consistency)**.** $x : u_0 \not\vdash t : x$.

Proof. Assume $x : u_0 \vdash t : x$. By subject reduction (Lemma VIII.2), there is some w s.t. $t \searrow w$ and $\Gamma \vdash t \equiv w : x$ and in particular, we must have $\Gamma \vdash w : x$. As x is neutral, it cannot be equal to u, $(y : \breve{\tau}_1) \Rightarrow \tau_2$, or $\lceil T \rceil$ (Lemma VIII.3). Thus w can also not be a sort, function, or contextual object. Hence, w can only be neutral, i.e. given the assumption $x : u_0$, the term w must be x. This implies that $\Gamma \vdash x : x$ and implies $\Gamma \vdash x \equiv u_0 : u$ by inversion lemma for typing. But this is impossible by Lemma VIII.3. □

An extended version with the full technical development is available at Pientka et al. (2019).

IX. Conclusion

COCON is a first step towards integrating LF methodology into Martin-Löf style dependent type theories and and bridges the longstanding gap between these two worlds. We have established type uniqueness, normalization, and consistency. The next immediate step is to derive an equivalence algorithm based on weak head reduction and show its completeness. We expect that this will follow a similar Kripke-style logical relation as the one we described. This would allow us to justify that type checking COCON programs is decidable.

It should be possible to implement COCON as an extension to BELUGA– from a syntactic point of view, it would be a small change. It also seems possible to extend existing implementation of Agda, however this might be more work, as in this case one needs to implement the LF infrastructure.

References

Abel, A., Öhman, J., and Vezzosi, A. (2018). Decidability of conversion for type theory in type theory. *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'18)*, PACMPL 2(POPL):23:1–23:29.

Abel, A. and Scherer, G. (2012). On irrelevance and algorithmic equality in predicative type theory. *Logical Methods in Computer Science*, 8(1).

Bertot, Y. and Castéran, P. (2004). *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer.

Cave, A. and Pientka, B. (2013). First-class substitutions in contextual type theory. In *8th ACM SIGPLAN International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'13)*, pages 15–24. ACM.

Despeyroux, J., Felty, A. P., and Hirschowitz, A. (1995). Higher-order abstract syntax in coq. In Dezani-Ciancaglini, M. and Plotkin, G. D., editors, *2nd International Conference on Typed Lambda Calculi and Applications (TLCA '95)*, Lecture Notes in Computer Science (LNCS 902), pages 124–138. Springer.

Despeyroux, J. and Leleu, P. (1999). Primitive recursion for higher order abstract syntax with dependent types. In *International Workshop on Intuitionistic Modal Logics and Applications (IMLA)*.

Despeyroux, J., Pfenning, F., and Schürmann, C. (1997). Primitive recursion for higher-order abstract syntax. In *Proceedings of the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)*, pages 147–163. Springer. Extended version available as Technical Report CMU-CS-96-172, Carnegie Mellon University.

Harper, R., Honsell, F., and Plotkin, G. (1993). A framework for defining logics. *Journal of the ACM*, 40(1):143–184.

Harper, R. and Pfenning, F. (2005). On equivalence and canonical forms in the LF type theory. *ACM Transactions on Computational Logic*, 6(1):61–101.

Jacob-Rao, R., Pientka, B., and Thibodeau, D. (2018). Index-stratified types. In Kirchner, H., editor, *3rdd International Conference on Formal Structures for Computation and Deduction (FSCD'18)*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Nanevski, A., Pfenning, F., and Pientka, B. (2008). Contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3):1–49.

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. Technical Report 33D.

Pientka, B. (2008). A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 371–382. ACM Press.

Pientka, B. and Abel, A. (2015). Structural recursion over contextual objects. In Altenkirch, T., editor, *13th International Conference on Typed Lambda Calculi and Applications (TLCA'15)*, pages 273–287. Leibniz International Proceedings in Informatics (LIPIcs) of Schloss Dagstuhl.

Pientka, B., Abel, A., Ferreira, F., Thibodeau, D., and Zucchini, R. (2019). Computation in contextual type theory. https://arxiv.org/submit/253219.

Pientka, B. and Cave, A. (2015). Inductive Beluga:Programming Proofs (System Description). In Felty, A. P. and Middeldorp, A., editors, *25th International Conference on Automated Deduction (CADE-25)*, Lecture Notes in Computer Science (LNCS 9195), pages 272–281. Springer.

Pientka, B. and Dunfield, J. (2008). Programming with proofs and explicit contexts. In *ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, pages 163–173. ACM.

Pientka, B. and Dunfield, J. (2010). Beluga: a framework for programming and reasoning with deductive systems (System Description). In Giesl, J. and Haehnle, R., editors, *5th International Joint Conference on Automated Reasoning (IJCAR'10)*, Lecture Notes in Artificial Intelligence (LNAI 6173), pages 15–21. Springer.

Scott, D. S. (1976). Data types as lattices. *SIAM Journal on Computing*, 5(3):522–587.

Werner, B. (1992). A normalization proof for an impredicative type system with large elimination over integers. In *International Workshop on Types for Proofs and Programs (TYPES)*, pages 341–357.

Xi, H. and Pfenning, F. (1999). Dependent types in practical programming. In *26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM Press.

Zenger, C. (1997). Indexed types. *Theoretical Computer Science*, 187(1-2):147–165.