The Münchhausen Method in Type Theory

Thorsten Altenkirch
School of Computer Science, University of Nottingham, UK

Ambrus Kaposi
Eötvös Loránd University, Budapest, Hungary

Artjoms Šinkarovs
Heriot-Watt University, Edinburgh, Scotland, UK

Tamás Végh
Eötvös Loránd University, Budapest, Hungary

Abstract

In one of his long tales, after falling into a swamp, Baron Münchhausen salvaged himself and the horse by lifting them both up by his hair. Inspired by this, the paper presents a technique to justify very dependent types. Such types reference the term that they classify, e.g. $x : F\ x$. While in most type theories this is not allowed, we propose a technique on salvaging the meaning of both the term and the type. The proposed technique does not refer to preterms or typing relations and works in a completely algebraic setting, e.g categories with families. With a series of examples we demonstrate our technique. We use Agda to demonstrate that our examples are implementable within a proof assistant.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Type structures

Keywords and phrases type theory, proof assistants, very dependent types

Digital Object Identifier 10.4230/LIPIcs.TYPES.2022.10

Supplementary Material *Software*: `https://bitbucket.org/akaposi/combinator/src/master/post-types2022`, archived at `swh:1:dir:9609c613a09bd552cbca4546d247e3cefebef9cd`

Funding *Ambrus Kaposi*: Supported by the "Application Domain Specific Highly Reliable IT Solutions" project which has been implemented with support from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme TKP2020-NKA-06 (National Challenges Subprogramme) funding scheme, and by Bolyai Scholarship BO/00659/19/3.
Artjoms Šinkarovs: Supported by the Engineering and Physical Sciences Research Council through the grant EP/N028201/1.

1 Introduction

When we want to understand how powerful the given type system is, we identify objects that the given type is allowed to depend on. For instance, in simply-typed systems types are built from a fixed set of ground types and operations. In System F we introduce type variables and binders making it possible to define new operations that compute types. In dependently-typed systems we are allowed to compute types from terms.

At the same time, we rarely explore dependencies within a typing relation. For example, consider the case when the type is allowed to depend on the term that it is typing:

$$x : F\,x$$

Such a situation is often referred to as *very dependent type* [10]. The immediate two questions arise: (i) does this ever occur in practice? (ii) how do you support this within a type system?

Let us consider an example where such a type occurs very naturally. There is a well-known type isomorphism, saying that pairs can be represented as functions from boolean:

$$A \times B \;\cong\; (b : \mathsf{Bool}) \to \mathsf{if}\, b\, \mathsf{then}\, A\, \mathsf{else}\, B.$$

Consider now upgrading the isomorphism to dependent product on the left hand side. Given $A : \mathsf{Type}$, $B : A \to \mathsf{Type}$, we want something like

$$\Sigma\, A\, B \;\cong\; (b : \mathsf{Bool}) \to \mathsf{if}\, b\, \mathsf{then}\, A\, \mathsf{else}\, (B\, \Box),$$

but what do you put in the placeholder $\Box$? It should be the output of the function when the input is $b = \mathsf{true}$. Once the function is given a name, we can refer to it:

$$f : (b : \mathsf{Bool}) \to \mathsf{if}\, b\, \mathsf{then}\, A\, \mathsf{else}\, (B\, (f\, \mathsf{true}))$$

Supporting such definitions in a type system can be tricky. Hickey gives [10] a type system with very dependent functions using pre-terms and typing relations [5]. However, it turns out that many very dependent types can be understood algebraically and even encoded in proof assistants.

Many practical examples are easier to understand when very dependent types are present. One familiar example is the use of type universes in proof assistants such as Coq or Agda. Both systems use Russell universes, and if we ignore the universe levels, Set is of type Set in Agda, and Type is of type Type in Coq. This is clearly the case of very dependent types.

Our main observation is that algebraic presentation requires cutting the cycle of a given very dependent type. This is achieved by introducing a temporary placeholder type and a number of equations that eliminate the placeholders. The proposed scheme can be summarised as follows. For a very dependent type $(x :\ \mathsf{F}\ x)$ find:

$$\mathsf{A} : \mathsf{Set};\ \mathsf{G} : \mathsf{A} \to \mathsf{Set};\ \alpha :\ \{a :\ \mathsf{A}\} \to \mathsf{G}\ a \to \mathsf{A}$$

Such that F can be decomposed in $\mathsf{G} \circ \alpha$. In this case, a very dependent type can be expressed as the following triplet:

```
a  : A       - Placeholder
x  : G a     - The data
eq : a ≡ α x - Closing the cycle
```

This approach works if these equations are propositional, but it forces a lot of transport along the newly introduced equations (this situation is commonly referred to as transport hell). In Agda we can turn these propositional equations into definitional ones by means of rewrite rules or forward declarations.

The main contribution of this paper lies in applying the Münchhausen method to five practical examples. Our setting is Martin-Löf type theory extended with function extensionality, UIP (uniqueness of identity proofs) and forward declarations. From [12] we know that such a formulation without forward declarations is conservative with respect to its intensional version. This means that, in principle, all the presented types that do not use forward declarations can be given in intensional type theory, but in a much more verbose way. We conjecture that the same holds for the type theory with forward declarations as well, but as these are not very well understood, we would not claim this.

We use Agda to demonstrate concrete implementation of our examples, but there is nothing Agda-specific in the method itself. In Agda, the Münchausen method can be realised in four different ways:

1. Identity types and explicit equations (Section 3);
2. Forward declarations (Sections 3, 4 and 5);
3. Shallow embedding as described in [14] (Sections 6 and 6);
4. Postulates and rewrite rules (Section 7)

While it is not yet clear whether all very dependent types as defined in [10] can be handled by the proposed method, we believe that the examples that we provide give a first step towards answering this question.

This paper is an Agda script, therefore all the examples in the paper have been typechecked.

The content of this paper was presented at the TYPES'22 conference in Nantes [3].

2 Background

In this section we give a brief introduction to Agda, which is an implementation of Martin-Löf's dependent type theory [16] extended with a number of constructions such as inductive data types, records, modules, *etc.* We make a brief overview of the features that are used in this paper. For the in-depth introduction please refer to Agda's user manual [1].

2.1 Datatypes

Datatypes are defined as follows:

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} → Fin n → Fin (suc n)
```

The type ℕ of unary natural numbers is a datatype with two constructors: zero and suc. The type of ℕ is Set which is Agda's builtin type of small types.

The type Fin is indexed by ℕ and it also has two constructors zero and suc. The names of the constructors can overlap. In the definition of the Fin constructors we used implicit argument syntax[1] to define the variable n. When using constructors of Fin, we can leave out specifying these arguments relying on Agda's automatic inference. These can be also passed explicitly as follows:

```
a : Fin 2
a = zero {n = 1}
```

Numbers 0, 1, 2, . . . are implicitly mapped into ℕ in the usual way.

2.2 Records

Agda makes it possible to define records[2]. They generalise dependent products, making it possible to name the fields. For example, we can define the type of dependent pairs using records as follows:

[1] https://agda.readthedocs.io/en/v2.6.3/language/implicit-arguments.html
[2] https://agda.readthedocs.io/en/v2.6.3/language/record-types.html

```
record Pair (A : Set) (B : A → Set) : Set where
  constructor _,_
  field
    fst : A
    snd : B fst
```

The Pair record is parametrised by the type A and the family B (over A). The type has two fields named fst and snd that correspond to first and second projections of the dependent pair. Finally, we can give a constructor _,_ that we can use to construct the values of type Pair. Note that the constructor uses the mixfix notation[3]. This means that arguments replace the underscores, so the comma , becomes a binary operation. The values can be constructed as follows:

```
b : Pair ℕ Fin
b = 5 , zero
```

2.3 Modules

Modules[4] make it possible to collect the definitions that logically belong together, giving them a separate namespace. Modules can accept parameters. They abstract variables for the definitions within the module. In the paper we only use modules to group the definitions together and reuse the names of the definitions. For example, here we define modules X and Y, where Y is parametrised with the variable n, which is a natural number.

```
module X where
  foo : ℕ
  foo = 5

module Y (n : ℕ) where
  foo : ℕ
  foo = n
```

2.4 Forward Declarations

Agda makes it possible[5] to make a declaration and provide a definition later. This is useful when dealing with mutual definitions. For example, we can have a mutual definition of even and odd numbers as the following indexed types:

```
data Even : ℕ → Set
data Odd : ℕ → Set

data Even where
  zero : Even zero
  suc : {n : ℕ} → Odd n → Even (suc n)
data Odd where
  suc : {n : ℕ} → Even n → Odd (suc n)
```

[3] https://agda.readthedocs.io/en/v2.6.3/language/mixfix-operators.html
[4] https://agda.readthedocs.io/en/v2.6.3/language/module-system.html
[5] https://agda.readthedocs.io/en/v2.6.3/language/mutual-recursion.html#mutual-recursion-forward-declaration

First we defined the signature of both data types; after that we gave the definitions of their constructors. By making such a forward declaration, we were able to refer Odd in the definition of the suc constructor in Even.

2.5 Postulates

Agda makes it possible[6] to declare objects without ever providing a definition. This can be thought of as a typed free variable. For example, we can postulate that there exists some natural number q:

```
postulate
  q : ℕ
```

2.6 Rewrite Rules

Agda makes it possible to define rewrite rules[7], which are typically used to turn propositional equations into definitional ones. However, in combination with postulates, we can also simulate some reduction behaviour. For example, we can postulate natural numbers, eliminator for natural numbers and reduction equalities. Then we can use rewrite rules to simulate reduction.

```
postulate
  Nat : Set
  z : Nat
  s : Nat → Nat
  elim : (P : Nat → Set) → P z → ((n : Nat) → P n → P (s n))
       → (n : Nat) → P n
  elim-z : ∀ {P pz ps} → elim P pz ps z ≡ pz
  elim-s : ∀ {P pz ps n} → elim P pz ps (s n) ≡ ps n (elim P pz ps n)
  {-# REWRITE elim-z elim-s #-}
```

We postulate the type for natural numbers Nat and its two constructors z and s. After that, we postulate the type for the eliminator for natural numbers in the usual way. Finally we define two β-like equalities for the eliminator. By turning these equalities into rewrite rules, we make our eliminator to reduce in the usual way.

3 Dependent Sequences

We start with a detailed exploration of the dependent product isomorphism presented in the introduction. While this example is not very practical, it is concise and easy to understand.

For a fixed pair of types, the encoding of non-dependent pair can be expressed in Agda as follows:

```
pair : (b : Bool) → if b then String else ℕ
pair true  = "Types" - first projection
pair false = 22      - second projection
```

[6] https://agda.readthedocs.io/en/v2.6.3/language/postulates.html
[7] https://agda.readthedocs.io/en/v2.6.3/language/rewriting.html

Surprisingly, similar presentation of dependent pairs for a fixed type and a family over it is expressible in Agda using forward declarations. For example, for ℕ and Fin we have:

```
dpair-hlpr : ℕ
dpair : (b : Bool) → if b then ℕ else Fin dpair-hlpr
dpair-hlpr = dpair true

dpair true  = 5    - first projection of type ℕ
dpair false = # 3 - second projection of type Fin 5
```

According to Münchhausen method, we "cut" the cyclic dependency of *dpair* by introducing a placeholder called *dpair-hlpr*. Here forward declarations make it possible to postpone the definition of *dpair-hlpr*. After that, we define *dpair* and we "close the cycle" by giving the value to the placeholder.

Let us try to abstract this encoding to arbitrary types, and prove the isomorphism from the introduction. For non-dependent pairs, we have:

```
module _ (ext : ∀ {a b} → Extensionality a b) where
  Pair : Set → Set → Set
  Pair A B = (b : Bool) → if b then A else B

  Pair≅× : ∀ A B → (A × B) ↔ Pair A B
  Pair≅× A B = mk↔ {f = to}{from} (to∘from , λ _ → refl)
    where
      to : _; from : _; to∘from : _
      to (a , b) = λ {true → a; false → b}
      from     f = f true , f false
      to∘from f = ext λ {true → refl; false → refl}
```

The ↔ is a type for bijections, and mk↔ constructs the bijection from forward and backward functions and a pair of proofs that they are inverses of each other. As can be seen, conversion from Pair is memoisation. Correspondingly, conversion into Pair is "unmemoisation". These operations are clearly inverses of each other, assuming functional extensionality.

Encoding of dependent pairs has to mention the placeholder h and the value that this placeholder gets (f true) by means of explicit equation eq.

```
record DPair (A : Set) (B : A → Set) : Set where
  constructor _▷_[_]
  field
    h  : A
    f  : (b : Bool) → if b then A else B h
    eq : h ≡ f true
```

Such an encoding corresponds to the first variant of the Münchhausen method, as the equality that closes the cycle is made explicit. Note that eq corresponds to the definition of *dpair-hlpr* in the presentation above.

The isomorphism between dependent pairs and DPair requires a little bit more work, as we are dealing with equations within the structure. Assuming functional extensionality ext and uniqueness of identity proofs uip, equality of two DPairs can be derived from point-wise pair equality given by $\_\equiv^d\_$.

```
module _ (ext : ∀ {a b} → Extensionality a b)
         (uip : ∀ {A : Set}{a b : A} → (p q : a ≡ b) → p ≡ q) where

  record _≡ᵈ_ {A}{B} (a b : DPair A B) : Set where
    constructor _&_
    field
      fst  : DPair.h a ≡ DPair.h b
      snd : DPair.f a false ≡ subst B (sym fst) (DPair.f b false)

  ≡ᵈ⇒≡ : ∀ {A B} {a b : DPair A B} → a ≡ᵈ b → a ≡ b
```

With these definitions at hand, the isomorphism between DPairs and Σ types is very similar to its non-dependent version:

```
  Pair≅Σ : ∀ A B → (Σ A B) ↔ DPair A B
  Pair≅Σ A B = mk↔ {f = to}{from} (to∘from , λ _ → refl)
    where
      to (a , b) = a ▹ (λ {true → a; false → b}) [ refl ]
      from (h ▹ f [ eq ]) = f true , subst B eq (f false)
      to∘from (h ▹ f [ eq ]) = ≡ᵈ⇒≡ (sym eq & cong (λ x → subst B x (f false)) (sym∘sym eq))
```

As can be seen, working with explicit equalities is tricky. Switching to more powerful type theories (*e.g.* cubical type theory) would eliminate the necessity to use axioms, but it would not solve the transport hell problem. The *pair* example works so nicely, because we essentially turned the propositional equality into the definitional one.

3.1 Infinite Sequences

In the type theory proposed by Hickey, the only extension to the standard type theory is addition of very dependent *functions*. It is observed that (very) dependent records can be always presented as very dependent functions by choosing a domain type that enumerates the fields. This is essentially what the example with dependent pairs does – Σ type has two fields that are enumerated by booleans.

However, dependent functions can do more than that, as their domain does not have to be finite. Let us now consider such an infinite case by defining non-increasing infinite sequences. With a little abuse of notation, we can present those as the following very dependent type.

```
 - ↓-seq : (n : Nat) → if n == 0 then ℕ else Fin (1 + ↓-seq (n - 1))
```

The same Münchhausen technique with forward declarations can be used to define such a function. We start by forward declaring Ty (expression on the right hand side of the arrow in the type above) and its interpretation I into natural numbers.

```
 Ty : ℕ → Set
 I : ∀ n → Ty n → ℕ
```

At the same time we forward declare the actual sequence that we want to define:

```
 ↓-seq : (n : ℕ) → Ty n
```

The type of the elements in the sequence is defined inductively as follows: for zero we have ℕ, the successor case gives us Fin of whatever the interpretation of the sequence that we are defining at predecessor is going to return us.

```
Ty 0       = ℕ
Ty (suc n) = Fin $ suc $ I n (↓-seq n)
```

The interpretation of the elements at the given sequence is straight-forward: zero case has a natural number that we return; elements of Fin types are casted into natural numbers.

```
I 0 n = n
I (suc n) i = toℕ i
```

Finally, we define the actual data of our non-increasing infinite sequence.

```
↓-seq 0 = 5
↓-seq 1 = # 3
↓-seq 2 = # 2
↓-seq (suc (suc (suc n))) = # 0
```

Notice that in this particular case, the types of the elements in sequence only depend on the previous element. We can imagine full induction, where the element can depend on all the previously defined elements. In this case induction-recursion becomes crucially important to generate an n-fold dependent type.

4 Multi-dimensional Arrays

The next example we consider is a type for multi-dimensional arrays that are commonly found in array languages such as APL [13]. Arrays can be thought of as n-dimensional rectangles, where the size of the rectangle is given by the *shape*, which is a vector of natural numbers describing extents along each dimension. Array languages follow the slogan "everything is an array", treating natural numbers and shape vectors as arrays. Natural numbers are 0-dimensional arrays, *e.g.* their shape is the empty vector. Shape vectors are 1-dimensional arrays, *e.g.* their shapes are 1-element vectors.

The problem with capturing this construction with inductive types is the following circularity. Array types depend on shapes, but the shapes are arrays. That is, the index of the type is the very type that we are defining.

4.1 Unshaped arrays

One way to define the array type inductively is to avoid the shape argument entirely. This construction is proposed by Jenkins [9]:

```
module Unshaped where
  data Ar : Set where
    z       : Ar           - Natural numbers with zero (z)
    s       : Ar → Ar      -   and successor (s)
    []      : Ar           - Cons lists with empty list ([])
    _::_    : Ar → Ar → Ar -   and cons operation (_::_).
    reshape : Ar → Ar → Ar - Multi-dimensional array constructor.
```

With these definitions we get a closed universe of arrays of natural numbers. On the positive side, we obtained the uniformity of arrays as in APL – if a function expects an array, it is possible to pass a number or a vector without any casting.

```
0ₐ 1ₐ 2ₐ 3ₐ : Ar        - Natural numbres
0ₐ = z; 1ₐ = s 0ₐ; 2ₐ = s 1ₐ; 3ₐ = s 2ₐ

v₂ v₄ mat₂₂ : Ar
v₂ = 2ₐ :: 2ₐ :: []     - Vector [2,2]
v₄ = 1ₐ :: 0ₐ           - Flattened identity matrix [[1,0],[0,1]]
  :: 0ₐ :: 1ₐ :: []
mat₂₂ = reshape v₂ v₄ - Identity matrix of shape [2,2]
```

On the negative side, our array type does not enforce any shape invariants. That is, we can produce non-rectangular arrays such as:

```
weird₁ = reshape (2ₐ :: 2ₐ :: []) (1ₐ :: 2ₐ :: 3ₐ :: [])
weird₂ = (3ₐ :: []) :: weird₁
```

While it might be possible to define the meaning for such cases, normally they are considered type errors. We could also try restricting these constructions with refinement types, but we are interested in intrinsically-typed solution instead.

4.2 Inductive-inductive

Intrinsically-typed array universe can be defined using inductive-inductive types, following the ideas from [18]. We define arrays and shapes mutually.

```
module Univ where
  data Sh : Set
  data Ar : Sh → Set
  data Sh where
    scal  : Sh
    vec   : Ar scal → Sh
    mda : ∀ {s} → Ar (vec s) → Sh
```

The shapes form the following hierarchy: scalars (*e.g.* natural numbers) have a unit shape; vector shapes are parametrised by scalars; multi-dimensional shapes are parametrised by vectors.

```
  Nat : Set
  Nat = Ar scal

  Vec : Nat → Set
  Vec n = Ar (vec n)

  prod : ∀ {n} → Vec n → Nat
```

We define names Nat and Vec which are synonyms for arrays of the corresponding shape. We also make a forward declaration of the *prod* function that computes the product of the given vector. Now we are ready to define the array universe as follows:

```
  data Ar where
    z        : Nat
    s        : Nat → Nat
```

```
[]      : Vec z
_::_    : ∀ {n} → Nat → Vec n → Vec (s n)
reshape : ∀ {n} → (s : Vec n) → Vec (prod s) → Ar (mda s)
```

We use exactly the same constructors as before, except vectors are indexed by their length, and multi-dimensional arrays are indexed by shape vectors. Also, the reshape constructor has a coherence condition saying that the number of elements (*prod s*) in the vector we are reshaping matches the new shape *s*.

We complete the definition of *prod*, expressing it as a fold with multiplication $\_*_n\_$ (defined as usual, not shown here).

```
0ₐ 1ₐ 2ₐ 3ₐ : Nat
0ₐ = z; 1ₐ = s z; 2ₐ = s 1ₐ; 3ₐ = s 2ₐ; 4ₐ = s 3ₐ

prod [] = 1ₐ
prod (x :: xs) = x *ₙ prod xs
```

Vector and matrix examples can be expressed as follows.

```
v₂ : Vec 2ₐ
v₂ = 2ₐ :: 2ₐ :: []

v₄ : Vec 4ₐ
v₄ = 1ₐ :: 0ₐ
   :: 0ₐ :: 1ₐ :: []

mat₂₂ : Ar (mda v₂)
mat₂₂ = reshape v₂ v₄
```

While the numbers, vectors and arrays are elements of the same universe, we did not achieve the desired array uniformity. The problem is that we maintain the distinction between arrays and shapes, even though morally they are the same thing. For example, the type of mat_{22} is Ar (mda v_2), not Ar v_2. Also, the expression reshape [] (1_a :: []) cannot be typed as Nat, even though it is an array of the empty shape.

4.3 Münchhausen universe

In order to resolve the lack of uniformity, we use the Münchhausen method (the variant with forward declarations). Our goal is to equate Ar and Sh. Therefore, we forward-declare Sh as a placeholder to bootstrap the array type. After that we close the cycle by defining Sh to be Ar with a certain index.

We start with forward-declaring types N (natural numbers) and shapes Sh that are indexed by natural numbers. Both of these types are placeholders that we will eliminate later.

```
module UniformUniv where
  N : Set
  Sh : N → Set
```

The array type is a concrete definition, whereas its parameter is a placeholder type.

```
  data Ar : ∀ {n} → Sh n → Set
```

We make forward declarations of N and Sh constructors that are needed to fill-in the indices of the array type. Note that N constructors are used to fill-in the indices of Sh constructors.

```
z' : N
s' : N → N
[]' : Sh z'
_::'_ : ∀ {n} → N → Sh n → Sh (s' n)
```

We define names Nat and Vec for 0-dimensional and 1-dimensional arrays correspondingly. We also make a forward declaration of *prod* as before, except we use placeholder types.

```
Nat : Set
Nat = Ar []'

Vec : N → Set
Vec n = Ar (n ::' []')

prod : ∀ {n} → Sh n → N
```

Now we can define an array universe, exactly as before.

```
data Ar where
  z       : Nat
  s       : Nat → Nat
  []      : Vec z'
  _::_    : ∀ {n} → Nat → Vec n → Vec (s' n)
  reshape : ∀ {n} → (s : Sh n) → Vec (prod s) → Ar s
```

Finally, we eliminate the placeholder types by equating N and Sh with 0-dimensional and 1-dimensional arrays correspondingly. After we do this, we define the placeholder constructors to be those defined in Ar.

```
N    = Ar []'
Sh n = Ar (n ::' []')
z' = z; s' = s; []' = []; _::'_ = _::_
```

This closes the cycle and turns Ar into a very dependent type that is witnessed by the following Agda expression:

```
_ : ∀ {n : Ar []} → (s : Ar (n :: [])) → Set
_ = Ar
```

As expected, our examples are definable, and 1-dimensional arrays can be immediately used as array shapes.

```
0ₐ 1ₐ 2ₐ 3ₐ 4ₐ : Nat
0ₐ = z; 1ₐ = s 0ₐ; 2ₐ = s 1ₐ; 3ₐ = s 2ₐ; 4ₐ = s 3ₐ

v₂ : Ar (2ₐ :: [])
v₂ = 2ₐ :: 2ₐ :: []

v₄ : Ar (4ₐ :: [])
v₄ = 1ₐ :: 0ₐ :: 0ₐ :: 1ₐ :: []
```

One technical drawback that we ran into is that with such a cyclic type, Agda loops when attempting to define pattern-matching functions. The loop happens when solving the unification problem – it has to check that two arrays type match. As array types are indexed, Agda has to unify the indices, which triggers unifying the type of the indices, and so on. As a workaround, we can define eliminators via rewrite rules, which makes it possible to define *prod*. The rest works as expected.

```
prod {n} xs = sh-elim _ 1ₐ (λ n x xs r → x *ₙ r) n xs

mat₂₂ : Ar v₂
mat₂₂ = reshape v₂ v₄

scal-test : Nat
scal-test = reshape [] (1ₐ ∷ [])
```

5 Russell Universes

In pure type systems [5] there is no separate sort for terms and types, there are only terms and those terms which appear on the right hand side of the colon in the typing relation are called types. Using well-typed terms, this would lead to the following very dependent type for the sort of terms: Tm : (Γ : Con) → Tm Γ U → Set. That is, terms depend on a context and a term of type U. Using the Münchhausen method (its variant with forward declarations), we can make sense of this. We temporarily introduce types and the type U' for the universe, then after declaring the sort of terms we can say that actually types are just terms of type U', then we can add the actual U operator for terms and close the loop by saying that U' is the same as U. Using forward declarations, part of the syntax of type theory is given as follows.

```
data Con : Set
Ty : Con → Set    -- forward declaration
data Con where
  •     : Con
  _▹_ : (Γ : Con) → Ty Γ → Con
U' : ∀ {Γ} → Ty Γ -- forward declaration
data Tm : (Γ : Con) → Ty Γ → Set
Ty Γ = Tm Γ U'
data Tm where
  U    : ∀ {Γ} → Ty Γ
  Π    : ∀ {Γ} → (A : Ty Γ) → Ty (Γ ▹ A) → Ty Γ
  lam  : ∀ {Γ A B} → Tm (Γ ▹ A) B → Tm Γ (Π A B)
U' = U
```

Note that such a theory is inconsistent through Russell's paradox, but it is easy to fix this by stratification (adding natural number indices to Ty and U, see e.g. [15]). More precisely, we say that a stratified category with families (CwF [6]) with a type former U : (i : ℕ) → Ty Γ (i+1) satisfying U i [σ]T = U i is Russell if the equations Ty Γ i = Tm Γ (U i) and A [σ]T = A [σ]t hold (where _[_]T and _[_]t are the substitution operations for types and terms, respectively).

Any CwF with a hierarchy of Tarski universes can be equipped with a Russell family structure supporting the same type formers as the Tarski universe. A hierarchy of Tarski universes is given by the universe types $\mathsf{U}\ \mathsf{i} : \mathsf{Ty}\ \Gamma\ (\mathsf{i}{+}1)$, their decoding $\mathsf{El} : \mathsf{Tm}\ \Gamma\ (\mathsf{U}\ \mathsf{i}) \to \mathsf{Ty}\ \Gamma\ \mathsf{i}$, a code for each universe $\mathsf{u}\ \mathsf{i} : \mathsf{Tm}\ \Gamma\ (\mathsf{U}\ (\mathsf{i}{+}1))$ such that their decoding is the actual universe $\mathsf{El}\ (\mathsf{u}\ \mathsf{i}) = \mathsf{U}\ \mathsf{i}$. We further have the evident substitution rules and additional operations expressing that U is closed under certain type formers. The Russell family structure then is defined by $\mathsf{Ty}^{\mathsf{R}}\ \Gamma\ \mathsf{i} := \mathsf{Tm}\ \Gamma\ (\mathsf{U}\ \mathsf{i})$ and $\mathsf{Tm}^{\mathsf{R}}\ \Gamma\ \mathsf{A} := \mathsf{Tm}\ \Gamma\ (\mathsf{El}\ \mathsf{A})$, both substitution operations are $\_[\_]\mathsf{t}$, context extension is $\Gamma \triangleright^{\mathsf{R}} \mathsf{A} := \Gamma \triangleright \mathsf{El}\ \mathsf{A}$. The Russell universe is defined as $\mathsf{U}^{\mathsf{R}}\ \mathsf{i} := \mathsf{u}\ \mathsf{i}$, thus we obtain the Russell sort equation by $\mathsf{Ty}^{\mathsf{R}}\ \Gamma\ \mathsf{i} = \mathsf{Tm}\ \Gamma\ (\mathsf{U}\ \mathsf{i}) = \mathsf{Tm}\ \Gamma\ (\mathsf{El}\ (\mathsf{u}\ \mathsf{i})) = \mathsf{Tm}^{\mathsf{R}}\ \Gamma\ (\mathsf{U}^{\mathsf{R}}\ \mathsf{i})$. We formalised this model construction using the shallow embedding trick of [14], the formalisation is part of the source code of the current paper[8].

6 Type Theory without Contexts

Having Russell universes could be called "type theory without types" as types are just special terms. Type theory without contexts is when contexts are just types without free variables.

When defining type theory as an algebraic theory, the final goal is to describe the rules for types and terms. Contexts and substitutions (the category structure) are only there as supporting infrastructure. However, when enough structure is added to types and terms, we don't need this supporting infrastructure anymore and we can get rid of it using the Münchhausen technique. We will still have explicit substitutions, but we use terms instead of context morphisms. The resulting theory with very dependent types includes the following sorts and operations. Note that some of these operations are not only very dependently typed, but the typing is very mutual: for example, the type of Ty includes $\top$ which is only listed later.

$$
\begin{aligned}
&\mathsf{Ty} &&: \mathsf{Ty}\ \top \to \mathsf{Set}\\
&\mathsf{Tm} &&: (\Gamma : \mathsf{Ty}\ \top) \to \mathsf{Ty}\ \Gamma \to \mathsf{Set}\\
&\_[\_]\mathsf{T} &&: \mathsf{Ty}\ \Gamma \to \mathsf{Tm}\ \Delta\ (\Gamma\ [\ \mathsf{tt}\ ]\mathsf{T}) \to \mathsf{Ty}\ \Delta\\
&\_[\_]\mathsf{t} &&: \mathsf{Tm}\ \Gamma\ \mathsf{A} \to (\sigma : \mathsf{Tm}\ \Delta\ (\Gamma\ [\ \mathsf{tt}\ ]\mathsf{T})) \to \mathsf{Tm}\ \Delta\ (\mathsf{A}\ [\ \sigma\ ]\mathsf{T})\\
&\mathsf{id} &&: \mathsf{Tm}\ \Gamma\ (\Gamma\ [\ \mathsf{tt}\ ]\mathsf{T})\\
&\top &&: \mathsf{Ty}\ \Gamma\\
&\mathsf{tt} &&: \mathsf{Tm}\ \Gamma\ \top\\
&\Sigma &&: (\mathsf{A} : \mathsf{Ty}\ \Gamma) \to \mathsf{Ty}\ (\Sigma\ \Gamma\ (\mathsf{A}\ [\ \mathsf{snd}\ \mathsf{id}\ ]\mathsf{T})) \to \mathsf{Ty}\ \Gamma\\
&\_,\_ &&: (\mathsf{a} : \mathsf{Tm}\ \Gamma\ \mathsf{A}) \to \mathsf{Tm}\ \Gamma\ (\mathsf{B}\ [\ \mathsf{id}\ ,\ \mathsf{a}\ ]\mathsf{T}) \to \mathsf{Tm}\ \Gamma\ (\Sigma\ \mathsf{A}\ \mathsf{B})\\
&\mathsf{fst} &&: \mathsf{Tm}\ \Gamma\ (\Sigma\ \mathsf{A}\ \mathsf{B}) \to \mathsf{Tm}\ \Gamma\ \mathsf{A}\\
&\mathsf{snd} &&: (\mathsf{w} : \mathsf{Tm}\ \Gamma\ (\Sigma\ \mathsf{A}\ \mathsf{B})) \to \mathsf{Tm}\ \Gamma\ (\mathsf{B}\ [\ \mathsf{id}\ ,\ \mathsf{fst}\ \mathsf{w}\ ]\mathsf{T})
\end{aligned}
$$

It is difficult to derive the above in Agda using forward declarations or rewrite rules, but working on paper (in extensional type theory) this is possible. A *model of type theory without contexts* is given by a CwF with $\top$ and Σ types[9] where the following equations hold.

[8] See `https://bitbucket.org/akaposi/combinator/src/master/post-types2022/russel.lagda`

[9] List of notations: the category is denoted Con, Sub, $\_\circ\_$, id, the empty context (terminal object) $\diamond$, the empty substitution ε, types are $\mathsf{Ty}\ \Gamma$ with the substitution operation $\_[\_]\mathsf{T}$, terms $\mathsf{Tm}\ \Gamma\ \mathsf{A}$ with $\_[\_]\mathsf{t}$, context extension $\_\triangleright\_$, substitution extension $\_,\_$ and projections $\mathsf{p} : \mathsf{Sub}\ (\Gamma \triangleright \mathsf{A})\ \Gamma$, $\mathsf{q} : \mathsf{Tm}\ (\Gamma \triangleright \mathsf{A})\ (\mathsf{A}\ [\ \mathsf{p}\ ]\mathsf{T})$. The type former $\top : \mathsf{Ty}\ \Gamma$ comes with constructor tt and η law. Σ's constructor is denoted $\_,\_$, the destructors are fst and snd and we have both β laws and an η law.

$$
\begin{aligned}
&\mathsf{Con} &&= \mathsf{Ty}\ \diamond \\
&\mathsf{Sub}\ \Delta\ \Gamma &&= \mathsf{Tm}\ \Delta\ (\Gamma\ [\ \varepsilon\ ]\mathsf{T}) \\
&\sigma \circ \nu &&= \sigma\ [\ \nu\ ]\mathsf{t} \\
&\diamond &&= \top \\
&\varepsilon &&= \mathsf{tt} \\
&\Gamma \triangleright \mathsf{A} &&= \Sigma\ \Gamma\ (\mathsf{A}\ [\ \mathsf{q}\ ]\mathsf{T}) \\
&\sigma\ ,\ \mathsf{t} &&= \sigma\ ,\ \mathsf{t} \\
&\mathsf{p} &&= \mathsf{fst\ id} \\
&\mathsf{q} &&= \mathsf{snd\ id}
\end{aligned}
$$

Note that the well-typedness of the second equation depends on the first equation, as $\Gamma : \mathsf{Con}$ has to be viewed as $\Gamma : \mathsf{Ty}\ \diamond$ and then we can substitute it with the empty substitution to obtain $\Gamma\ [\ \varepsilon\ ]\mathsf{T} : \mathsf{Ty}\ \Delta$. Just as substitutions are special terms, composition of substitutions is a special case of substitution of terms, the empty context is $\top : \mathsf{Ty}\ \diamond$, context extension is a Σ type where $\mathsf{A} : \mathsf{Ty}\ \Gamma$, but Σ requires a $\mathsf{Ty}\ (\diamond \triangleright \Gamma)$, so we need a $\mathsf{Sub}\ (\diamond \triangleright \Gamma)\ \Gamma = \mathsf{Tm}\ (\diamond \triangleright \Gamma)\ (\Gamma\ [\ \varepsilon\ ]\mathsf{T}) = \mathsf{Tm}\ (\diamond \triangleright \Gamma)\ (\Gamma\ [\ \mathsf{p}\ ]\mathsf{T})$, which is given by $\mathsf{q}\ \{\diamond\}\{\Gamma\}$.

We can check that in a model of type theory without contexts, the very dependent types listed above are all valid.

As for Russell models, we have a model construction which replaces any CwF with $\top$ and Σ with a model without contexts. We cannot directly use the equations of model without contexts above for the model construction. *E.g.* if we said that $\mathsf{Con}' := \mathsf{Ty}\ \diamond$ and $\diamond' := \top$ and $\mathsf{Ty}'\ \Gamma := \mathsf{Ty}\ (\diamond \triangleright \Gamma)$ then we would have $\mathsf{Con}' = \mathsf{Ty}\ \diamond \neq \mathsf{Ty}\ (\diamond \triangleright \top) = \mathsf{Ty}'\ \diamond'$. Instead we define $\mathsf{Con}' := \mathsf{Ty}\ (\diamond \triangleright \top)$, $\diamond' := \top$ and $\mathsf{Ty}'\ \Gamma := \mathsf{Ty}\ (\diamond \triangleright \Gamma\ [\ \varepsilon\ ,\ \mathsf{tt}\ ]\mathsf{T})$. Now we have $\mathsf{Con}' = \mathsf{Ty}\ (\diamond \triangleright \top) = \mathsf{Ty}\ (\diamond \triangleright \top\ [\ \varepsilon\ ,\ \mathsf{tt}\ ]\mathsf{T}) = \mathsf{Ty}\ (\diamond \triangleright \diamond'\ [\ \varepsilon\ ,\ \mathsf{tt}\ ]\mathsf{T}) = \mathsf{Ty}'\ \diamond'$. We refer to Appendix A for the definition of the rest of the components of the output model and also for a proof that if the input model has Π types then so does the output model. We formalised this model construction using shallow embedding [14], the formalisation is part of the source code of the current paper[10].

7 Combinatory Type Theory

In our final example, we also present a (dependnet) type theory without contexts. Instead of eliminating contexts with equations as we did in the previous section, we avoid introducing them in the encoding. This raises the question: if there are no contexts, how do we talk about well-scoped variables? As a matter of fact, we do not talk about variables at all.

It is well known that for simply-typed systems, combinator calculus [11] gives a contextless presentation of the type system. There are no variables, function space is built-in, and the combinators S and K are used to define functions.

Combinator calculus for dependently-typed systems is a much more challenging [17] task, and it was never defined. Unsurprisingly, contextless dependently-typed theory is an example of a very dependent type, and we use Münchhausen method to define it. Specifically, we use postulates and rewrite rules to encode very dependent types. While there might be a solution with forward declarations, we chose rewrite rules for the sake of simplicity of the presentation.

[10] See `https://bitbucket.org/akaposi/combinator/src/master/post-types2022/uncat.lagda`

First Attempt

In our first attempt we started defining a non-dependent function type, hoping to internalise it in the universe and define Π types afterwards. Concretely, we define types Ty, terms Tm indexed by types, the universe U, and eliminate the Ty type. After that, we define the arrow type _⇒_, applications, the arrow type within the universe |⇒| and the equation that turns the arrow into the internal arrow.

```
module FirstAttempt where
  postulate
    Ty  : Set                        - Types
    Tm  : Ty → Set                   - Terms
    U   : Ty                         - Universe
    TmU : Ty ≡ Tm U                  - Russell-ification
    {-# REWRITE TmU #-}
    _⇒_ : Ty → Ty → Ty               - Non-dependent (external) arrow type
    _$_ : Tm (X ⇒ Y) → Tm X → Tm Y - Applications
    |⇒| : Tm (U ⇒ U ⇒ U)             - Internal arrow type
    ∅⇒ : X ⇒ Y ≡ |⇒| $ X $ Y         - Internalising arrow
```

While this looks promising, after rewriting ∅⇒ we run into the following problem. Consider the sequence of rewrites that is happening for the type Tm $(X \Rightarrow Y)$ which is the type of the first argument of the application _$_.

```
_₁ : Tm (X ⇒ Y)                       - Expands to
_₁ : Tm (|⇒| $ X $ Y)                 - Show hidden arguments
_₂ : Tm ((_$_ {U}{U ⇒ U} |⇒| X) $ Y) - Arrow in (U ⇒ U) again!
```

As Agda applies all the rewrite rules before type checking, we end-up in the infinite rewrite loop. There does not seem to be an easy fix.

Second Attempt

Now we start with Π types straight away and use them to define dependent combinators. The notion of types, terms and the universe is the same as before.

```
module SecondAttempt where
  postulate
    Ty : Set
    Tm : Ty → Set
    U : Ty
    Tm-U : Tm U ≡ Ty
    {-# REWRITE Tm-U #-}
```

We introduce the notion of a U-valued family and the application operation for it. Using family we can immediately define Π types and applications to the terms of Π types.

```
    Fam  : Ty → Ty                   - Fam X ≈ (X ⇒ U)
    _$f_ : Tm (Fam X) → Tm X → Ty    - Apply (x : X) to (t : Fam X)

    Pi   : (X : Ty) → Tm (Fam (Fam X)) - X → ((X ⇒ U) ⇒ U)
    _$_  : {X : Ty}{Y : Tm (Fam X)}
         → Tm (Pi X $f Y) → (a : Tm X) → Tm (Y $f a)
```

Consider defining a non-dependent function type for $(X\ Y : \mathsf{Ty})$ using the Pi type. We can immediately apply X to Pi, but then we need to turn Y into a constant X-family in order to complete the definition (Pi X \$f □). To achieve this we introduce the Kf combinator that turns a type into a constant family. Its beta rule is the same as of the standard K combinator. Using Kf we can complete the definition of non-dependent arrow.

```
  Kf    : (Y : Ty) → Tm (Fam X) -- Kf Y ≈ λ a → Y
  Kf$   : ∀ {a : Tm X} → _$f_ {X} (Kf Y) a ≡ Y
  {-# REWRITE Kf$ #-}

_⇒_ : (X Y : Ty) → Ty
X ⇒ Y = Pi X $f (Kf Y)
```

Let us remind ourselves, the type of dependent K combinator:

```
[K] : (X : Set)(Y : X → Set) → (x : X) (y : Y x) → X
[K] X Y x y = x
```

Translation into our formalism requires expressing $(x : X)(y : Y\ x) \to X$ as a Pi type. More precisely, how do we express $(Y\ x \to X)$ as an X-family? We do this by introducing a helper combinator with the corresponding beta rule. After that, defining dependent K and its beta rule becomes straight-forward.

```
postulate -- Dependent K
  Yx⇒Z : ∀ X (Y : Tm (Fam X)) → (Z : Ty) → Tm (Fam X)
  Yx⇒Z$ : ∀ X Y Z {x : Tm X} → Yx⇒Z X Y Z $f x ≡ Y $f x ⇒ Z
  {-# REWRITE Yx⇒Z$ #-}

  Kd : {Y : Tm (Fam X)} → Tm (Pi X $f Yx⇒Z X Y X)
  Kd$ : ∀ {Y : Tm (Fam X)}{x : Tm X}{y : Tm (Y $f x)}
       → Kd {X = X}{Y = Y} $ x $ y ≡ x
  {-# REWRITE Kd$ #-}
```

Similarly to dependent K, we start with reminding ourselves the type of the dependent S combinator. We will use the same strategy of defining extra combinators to construct parts of the type signature.

```
[S] : (X : Set)(Y : X → Set)
       (Z : (x : X) → Y x → Set)          -- λ (x : X) → Yx⇒U x
    → (f : (x : X) → (y : Y x) → Z x y) -- λ (x : X) → Π[Yx][Zx] x
    → (g : (x : X) → Y x)
    → ((x : X) → Z x (g x))             -- λ (g : ΠXY) → ΠX[Zx[gx]] g
[S] X Y Z f g x = f x (g x)
```

We annotate the combinators we introduced at the corresponding positions of the [S] type. With these definitions, we can define dependent S and its beta rule as follows.

```
postulate -- Dependent S
  Yx⇒U : ∀ X (Y : Tm (Fam X)) → Tm (Fam X)
  Yx⇒U$ : ∀ X Y {x : Tm X} → Yx⇒U X Y $f x ≡ Fam (Y $f x)
  {-# REWRITE Yx⇒U$ #-}

  Π[Yx][Zx]  : ∀ X (Y : Tm (Fam X)) → (Z : Tm (Pi X $f Yx⇒U X Y)) → Tm (Fam X)
  Π[Yx][Zx]$ : ∀ X Y Z {x : Tm X} → Π[Yx][Zx] X Y Z $f x ≡ Pi (Y $f x) $f (Z $ x)
  {-# REWRITE Π[Yx][Zx]$ #-}

  Zx[gx] : ∀ X (Y : Tm (Fam X)) (Z : Tm (Pi X $f Yx⇒U X Y))
         → Tm (Pi X $f Y) → Tm (Fam X)
  Zx[gx]$ : ∀ X Y Z g {x} → Zx[gx] X Y Z g $f x ≡ Z $ x $f (g $ x)
  {-# REWRITE Zx[gx]$ #-}

  ΠX[Zx[gx]] : ∀ X (Y : Tm (Fam X)) (Z : Tm (Pi X $f Yx⇒U X Y))
             → Tm (Fam (Pi X $f Y))
  ΠX[Zx[gx]]$ : ∀ X Y Z g → ΠX[Zx[gx]] X Y Z $f g ≡ Pi X $f (Zx[gx] X Y Z g)
  {-# REWRITE ΠX[Zx[gx]]$ #-}

  Sd : {Y : Tm (Fam X)}{Z : Tm (Pi X $f Yx⇒U X Y)}
     → Tm (Pi X $f Π[Yx][Zx] X Y Z
              ⇒ (Pi (Pi X $f Y) $f (ΠX[Zx[gx]] X Y Z)))
  Sd$ : {Y : Tm (Fam X)}{Z : Tm (Pi X $f Yx⇒U X Y)}
      → {f : Tm (Pi X $f Π[Yx][Zx] X Y Z) }
      → {g : Tm (Pi X $f Y)}
      → {x : Tm X}
      → Sd $ f $ g $ x ≡ f $ x $ (g $ x)
  {-# REWRITE Sd$ #-}
```

Finally, with a few more rewrite rules, we can define non-dependent S and K combinators as special cases of their dependent versions.

```
K : Tm (X ⇒ Y ⇒ X)
K {X}{Y} = Kd {X}{Kf Y}

S : Tm ((X ⇒ Y ⇒ Z) ⇒ (X ⇒ Y) ⇒ X ⇒ Z)
S {X}{Y}{Z} = Sd {X}{Kf Y}{K $ (Kf Z)}
```

We made a good progress with defining combinatory type theory. However, current combinators are not yet powerful enough to internalise Pi and Fam. The problem is that in Pi, Kd and Sd type parameters X, Y and Z are quantified externally. We need to define the version of these combinators that internalises this quantification within U. There is no conceptual problem in doing so, but the resulting terms become incredibly large and inconvenient to work with. Specifically, the one for the dependent S combinator. It is not clear whether there is a more elegant way of doing this.

8 Conclusions

This paper demonstrates a technique to justify and make practical use of very dependent types. Our method is based on the observation that the "cycle" of a very dependent type can be "cut" by introducing placeholder types, defining the data and then eliminating placeholders by means of equations.

When we try to apply the proposed technique within the actual theorem provers such as Agda, we have a few choices on how to implement this. First, we can pack together placeholders, data and explicit equalities, *e.g.* as we do in DPair type in Section 3. This is a straight-forward implementation of the Münchhausen technique. However, dealing with explicit propositional equalities as parts of data often brings us to the situation called "transport hell". For example, the isomorphism proof about DPairs is an instance of that. Alternatively, for the objects of very dependent types, we can turn propositional equalities into definitional ones. On paper, extensional type theory achieves this, and in special cases we can use shallow embedding (as in the formalisation of Sections 5 and 6). In Agda, there are two ways to do this: forward declarations and rewrite rules. Forward declarations are demonstrated when declaring pair in Section 3, Ar universe in Section 4 and Tm in Section 5. While this is a very convenient feature of Agda, it is considered[11] not very well understood by many Agda developers. Also, as we have seen with Ar example, currently it leads to loops in the typechecker, which is clearly a bug.

Rewrite rules [7] make it possible to turn arbitrary propositional equalities into definitional ones, but this feature of Agda is considered unsafe. However, it is clearly a localized implementation of extensional type theory which is conservative over intensional type theory with extensionality principles (as available in Cubical Agda). We expect that the conservativity result [12] extends to our setting and hence the use of rewriting rules is only a cosmetic and labour saving tool to avoid *transport hell*. We use rewrite rules in Section 7. Currently, the interplay between the rewrite rules and the typechecker is not always satisfying. For example, our first attempt in Section 7 ends up in an infinite rewrite, as all the rules have to fire before the typechecker. We believe that more interleaved approach to rewriting could make our example to typecheck.

The examples show that very dependent types can be used in a fully algebraic setting, *i.e.* without referring to untyped preterms as in [10]. The essential ingredient are forward declarations, *i.e.* we introduce the type of an object but only define it later while already using it in the types of other objects – see [2] for a formal definition of this concept. This is also the idea in inductive-inductive definitions, where constructors may depend on previous constructors [4, 8].

Clearly, Agda provides us with a mechanism to play around with these concepts but it is not yet clear what exactly the theory behind these constructions is. In this sense, our paper raises questions instead of answering them. We believe that this is a valuable contribution to the subject.

References

1 Agda Development Team. *Agda 2.6.3 documentation*, 2023. Accessed [2023/05/01]. URL: `https://agda.readthedocs.io/en/v2.6.3/`.

2 Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh, and Nicolas Oury. ΠΣ: Dependent types without the sugar. *Functional and Logic Programming*, pages 40–55, 2010.

[11] See the following Agda issue `https://github.com/agda/agda/issues/1556` that discusses forward declarations and very dependent types.

3 Thorsten Altenkirch, Ambrus Kaposi, Artjoms Šinkarovs, and Tamás Végh. The Münchhausen method and combinatory type theory. In Delia Kesner and Pierre-Marie Pédrot, editors, *28th International Conference on Types for Proofs and Programs (TYPES 2022)*. University of Nantes, 2022. URL: `https://types22.inria.fr/files/2022/06/TYPES_2022_paper_8.pdf`.

4 Thorsten Altenkirch, Peter Morris, Fredrik Nordvall Forsberg, and Anton Setzer. A categorical semantics for inductive-inductive definitions. In *CALCO*, pages 70–84, 2011. `doi:10.1007/978-3-642-22944-2_6`.

5 Henk Barendregt. Introduction to generalized type systems. *J. Funct. Program.*, 1(2):125–154, 1991. `doi:10.1017/s0956796800020025`.

6 Simon Castellan, Pierre Clairambault, and Peter Dybjer. Categories with families: Unityped, simply typed, and dependently typed. *CoRR*, abs/1904.00827, 2019. `arXiv:1904.00827`.

7 Jesper Cockx. Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules. In Marc Bezem and Assia Mahboubi, editors, *25th International Conference on Types for Proofs and Programs (TYPES 2019)*, volume 175 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:27, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.TYPES.2019.2`.

8 Fredrik Nordvall Forsberg. *Inductive-inductive definitions.* PhD thesis, Swansea University (United Kingdom), 2013.

9 J. I. Glasgow and M. A. Jenkins. Array theory, logic and the nial language. In *Proceedings. 1988 International Conference on Computer Languages*, pages 296–303, October 1988. `doi:10.1109/ICCL.1988.13077`.

10 Jason J. Hickey. Formal objects in type theory using very dependent types. In *In Foundations of Object Oriented Languages 3*, 1996.

11 J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and Lambda-Calculus.* Cambridge University Press, 1986.

12 Martin Hofmann. Conservativity of equality reflection over intensional type theory. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs*, pages 153–164, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

13 Kenneth E. Iverson. *A Programming Language.* John Wiley & Sons, Inc., New York, NY, USA, 1962.

14 Ambrus Kaposi, András Kovács, and Nicolai Kraus. Shallow embedding of type theory is morally correct. In Graham Hutton, editor, *Mathematics of Program Construction*, pages 329–365, Cham, 2019. Springer International Publishing.

15 András Kovács. Generalized universe hierarchies and first-class universe levels. In Florin Manea and Alex Simpson, editors, *30th EACSL Annual Conference on Computer Science Logic, CSL 2022, February 14-19, 2022, Göttingen, Germany (Virtual Conference)*, volume 216 of *LIPIcs*, pages 28:1–28:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. `doi:10.4230/LIPIcs.CSL.2022.28`.

16 Per Martin-Löf. An intuitionistic theory of types. In Giovanni Sambin and Jan M. Smith, editors, *Twenty-five years of constructive type theory (Venice, 1995)*, volume 36 of *Oxford Logic Guides*, pages 127–172. Oxford University Press, 1998.

17 Conor McBride. What is the combinatory logic equivalent of intuitionistic type theory? Answer to question on StackOverflow, 2012. URL: `https://stackoverflow.com/questions/11406786/what-is-the-combinatory-logic-equivalent-of-intuitionistic-type-theory`.

18 Artjoms Šinkarovs. Multi-dimensional arrays with levels. In Max S. New and Sam Lindley, editors, *Proceedings Eighth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2020, Dublin, Ireland, 25th April 2020*, volume 317 of *EPTCS*, pages 57–71, 2020. `doi:10.4204/EPTCS.317.4`.

A The Type Theory without Contexts model construction

The input is a CwF with $\top$ and Σ types both having η rules. We use the same notation as in Section 6, the components of the model are Con, Ty, and so on. The components of the output model without contexts are denoted the same. We list all of them here in the following order: category, terminal object, types, terms, context extension, unit, Σ. This model construction was fully formalised in Agda.

$$
\begin{aligned}
&\mathsf{Con} &&:= \mathsf{Ty}\ (\diamond \triangleright \top)\\
&\mathsf{Sub}\ \Delta\ \Gamma &&:= \mathsf{Tm}\ (\diamond \triangleright \Delta\ [\ \varepsilon\ ,\ \mathsf{tt}\ ]\mathsf{T})\ (\Gamma\ [\ \varepsilon\ ,\ \mathsf{tt}\ ]\mathsf{T})\\
&\sigma \circ \nu &&:= \sigma\ [\ \varepsilon\ ,\ \nu\ ]\mathsf{t}\\
&\mathsf{id} &&:= \mathsf{q}\\
&\diamond &&:= \top\\
&\varepsilon &&:= \mathsf{tt}\\
&\mathsf{Ty}\ \Gamma &&:= \mathsf{Ty}\ (\diamond \triangleright \Gamma\ [\ \varepsilon\ ,\ \mathsf{tt}\ ]\mathsf{T})\\
&\mathsf{A}\ [\ \sigma\ ]\mathsf{T} &&:= \mathsf{A}\ [\ \varepsilon\ ,\ \sigma\ ]\mathsf{T}\\
&\mathsf{Tm}\ \Gamma\ \mathsf{A} &&:= \mathsf{Tm}\ (\diamond \triangleright \Gamma\ [\ \varepsilon\ ,\ \mathsf{tt}\ ]\mathsf{T})\ \mathsf{A}\\
&\mathsf{t}\ [\ \sigma\ ]\mathsf{t} &&:= \mathsf{t}\ [\ \varepsilon\ ,\ \sigma\ ]\mathsf{t}\\
&\Gamma \triangleright \mathsf{A} &&:= \Sigma\ \Gamma\ (\mathsf{A}\ [\ \varepsilon\ ,\ \mathsf{q}\ ]\mathsf{T})\\
&\sigma\ ,\ \mathsf{t} &&:= \sigma\ ,\ \mathsf{t}\\
&\mathsf{p} &&:= \mathsf{fst}\ \mathsf{q}\\
&\mathsf{q} &&:= \mathsf{snd}\ \mathsf{q}\\
&\top &&:= \top\\
&\mathsf{tt} &&:= \mathsf{tt}\\
&\Sigma\ \mathsf{A}\ \mathsf{B} &&:= \Sigma\ \mathsf{A}\ (\mathsf{B}[\ \varepsilon\ ,\ (\mathsf{q}\ [\ \mathsf{p}\ ]\mathsf{t}\ ,\ \mathsf{q})\ ])\mathsf{T}\\
&\mathsf{u}\ ,\ \mathsf{v} &&:= \mathsf{u}\ ,\ \mathsf{v}\\
&\mathsf{fst}\ \mathsf{t} &&:= \mathsf{fst}\ \mathsf{t}\\
&\mathsf{snd}\ \mathsf{t} &&:= \mathsf{snd}\ \mathsf{t}
\end{aligned}
$$

All the equations hold. If the input model has Π types, so does the output model. If the input model has a Coquand-universe, so does the output model. The operations are the following.

$$
\begin{aligned}
&\Pi\ \mathsf{A}\ \mathsf{B} &&:= \Pi\ \mathsf{A}\ (\mathsf{B}[\ \varepsilon\ ,\ (\mathsf{q}\ [\ \mathsf{p}\ ]\mathsf{t}\ ,\ \mathsf{q})\ ])\mathsf{T}\\
&\mathsf{lam}\ \mathsf{t} &&:= \mathsf{lam}\ (\mathsf{t}\ [\ \varepsilon\ ,\ (\mathsf{q}\ [\ \mathsf{p}\ ]\mathsf{t}\ ,\ \mathsf{q})\ ]\mathsf{t})\\
&\mathsf{app}\ \mathsf{t} &&:= (\mathsf{app}\ \mathsf{t})\ [\ \varepsilon\ ,\ \mathsf{fst}\ \mathsf{q}\ ,\ \mathsf{snd}\ \mathsf{q}\ ]\mathsf{t}\\
&\mathsf{U} &&:= \mathsf{U}\\
&\mathsf{El}\ \mathsf{t} &&:= \mathsf{El}\ \mathsf{t}\\
&\mathsf{c}\ \mathsf{A} &&:= \mathsf{c}\ \mathsf{A}
\end{aligned}
$$

All the equations hold.