System *F* in Agda, for fun and profit

James Chapman[1][0000−0001−9036−8252], Roman Kireev[1][0000−0003−4687−2739], Chad Nester[2], and Philip Wadler[2][0000−0001−7619−6378]

[1] IOHK, Hong Kong {james.chapman,roman.kireev}@iohk.io
[2] University of Edinburgh, UK {cnester,wadler}@inf.ed.ac.uk

Abstract. System F, also known as the polymorphic λ-calculus, is a typed λ-calculus independently discovered by the logician Jean-Yves Girard and the computer scientist John Reynolds. We consider $F_{\omega\mu}$, which adds higher-order kinds and iso-recursive types. We present the first complete, intrinsically typed, executable, formalisation of System $F_{\omega\mu}$ that we are aware of. The work is motivated by verifying the core language of a smart contract system based on System $F_{\omega\mu}$. The paper is a literate Agda script [14].

1 Introduction

System F, also known as the polymorphic λ-calculus, is a typed λ-calculus independently discovered by the logician Jean-Yves Girard and the computer scientist John Reynolds. System F extends the simply-typed λ-calculus (STLC). Under the principle of Propositions as Types, the $\rightarrow$ type of STLC corresponds to implication; to this System F adds a $\forall$ type that corresponds to universal quantification over propositions. Formalisation of System F is tricky: it, when extended with subtyping, formed the basis for the POPLmark challenge [8], a set of formalisation problems widely attempted as a basis for comparing different systems.

System F is small but powerful. By a standard technique known as Church encoding, it can represent a wide variety of datatypes, including natural numbers, lists, and trees. However, while System F can encode the type "list of A" for any type A that can also be encoded, it cannot encode "list" as a function from types to types. For that one requires System F with higher-kinded types, known as System F_ω. Girard's original work also considered this variant, though Reynolds did not.

The basic idea of System F_ω is simple. Not only does each *term* have a *type*, but also each *type* level object has a *kind*. Notably, type *families* are classified by higher kinds. The first level, relating terms and types, includes an embedding of STLC (plus quantification); while the second level, relating types and kinds, is an isomorphic image of STLC.

Church encodings can represent any algebraic datatype recursive only in positive positions; though extracting a component of a structure, such as finding the tail of a list, takes time proportional to the size of the structure. Another standard technique, known as Scott encoding, can represent any algebraic type whatsoever; and extracting a component now takes constant time. However, Scott encoding requires a second extension to System F, to represent arbitrary recursive types, known as System F_μ. The

system with both extensions is known as System $F_{\omega\mu}$, and will be the subject of our formalisation.

Terms in Systems F and F_ω are strongly normalising. Recursive types with recursion in a negative position permit encoding arbitrary recursive functions, so normalisation of terms in Systems F_μ and $F_{\omega\mu}$ may not terminate. However, constructs at the type level of Systems F_ω and $F_{\omega\mu}$ are also strongly normalising.

There are two approaches to recursive types, *equi-recursive* and *iso-recursive* [33]. In an equi-recursive formulation, the types $\mu\alpha.A[\alpha]$ and $A[\mu\alpha.A[\alpha]]$ are considered equal, while in an iso-recursive formulation they are considered isomorphic, with an *unfold* term to convert the former to the latter, and a *fold* term to convert the other way. Equi-recursive formulation makes coding easier, as it doesn't require extra term forms. But it makes type checking more difficult, and it is not known whether equi-recursive types for System $F_{\omega\mu}$ are decidable [19,11]. Accordingly, we use iso-recursive types, which are also used by Dreyer [18] and Brown and Palsberg [10].

There are also two approaches to formalising a typed calculus, *extrinsic* and *intrinsic* [35]. In an extrinsic formulation, terms come first and are assigned types later, while in an intrinsic formulation, types come first and a term can be formed only at a given type. The two approaches are sometimes associated with Curry and Church, respectively [23]. There is also the dichotomy between named variables and de Bruijn indices. De Bruijn indices ease formalisation, but require error-prone arithmetic to move a term underneath a lambda expression. An intrinsic formulation catches such errors, because they would lead to incorrect types. Accordingly, we use an intrinsic formulation with de Bruijn indices. The approach we follow was introduced by Altenkirch and Reus [6], and used by Chapman [13] and Allais et al. [2] among others.

1.1 For Fun and Profit

Our interest in System $F_{\omega\mu}$ is far from merely theoretical. Input Output HK Ltd. (IOHK) is developing the Cardano blockchain, which features a smart contract language known as Plutus [12]. The part of the contract that runs off-chain is written in Haskell with an appropriate library, while the part of the contract that runs on-chain is written using Template Haskell and compiled to a language called Plutus Core. Any change to the core language would require all participants of the blockchain to update their software, an event referred to as a *hard fork*. Hard forks are best avoided, so the goal with Plutus Core was to make it so simple that it is unlikely to need revision. The design settled on is System $F_{\omega\mu}$ with suitable primitives, using Scott encoding to represent data structures. Supported primitives include integers, bytestrings, and a few cryptographic and blockchain-specific operations.

The blockchain community puts a high premium on rigorous specification of smart contract languages. Simplicity, a proposed smart contract language for Bitcoin, has been formalised in Coq [31]. The smart contract language Michelson, used by Tezos, has also been formalised in Coq [30]. EVM, the virtual machine of Ethereum, has been formalised in K [32], in Isabelle/HOL [24,7], and in F^* [21]. For a more complete account of blockchain projects involving formal methods see [22].

IOHK funded the development of our formalisation of System $F_{\omega\mu}$ because of the correspondence to Plutus Core. The formal model in Agda and associated proofs give

us high assurance that our specification is correct. Further, we plan to use the evaluator that falls out from our proof of progress for testing against the evaluator for Plutus Core that is used in Cardano.

1.2 Contributions

This paper represents the first complete intrinsically typed, executable, formalisation of System $F_{\omega\mu}$ that we are aware of. There are other intrinsically typed formalisations of fragments of System $F_{\omega\mu}$. But, as far as we are aware none are complete. András Kovács has formalised System F_{ω}[27] using hereditary substitutions [38] at the type level. Kovács' formalisation does not cover iso-recursive types and also does not have the two different presentations of the syntax and the metatheory relating them that are present here.

Intrinsically typed formalisations of arguably more challenging languages exist such as those of Chapman [13] and Danielsson [16] for dependently typed languages. However, they are not complete and do not consider features such as recursive types. This paper represents a more complete treatment of a different point in the design space which is interesting in its own right and has computation at the type level but stops short of allowing dependent types. We believe that that techniques described here will be useful when scaling up to greater degrees of dependency.

A key challenge with the intrinsically typed approach for System F_{ω} is that due to computation at the type level, it is necessary to make use of the implementations of type level operations and even proofs of their correctness properties when defining the term level syntax and term level operations. Also, if we want to run term level programs, rather than just formalise them, it is vital that these proofs of type level operations compute, which means that we cannot assume any properties or rely on axioms in the metatheory such as functional extensionality. Achieving this level of completeness is a contribution of this paper as is the fact that this formalisation is executable. We do not need extensionality despite using higher order representations of renamings, substitutions, and (the semantics of) type functions. First order variants of these concepts are more cumbersome and long winded to work with. As the type level language is a strongly normalising extension of the simply-typed λ-calculus we were able to leverage work about renaming, substitution and normalisation from simply-typed λ-calculus. Albeit with the greater emphasis that proofs must compute. We learnt how to avoid using extensionality when reasoning about higher order/functional representations of renamings and substitutions from Conor McBride. The normalisation algorithm is derived from work by Allais et al. and McBride [3,29]. The normalisation proof also builds on their work, and in our opinion, simplifies and improves it as the uniformity property in the completeness proof becomes simply a type synonym required only at function type (kind in our case) rather than needing to be mutually defined with the logical relation at every type (kind), simplifying the construction and the proofs considerably. In addition we work with β-equality not $\beta\eta$-equality which, in the context of NBE makes things a little more challenging. The reason for this choice is that our smart contract core language Plutus Core has only β-equality.

Another challenge with the intrinsically typed approach for System F_{ω}, where typing derivations and syntax coincide, is that the presence of the conversion rule in the

syntax makes computation problematic as it can block β-reduction. Solving/avoiding this problem is a contribution of this paper.

The approach to the term level and the notation borrow heavily from PLFA [37] where the chapters on STLC form essentially a blueprint for and a very relevant introduction to this work. The idea of deriving an evaluator from a proof of progress appears in PLFA, and appears to be not widely known [36].

1.3 Overview

This paper is a literate Agda program that is machine checked and executable either via Agda's interpreter or compiled to Haskell. The code (i.e. the source code of the paper) is available as a supporting artefact. In addition the complete formalisation of the extended system (Plutus Core) on which this paper is based is also available as a supporting artefact.

In the paper we aim to show the highlights of the formalisation: we show as much code as possible and the statements of significant lemmas and theorems. We hide many proofs and minor auxiliary lemmas.

Dealing with the computation in types and the conversion rule was the main challenge in this work for us. The approaches taken to variable binding, renaming/substitution and normalisation lifted relatively easily to this setting. In addition to the two versions of syntax where types are (1) not normalised and (2) completely normalised we also experimented with a version where types are in weak head normal form (3). In (1) the conversion rule takes an inductive witness of type equality relation as an argument. In (2) conversion is derivable as type equality is replaced by identity. In (3), the type equality relation in conversion can be replaced by a witness of a logical relation that computes, indeed it is the same logical relation as described in the completeness of type normalisation proof. We did not pursue this further in this work so far as this approach is not used in Plutus Core but this is something that we would like to investigate further in future.

In section 2 we introduce intrinsically typed syntax (kinds, types and terms) and the dynamics of types (type equality). We also introduce the necessary syntactic operations for these definitions: type weakening and substitution (and their correctness properties) are necessary to define terms. In section 3 we introduce an alternative version of the syntax where the types are β-normal forms. We also introduce the type level normalisation algorithm, its correctness proof and a normalising substitution operation on normal types. In section 4 we reconcile the two versions of the syntax, prove soundness and completeness results and also demonstrate that normalising the types preserves the *semantics* of terms where semantics refers to corresponding untyped terms. In section 5 we introduce the dynamics of the algorithmic system (type preserving small-step reduction) and we prove progress in section 3. Preservation holds intrinsically. In section 6 we provide a step-indexed evaluator that we can use to execute programs for a given number of reduction steps. In section 7 we show examples of Church and Scott Numerals. In section 8 we discuss extensions of the formalisation to higher kinded recursive types and intrinsically sized integers and bytestrings.

2 Intrinsically typed syntax of System $F_{\omega\mu}$

We take the view that when writing a program such as an interpreter we want to specify very precisely how the program behaves on meaningful input and we want to rule out meaningless input as early and as conclusively as possible. Many of the operations we define in this paper, including substitution, evaluation, and normalisation, are only intended to work on well-typed input. In a programming language with a less precise type system we might need to work under the informal assumption that we will only ever feed meaningful inputs to our programs and otherwise their behaviour is unspecified, and all bets are off. Working in Agda we can guarantee that our programs will only accept meaningful input by narrowing the definition of valid input. This is the motivation for using intrinsic syntax as the meaningful inputs are those that are guaranteed to be type correct and in Agda we can build this property right into the definition of the syntax.

In practice, in our setting, before receiving the input (some source code in a file) it would have been run through a lexing, parsing, scope checking and most importantly *type checking* phase before reaching our starting point in this paper: intrinsically typed syntax. Formalising the type checker is future work.

One can say that in intrinsically typed syntax, terms carry their types. But, we can go further, the terms are actually typing derivations. Hence, the definition of the syntax and the type system, as we present it, coincide: each syntactic constructor corresponds to one typing rule and vice versa. As such we dispense with presenting them separately and instead present them in one go.

There are three levels in this syntax:

1. kinds, which classify types;
2. types, which classify terms;
3. terms, the level of ordinary programs.

The kind level is needed as there are functions at the type level. Types appear in terms, but terms do not appear in types.

2.1 Kinds

The kinds consist of a base kind `*`, which is the kind of types, and a function kind.[3]

```
data Kind : Set where
  *    : Kind                  -- type
  _⇒_ : Kind → Kind → Kind -- function kind
```

Let K and J range over kinds.

[3] The code in this paper is typeset in `colour`.

2.2 Type Contexts

To manage the types of variables and their scopes we introduce contexts. Our choice of how to deal with variables is already visible in the representation of contexts. We will use de Bruijn indices to represent variables. While this makes terms harder to write, it makes the syntactic properties of the language clear and any potential off-by-one errors etc. are mitigated by working with intrinsically scoped terms and the fact that syntactic properties are proven correct. We intend to use the language as a compilation target so ease of manually writing programs in this language is not a high priority.

We refer to type contexts as Ctx⋆ and reserve the name Ctx for term level contexts. Indeed, when a concept occurs at both type and term level we often suffix the name of the type level version with ⋆.

Type contexts are essentially lists of types written in reverse. No names are required.

```
data Ctx⋆ : Set where
  ∅   : Ctx⋆                  -- empty context
  _,⋆_ : Ctx⋆ → Kind → Ctx⋆ -- context extension
```

Let Φ and Ψ range over contexts.

2.3 Type Variables

We use de Bruijn indices for variables. They are natural numbers augmented with additional kind and context information. The kind index tells us the kind of the variable and the context index ensures that the variable is in scope. It is impossible to write a variable that isn't in the context. Z refers to the last variable introduced on the right hand end of the context. Adding one to a variable via S moves one position to the left in the context. Note that there is no way to construct a variable in the empty context as it would be out of scope. Indeed, there is no way at all to construct a variable that is out of scope.

```
data _∋⋆_ : Ctx⋆ → Kind → Set where
  Z : ∀ {Φ J}                → Φ ,⋆ J ∋⋆ J
  S : ∀ {Φ J K} → Φ ∋⋆ J → Φ ,⋆ K ∋⋆ J
```

Let α and β range over type variables.

2.4 Types

Types, like type variables, are indexed by context and kind, ensuring well-scopedness and well-kindedness. The first three constructors ‘ λ and $\cdot$ are analogous to the terms of STLC. This is extended with the Π type to classify type abstractions at the type level, function type $\Rightarrow$ to classify functions, and μ to classify recursive terms. Note that Π, $\Rightarrow$, and μ are effectively base types as they live at kind ⋆.

```
data _⊢⋆_ Φ : Kind → Set where
  ‘    : ∀{J}   → Φ ∋⋆ J                → Φ ⊢⋆ J      -- type variable
  ƛ    : ∀{K J} → Φ ,⋆ K ⊢⋆ J           → Φ ⊢⋆ K ⇒ J -- type lambda
```

```
_·_  : ∀{K J} → Φ ⊢⋆ K ⇒ J → Φ ⊢⋆ K → Φ ⊢⋆ J     -- type application
_⇒_  :          Φ ⊢⋆ ⋆     → Φ ⊢⋆ ⋆ → Φ ⊢⋆ ⋆     -- function type
Π    : ∀{K}   → Φ ,⋆ K ⊢⋆ ⋆           → Φ ⊢⋆ ⋆     -- Pi/forall type
μ    :          Φ ,⋆ ⋆ ⊢⋆ ⋆           → Φ ⊢⋆ ⋆     -- recursive type
```

Let A and B range over types.

2.5 Type Renaming

Types can contain functions and as such are subject to a nontrivial equality relation. To explain the computation equation (the β-rule) we need to define substitution for a single type variable in a type. Later, when we define terms that are indexed by their type we will need to be able to weaken types by an extra kind (section 2.9) and also, again, substitute for a single type variable in a type (section 2.10). There are various different ways to define the required weakening and substitution operations. We choose to define so-called parallel renaming and substitution i.e. renaming/substitution of several variables at once. Weakening and single variable substitution are special cases of these operations.

We follow Altenkirch and Reus [6] and implement renaming first and then substitution using renaming. In our opinion the biggest advantage of this approach is that it has a very clear mathematical theory. The necessary correctness properties of renaming are identified with the notion of a functor and the correctness properties of substitution are identified with the notion of a relative monad. For the purposes of reading this paper it is not necessary to understand relative monads in detail. The important thing is that, like ordinary monads, they have a return and bind and the rules that govern them are the same. It is only the types of the operations involved that are different. The interested reader may consult [5] for a detailed investigation of relative monads and [4] for a directly applicable investigation of substitution of STLC as a relative monad.

A type renaming is a function from type variables in one context to type variables in another. This is much more flexibility than we need. We only need the ability to introduce new variable on the right hand side of the context. The simplicity of the definition makes it easy to work with and we get some properties for free that we would have to pay for with a first order representation, such as not needing to define a lookup function, and we inherit the properties of functions provided by η-equality, such as associativity of composition, for free. Note that even though renamings are functions we do not require our metatheory (Agda's type system) to support functional extensionality. As pointed out to us by Conor McBride we only ever need to make use of an equation between renamings on a point (a variable) and therefore need only a pointwise version of equality on functions to work with equality of renamings and substitutions.

```
Ren⋆ : Ctx⋆ → Ctx⋆ → Set
Ren⋆ Φ Ψ = ∀ {J} → Φ ∋⋆ J → Ψ ∋⋆ J
```

Let ρ range over type renamings.

As we are going to push renamings through types we need to be able to push them under a binder. To do this safely the newly bound variable should remain untouched and

other renamings should be shifted by one to accommodate this. This is exactly what the lift⋆ function does and it is defined by recursion on variables:

```
lift⋆ : ∀ {Φ Ψ} → Ren⋆ Φ Ψ → ∀ {K} → Ren⋆ (Φ ,⋆ K) (Ψ ,⋆ K)
lift⋆ ρ Z     = Z       -- leave newly bound variable untouched
lift⋆ ρ (S α) = S (ρ α) -- apply renaming to other variables and add 1
```

Next we define the action of renaming on types. This is defined by recursion on the type. Observe that we lift the renaming when we go under a binder and actually apply the renaming when we hit a variable:

```
ren⋆ : ∀ {Φ Ψ} → Ren⋆ Φ Ψ → ∀ {J} → Φ ⊢⋆ J → Ψ ⊢⋆ J
ren⋆ ρ (` α)       = ` (ρ α)
ren⋆ ρ (ƛ B)       = ƛ (ren⋆ (lift⋆ ρ) B)
ren⋆ ρ (A · B)     = ren⋆ ρ A · ren⋆ ρ B
ren⋆ ρ (A ⇒ B)     = ren⋆ ρ A ⇒ ren⋆ ρ B
ren⋆ ρ (Π B)       = Π (ren⋆ (lift⋆ ρ) B)
ren⋆ ρ (μ B)       = μ (ren⋆ (lift⋆ ρ) B)
```

Weakening is a special case of renaming. We apply the renaming S which does double duty as the variable constructor, if we check the type of S we see that it is a renaming.

Weakening shifts all the existing variables one place to the left in the context:

```
weaken⋆ : ∀ {Φ J K} → Φ ⊢⋆ J → Φ ,⋆ K ⊢⋆ J
weaken⋆ = ren⋆ S
```

2.6 Type Substitution

Having defined renaming we are now ready to define substitution for types. Substitutions are defined as functions from type variables to types:

```
Sub⋆ : Ctx⋆ → Ctx⋆ → Set
Sub⋆ Φ Ψ = ∀ {J} → Φ ∋⋆ J → Ψ ⊢⋆ J
```

Let σ range over substitutions.

We must be able to lift substitutions when we push them under binders. Notice that we leave the newly bound variable intact and make use of weaken⋆ to weaken a term that is substituted.

```
lifts⋆ : ∀ {Φ Ψ} → Sub⋆ Φ Ψ → ∀ {K} → Sub⋆ (Φ ,⋆ K) (Ψ ,⋆ K)
lifts⋆ σ Z     = ` Z -- leave newly bound variable untouched
lifts⋆ σ (S α) = weaken⋆ (σ α) -- apply substitution and weaken
```

Analogously to renaming, we define the action of substitutions on types:

```
sub⋆ : ∀ {Φ Ψ} → Sub⋆ Φ Ψ → ∀ {J} → Φ ⊢⋆ J → Ψ ⊢⋆ J
sub⋆ σ (` α)       = σ α
```

```
sub⋆ σ (ƛ B)     = ƛ (sub⋆ (lifts⋆ σ) B)
sub⋆ σ (A · B)   = sub⋆ σ A · sub⋆ σ B
sub⋆ σ (A ⇒ B)   = sub⋆ σ A ⇒ sub⋆ σ B
sub⋆ σ (Π B)     = Π (sub⋆ (lifts⋆ σ) B)
sub⋆ σ (μ B)     = μ (sub⋆ (lifts⋆ σ) B)
```

Substitutions could be implemented as lists of types and then the *cons* constructor would extend a substitution by an additional term. Using our functional representation for substitutions it is convenient to define an operation for this. This effectively defines a new function that, if it is applied to the Z variable, returns our additional terms and otherwise invokes the original substitution.

```
extend⋆ : ∀{Φ Ψ} → Sub⋆ Φ Ψ → ∀{J}(A : Ψ ⊢⋆ J) → Sub⋆ (Φ ,⋆ J) Ψ
extend⋆ σ A Z     = A   -- project out additional term
extend⋆ σ A (S α) = σ α -- apply original substitution
```

Substitution of a single type variable is a special case of parallel substitution sub⋆. Note we make use of extend⋆ to define the appropriate substitution by extending the substitution ‘ with the type A. Notice that the variable constructor ‘ serves double duty as the identity substitution:

```
_[_]⋆ : ∀ {Φ J K} → Φ ,⋆ K ⊢⋆ J → Φ ⊢⋆ K → Φ ⊢⋆ J
B [ A ]⋆ = sub⋆ (extend⋆ ‘ A) B
```

At this point the reader may well ask how we know that our substitution and renaming operations are the right ones. One indication that we have the right definitions is that renaming defines a functor, and that substitution forms a relative monad. Further, evaluation (eval defined in section 3.2) can be seen as an algebra of this relative monad. This categorical structure results in clean proofs.

Additionally, without some sort of compositional structure to our renaming and substitution, we would be unable to define coherent type level operations. For example, we must have that performing two substitutions in sequence results in the same type as performing the composite of the two substitutions. We assert that these are necessary functional correctness properties and structure our proofs accordingly.

Back in our development we show that lifting a renaming and the action of renaming satisfy the functor laws where lift⋆ and ren⋆ are both functorial actions.

```
lift⋆-id      : ∀ {Φ J K}(α : Φ ,⋆ K ∋⋆ J) → lift⋆ id α ≡ α
lift⋆-comp : ∀{Φ Ψ Θ}{ρ : Ren⋆ Φ Ψ}{ρ' : Ren⋆ Ψ Θ}{J K}(α : Φ ,⋆ K ∋⋆ J)
  → lift⋆ (ρ' ∘ ρ) α ≡ lift⋆ ρ' (lift⋆ ρ α)

ren⋆-id      : ∀{Φ J}(A : Φ ⊢⋆ J) → ren⋆ id A ≡ A
ren⋆-comp : ∀{Φ Ψ Θ}{ρ : Ren⋆ Φ Ψ}{ρ' : Ren⋆ Ψ Θ}{J}(A : Φ ⊢⋆ J)
  → ren⋆ (ρ' ∘ ρ) A ≡ ren⋆ ρ' (ren⋆ ρ A)
```

Lifting a substitution satisfies the functor laws where lift⋆ is a functorial action:

```
lifts⋆-id   : ∀ {Φ J K}(x : Φ ,⋆ K ∋⋆ J) → lifts⋆ ‘ x ≡ ‘ x
lifts⋆-comp : ∀{Φ Ψ Θ}{σ : Sub⋆ Φ Ψ}{σ’ : Sub⋆ Ψ Θ}{J K}(α : Φ ,⋆ K ∋⋆ J)
  → lifts⋆ (sub⋆ σ’ ∘ σ) α ≡ sub⋆ (lifts⋆ σ’) (lifts⋆ σ α)
```

The action of substitution satisfies the relative monad laws where ‘ is return and sub⋆ is bind:

```
sub⋆-id   : ∀ {Φ J}(A : Φ ⊢⋆ J) → sub⋆ ‘ A ≡ A
sub⋆-var  : ∀ {Φ Ψ}{σ : Sub⋆ Φ Ψ}{J}(α : Φ ∋⋆ J) → sub⋆ σ (‘ α) ≡ σ α
sub⋆-comp : ∀{Φ Ψ Θ}{σ : Sub⋆ Φ Ψ}{σ’ : Sub⋆ Ψ Θ}{J}(A : Φ ⊢⋆ J)
  → sub⋆ (sub⋆ σ’ ∘ σ) A ≡ sub⋆ σ’ (sub⋆ σ A)
```

Note that the second law holds definitionally, it is the first line of the definition of sub⋆.

2.7 Type Equality

We define type equality as an intrinsically scoped and kinded relation. In particular, this means it is impossible to state an equation between types in different contexts, or of different kinds. The only interesting rule is the β-rule from the lambda calculus. We omit the η-rule as Plutus Core does not have it. The formalisation could be easily modified to include it and it would slightly simplify the type normalisation proof. The additional types ($\Rightarrow$, $\forall$, and μ) do not have any computational behaviour, and are essentially inert. In particular, the fixed point operator μ does not complicate the equational theory.

```
data _≡β_ {Φ} : ∀{J} → Φ ⊢⋆ J → Φ ⊢⋆ J → Set where
  β≡β : ∀{K J}(B : Φ ,⋆ J ⊢⋆ K)(A : Φ ⊢⋆ J) → ƛ B · A ≡β B [ A ]⋆
  -- remaining rules hidden
```

We omit the rules for reflexivity, symmetry, transitivity, and congruence rules for type constructors.

2.8 Term contexts

Having dealt with the type level, we turn our attention to the term level.

Terms may contain types, and so the term level contexts must also track information about type variables in addition to term variables. We would like to avoid having the extra syntactic baggage of multiple contexts. We do so by defining term contexts which contain both (the kinds of) type variables and (the types of) term variables. Term contexts are indexed over type contexts. In an earlier version of this formalisation instead of indexing by type contexts we defined inductive term contexts simultaneously with a recursive erasure operation that converts a term level context to a type level context by dropping the term variables but keeping the type variables. Defining an inductive data type simultaneously with a recursive function is referred to as *induction recursion* [20]. This proved to be too cumbersome in later proofs as it can introduce a situation where there can be multiple provably equal ways to recover the same type context and expressions become cluttered with proofs of such equations. In addition to the difficulty of

working with this version, it also made type checking the examples in our formalisation much slower. In the version presented here neither of these problems arise.
A context is either empty, or it extends an existing context by a type variable of a given kind, or by a term variable of a given type.

```
data Ctx : Ctx⋆ → Set where
  ∅  : Ctx ∅
  -- empty term context
  _,⋆_ : ∀{Φ} → Ctx Φ → ∀ J → Ctx (Φ ,⋆ J)
  -- extension by (the kind of) a type variable
  _,_ : ∀ {Φ} → Ctx Φ → Φ ⊢⋆ * → Ctx Φ
  -- extension by (the type of) a term variable
```

Let Γ, Δ, range over contexts. Note that in the last rule _,_, the type we are extending by may only refer to variables in the type context, a term that inhabits that type may refer to any variable in its context.

2.9 Term variables

A variable is indexed by its context and type. While type variables can appear in types, and those types can appear in terms, the variables defined here are term level variables only.

Notice that there is only one base constructor Z. This gives us exactly what we want: we can only construct term variables. We have two ways to shift these variables to the left, we use S to shift over a type and T to shift over a kind in the context.

```
data _∋_ : ∀{Φ} → Ctx Φ → Φ ⊢⋆ * → Set where
  Z : ∀{Φ Γ} {A : Φ ⊢⋆ *}                    → Γ , A ∋ A
  S : ∀{Φ Γ} {A : Φ ⊢⋆ *} {B : Φ ⊢⋆ *} → Γ ∋ A → Γ , B ∋ A
  T : ∀{Φ Γ} {A : Φ ⊢⋆ *} {K}           → Γ ∋ A → Γ ,⋆ K ∋ weaken⋆ A
```

Let x, y range over variables. Notice that we need weakening of (System F) types in the (Agda) type of T. We must weaken A to shift it from context Γ to context Γ ,⋆ K. Indeed, weaken⋆ is a function and it appears in a type. This is possible due to the rich support for dependent types and in particular inductive families in Agda. It is however a feature that must be used with care and while it often seems to be the most natural option it can be more trouble than it is worth. We have learnt from experience, for example, that it is easier to work with renamings (morphisms between contexts) ρ : Ren Γ Δ rather than context extensions Γ + Δ where the contexts are built from concatenation. The function +, whose associativity holds only propositionally, is awkward to work with when it appears in type indices. Renamings do not suffer from this problem as no additional operations on contexts are needed as we commonly refer to a renaming into an *arbitrary* new context (e.g., Δ) rather than, precisely, an extension of an existing one (e.g., Γ + Δ). In this formalisation we could have chosen to work with explicit renamings and substitutions turning operations like weaken⋆ into more benign constructors but this would have been overall more cumbersome and in this case we are able to work with executable renaming and substitution cleanly. Doing so cleanly is a contribution of this work.

2.10 Terms

A term is indexed by its context and type. A term is a variable, an abstraction, an application, a type abstraction, a type application, a wrapped term, an unwrapped term, or a term whose type is cast to another equal type.

```
data _⊢_ {Φ} Γ : Φ ⊢⋆ * → Set where
  `       : ∀{A}     → Γ ∋ A                          → Γ ⊢ A          -- variable
  ƛ       : ∀{A B}   → Γ , A ⊢ B                      → Γ ⊢ A ⇒ B      -- term λ
  _·_     : ∀{A B}   → Γ ⊢ A ⇒ B                      → Γ ⊢ A → Γ ⊢ B  -- term app
  Λ       : ∀{K B}   → Γ ,⋆ K ⊢ B                     → Γ ⊢ Π B        -- type λ
  _·⋆_    : ∀{K B}   → Γ ⊢ Π B → (A : Φ ⊢⋆ K)         → Γ ⊢ B [ A ]⋆   -- type app
  wrap    : ∀ A      → Γ ⊢ A [ μ A ]⋆                 → Γ ⊢ μ A         -- wrap
  unwrap  : ∀{A}     → Γ ⊢ μ A                        → Γ ⊢ A [ μ A ]⋆ -- unwrap
  conv    : ∀{A B}   → A ≡β B  → Γ ⊢ A                → Γ ⊢ B          -- type cast
```

Let L, M range over terms. The last rule conv is required as we have computation in types. So, a type which has a β-redex in it is equal, via type equality, to the type where that redex is reduced. We want a term which is typed by the original unreduced type to also be typed by the reduced type. This is a standard typing rule but it looks strange as a syntactic constructor. See [17] for a discussion of syntax with explicit conversions.

We could give a dynamics for this syntax as a small-step reduction relation but the conv case is problematic. It is not enough to say that a conversion reduces if the underlying term reduces. If a conversion is in the function position (also called head position) in an application it would block β-reduction. We cannot prove progress directly for such a relation. One could try to construct a dynamics for this system where during reduction both terms and also types can make reduction steps and we could modify progress and explicitly prove preservation. We do not pursue this here. In the system we present here we have the advantage that the type level language is strongly normalising. In section 3 we are able to make use of this advantage quite directly to solve the conversion problem in a different way. An additional motivation for us to choose the normalisation oriented approach is that in Plutus, contracts are stored and executed on chain with types normalised and this mode of operation is therefore needed anyway.

If we forget intrinsically typed syntax for a moment and consider these rules as a type system then we observe that it is not syntax directed, we cannot use it as the algorithmic specification of a type checker as we can apply the conversion rule at any point. This is why we refer to this version of the rules as *declarative* and the version presented in section 3, which is (in this specific sense) syntax directed, as *algorithmic*.

3 Algorithmic Rules

In this section we remove the conversion rule from our system. Two promising approaches to achieving this are (1) to push traces of the conversion rule into the other rules which is difficult to prove complete [34] and (2) to normalise the types which collapses all the conversion proofs to reflexivity. In this paper we will pursue the latter.

In the pursuit of (2) we have another important design decision to make: which approach to take to normalisation. Indeed, another additional aspect to this is that we need not only a normaliser but a normal form respecting substitution operation. We choose to implement a Normalisation-by-Evaluation (NBE) style normaliser and use that to implement a substitution operation on normal forms.

We chose NBE as we are experienced with it and it has a clear mathematical structure (e.g., evaluation is a relative algebra for the relative monad given by substitution) which gave us confidence that we could construct a well structured normalisation proof that would compute. The NBE approach is also centred around a normalisation *algorithm*: something that we want to use. Other approaches would also work we expect. One option would be to try hereditary substitutions where the substitution operation is primary and use that to define a normaliser.

Section 3.1–section 3.6 describe the normal types, the normalisation algorithm, its correctness proof, and a normalising substitution operation. Readers not interested in these details may skip to section 3.7.

3.1 Normal types

We define a data type of β-normal types which are either in constructor form or neutral. Neutral types, which are defined mutually with normal types, are either variables or (possibly nested) applications that are stuck on a variable in a function position, so cannot reduce. In this syntax, it is impossible to define an expression containing a β-redex.

```
data _⊢Nf⋆_ Φ : Kind → Set

data _⊢Ne⋆_ Φ J : Set where
  `     :          Φ ∋⋆ J                        → Φ ⊢Ne⋆ J -- type var
  _·_   : ∀{K} → Φ ⊢Ne⋆ (K ⇒ J) → Φ ⊢Nf⋆ K → Φ ⊢Ne⋆ J -- neutral app

data _⊢Nf⋆_ Φ where
  ƛ     : ∀{K J} → Φ ,⋆ K ⊢Nf⋆ J          → Φ ⊢Nf⋆ (K ⇒ J) -- type lambda
  ne    : ∀{K}   → Φ ⊢Ne⋆ K               → Φ ⊢Nf⋆ K       -- neutral type
  _⇒_   :            Φ ⊢Nf⋆ ∗ → Φ ⊢Nf⋆ ∗ → Φ ⊢Nf⋆ ∗       -- function type
  Π     : ∀{K}   → Φ ,⋆ K ⊢Nf⋆ ∗          → Φ ⊢Nf⋆ ∗       -- pi/forall type
  μ     :          Φ ,⋆ ∗ ⊢Nf⋆ ∗          → Φ ⊢Nf⋆ ∗       -- recursive type
```

Let A, B range over neutral and normal types.

As before, we need weakening at the type level in the definition of term level variables. As before, we define it as a special case of renaming whose correctness we verify by proving the functor laws.

```
renNf⋆      : ∀{Φ Ψ} → Ren⋆ Φ Ψ → ∀ {J} → Φ ⊢Nf⋆ J → Ψ ⊢Nf⋆ J
renNe⋆      : ∀{Φ Ψ} → Ren⋆ Φ Ψ → ∀ {J} → Φ ⊢Ne⋆ J → Ψ ⊢Ne⋆ J
weakenNf⋆ : ∀{Φ J K} → Φ ⊢Nf⋆ J → Φ ,⋆ K ⊢Nf⋆ J
```

Renaming of normal and neutral types satisfies the functor laws where renNf⋆ and renNe⋆ are both functorial actions:

```
renNf⋆-id   : ∀{Φ J}(A : Φ ⊢Nf⋆ J) → renNf⋆ id A ≡ A
renNf⋆-comp : ∀{Φ Ψ Θ}{ρ : Ren⋆ Φ Ψ}{ρ' : Ren⋆ Ψ Θ}{J}(A : Φ ⊢Nf⋆ J)
  → renNf⋆ (ρ' ∘ ρ) A ≡ renNf⋆ ρ' (renNf⋆ ρ A)

renNe⋆-id   : ∀{Φ J}(A : Φ ⊢Ne⋆ J) → renNe⋆ id A ≡ A
renNe⋆-comp : ∀{Φ Ψ Θ}{ρ : Ren⋆ Φ Ψ}{ρ' : Ren⋆ Ψ Θ}{J}(A : Φ ⊢Ne⋆ J)
  → renNe⋆ (ρ' ∘ ρ) A ≡ renNe⋆ ρ' (renNe⋆ ρ A)
```

3.2 Type Normalisation algorithm

We use the NBE approach introduced by [9]. This is a two stage process, first we evaluate into a semantic domain that supports open terms, then we reify these semantic terms back into normal forms.

The semantic domain $\models$, our notion of semantic value is defined below. Like syntactic types and normal types it is indexed by context and kind. However, it is not a type defined as an inductive data type. Instead, it is function that returns a type. More precisely, it is a function that takes a context and, by recursion on kinds, defines a new type. At base kind it is defined to be the type of normal types. At function kind it is either a neutral type at function kind or a semantic function. If it is a semantic function then we are essentially interpreting object level (type) functions as meta level (Agda) functions. The additional renaming argument means we have a so-called *Kripke function space* ([25]). This is essential for our purposes as it allows us to introduce new free variables into the context and then apply functions to them. Without this feature we would not be able to reify from semantic values to normal forms.

```
_⊨_ : Ctx⋆ → Kind → Set
Φ ⊨ ⋆         = Φ ⊢Nf⋆ ⋆
Φ ⊨ (K ⇒ J) = Φ ⊢Ne⋆ (K ⇒ J) ⊎ ∀ {Ψ} → Ren⋆ Φ Ψ → Ψ ⊨ K → Ψ ⊨ J
```

Let V, W range over values. Let F, G range over meta-level (Agda) functions. The definition $\models$ is a Kripke Logical Predicate. It is also a so-called large elimination, as it is a function which returns a new type (a Set in Agda terminology). This definition is inspired by Allais et al. [3]. Their normalisation proof, which we also took inspiration from, is, in turn, based on the work of C. Coquand [15]. The coproduct at the function kind is present in McBride [29]. Our motivation for following these three approaches was to be careful not to perturb neutral terms where possible as we want to use our normaliser in substitution and we want the identity substitution for example not to modify variables. We also learned from [3] how to move the uniformity condition out of the definition of values into the completeness relation.

We will define an evaluator to interpret syntactic types into this semantic domain but first we need to explain how to reify from semantics to normal forms. This is needed first as, at base type, our semantic values are normal forms, so we need a way to convert from values to normal forms during evaluation. Note that usual NBE operations of reify and reflect are not mutually defined here as they commonly are in $\beta\eta$-NBE. This is a characteristic of the coproduct style definition above.

Reflection takes a neutral type and embeds it into a semantic type. How we do this depends on what kind we are at. At base kind *, semantic values are normal forms, so we embed our neutral term using the ne constructor. At function kind, semantic values are a coproduct of either a neutral term or a function, so we embed our neutral term using the inl constructor.

```
reflect : ∀{K Φ} → Φ ⊢Ne⋆ K → Φ ⊨ K
reflect {*}       A = ne A
reflect {K ⇒ J} A = inl A
```

Reification is the process of converting from a semantic type to a normal syntactic type. At base kind and for neutral functions it is trivial, either we already have a normal form or we have a neutral term which can be embedded. The last line, where we have a semantic function is where the action happens. We create a fresh variable of kind K using reflect and apply f to it making use of the Kripke function space by supplying f with the weakening renaming S. This creates a semantic value of kind J in context Φ , K which we can call reify recursively on. This, in turn, gives us a normal form in Φ , K ⊢Nf⋆ J. We can then wrap this normal form in a λ.

```
reify : ∀ {K Φ} → Φ ⊨ K → Φ ⊢Nf⋆ K
reify {*}       A       = A
reify {K ⇒ J} (inl A) = ne A
reify {K ⇒ J} (inr F) = λ (reify (F S (reflect (‘ Z))))
```

We define renaming for semantic values. In the semantic function case, the new renaming is composed with the existing one.

```
ren⊨ : ∀ {σ Φ Ψ} → Ren⋆ Φ Ψ → Φ ⊨ σ → Ψ ⊨ σ
ren⊨ {*}       ρ A       = renNf⋆ ρ A
ren⊨ {K ⇒ J} ρ (inl A) = inl (renNe⋆ ρ A)
ren⊨ {K ⇒ J} ρ (inr F) = inr (λ ρ’ → F (ρ’ ∘ ρ))
```

Weakening for semantic values is a special case of renaming:

```
weaken⊨ : ∀ {σ Φ K} → Φ ⊨ σ → (Φ ,⋆ K) ⊨ σ
weaken⊨ = ren⊨ S
```

Our evaluator will take an environment giving semantic values to syntactic variables, which we represent as a function from variables to values:

```
Env : Ctx⋆ → Ctx⋆ → Set
Env Ψ Φ = ∀{J} → Ψ ∋⋆ J → Φ ⊨ J
```

Let η, η’ range over environments.

It is convenient to extend an environment with an additional semantic type:

```
extende : ∀{Ψ Φ} → (η : Env Φ Ψ) → ∀{K}(A : Ψ ⊨ K) → Env (Φ ,⋆ K) Ψ
extende η V Z     = V
extende η V (S α) = η α
```

Lifting of environments to push them under binders can be defined as follows. One could also define it analogously to the lifting of renamings and substitutions defined in section 2.

```
lifte : ∀ {Φ Ψ} → Env Φ Ψ → ∀ {K} → Env (Φ ,⋆ K) (Ψ ,⋆ K)
lifte η = extende (weaken⊨ ∘ η) (reflect (‘ Z))
```

We define a semantic version of application called ·V which applies semantic functions to semantic arguments. As semantic values at function kind can either be neutral terms or genuine semantic functions we need to pattern match on them to see how to apply them. Notice that the identity renaming id is used in the case of a semantic function. This is because, as we can read of from the type of $\cdot V$, the function and the argument are in the same context.

```
_·V_ : ∀{Φ K J} → Φ ⊨ (K ⇒ J) → Φ ⊨ K → Φ ⊨ J
inl A ·V V = reflect (A · reify V)
inr F ·V V = F id V
```

Evaluation is defined by recursion on types:

```
eval : ∀{Φ Ψ K} → Ψ ⊢⋆ K → Env Ψ Φ → Φ ⊨ K
eval (‘ α)       η = η α
eval (ƛ B)       η = inr λ ρ v → eval B (extende (ren⊨ ρ ∘ η) v)
eval (A · B)     η = eval A η ·V eval B η
eval (A ⇒ B) η = reify (eval A η) ⇒ reify (eval B η)
eval (Π B)       η = Π (reify (eval B (lifte η)))
eval (μ B)       η = μ (reify (eval B (lifte η)))
```

We can define the identity environment as a function that embeds variables into neutral terms with ‘ and then reflects them into values:

```
idEnv : ∀ Φ → Env Φ Φ
idEnv Φ = reflect ∘ ‘
```

We combine reify with eval in the identity environment idEnv to yield a normalisation function that takes types in a given context and kind and returns normal forms in the same context and kind:

```
nf : ∀{Φ K} → Φ ⊢⋆ K → Φ ⊢Nf⋆ K
nf A = reify (eval A (idEnv _))
```

In the next three sections we prove the three correctness properties about this normalisation algorithm: completeness; soundness; and stability.

3.3 Completeness of Type Normalisation

Completeness states that normalising two β-equal types yields the same normal form. This is an important correctness property for normalisation: it ensures that normalisation picks out unique representatives for normal forms. In a similar way to how we

defined the semantic domain by recursion on kinds, we define a Kripke Logical Relation on kinds which is a sort of equality on values. At different kinds and for different semantic values it means different things: at base type and for neutral functions it means equality of normal forms; for semantic functions it means that in a new context and given a suitable renaming into that context, we take related arguments to related results. We also require an additional condition on semantic functions, which we call uniformity, following Allais et al.[3]. However, our definition is, we believe, simpler as uniformity is just a type synonym (rather than being mutually defined with the logical relation) and we do not need to prove any auxiliary lemmas about it throughout the completeness proof. Uniformity states that if we receive a renaming and related arguments in the target context of the renaming, and then a further renaming, we can apply the function at the same context as the arguments and then rename the result or rename the arguments first and then apply the function in the later context.

It should not be possible that a semantic function can become equal to a neutral term so we rule out these cases by defining them to be $\bot$. This would not be necessary if we were doing $\beta\eta$-normalisation.

```
CR : ∀{Φ} K → Φ ⊨ K → Φ ⊨ K → Set
CR *         A        A'       = A ≡ A'
CR (K ⇒ J) (inl A)  (inl A')  = A ≡ A'
CR (K ⇒ J) (inr F)  (inl A')  = ⊥
CR (K ⇒ J) (inl A)  (inr F')  = ⊥
CR (K ⇒ J) (inr F)  (inr F')  = Unif F × Unif F' ×
  ∀ {Ψ}(ρ : Ren⋆ _ Ψ){V V' : Ψ ⊨ K} → CR K V V' → CR J (F ρ V) (F' ρ V')
  where
    -- Uniformity
    Unif : ∀{Φ K J} → (∀ {Ψ} → Ren⋆ Φ Ψ → Ψ ⊨ K → Ψ ⊨ J) → Set
    Unif {Φ}{K}{J} F = ∀{Ψ Ψ'}(ρ : Ren⋆ Φ Ψ)(ρ' : Ren⋆ Ψ Ψ')(V V' : Ψ ⊨ K)
      → CR K V V' → CR J (ren⊨ ρ' (F ρ V)) (F (ρ' ∘ ρ) (ren⊨ ρ' V'))
```

The relation CR is not an equivalence relation, it is only a partial equivalence relation (PER) as reflexivity does not hold. However, as is always the case for PERs there is a limited version of reflexivity for elements that are related to some other element.

```
symCR  : ∀{Φ K}{V V' : Φ ⊨ K}      → CR K V V' → CR K V' V
transCR : ∀{Φ K}{V V' V'' : Φ ⊨ K} → CR K V V' → CR K V' V'' → CR K V V''
reflCR  : ∀{Φ K}{V V' : Φ ⊨ K}      → CR K V V' → CR K V V
```

We think of CR as equality of semantic values. Renaming of semantic values ren⊨ (defined in the section 3.2) is a functorial action and we can prove the functor laws. The laws hold up to CR not up to propositional equality ≡:

```
ren⊨-id   : ∀{K Φ}{V V' : Φ ⊨ K} → CR K V V' → CR K (ren⊨ id V) V'
ren⊨-comp : ∀{K Φ Ψ Θ}(ρ : Ren⋆ Φ Ψ)(ρ' : Ren⋆ Ψ Θ){V V' : Φ ⊨ K}
  → CR K V V' → CR K (ren⊨ (ρ' ∘ ρ) V) (ren⊨ ρ' (ren⊨ ρ V'))
```

The completeness proof follows a similar structure as the normalisation algorithm. We define reflectCR and reifyCR analogously to the reflect and reify of the algorithm.

```
reflectCR : ∀{Φ K}{A A' : Φ ⊢Ne⋆ K} → A ≡ A'       → CR K (reflect A) (reflect A')
reifyCR   : ∀{Φ K}{V V' : Φ ⊨ K}    → CR K V V' → reify V ≡ reify V'
```

We define a pointwise partial equivalence for environments analogously to the definition of environments themselves:

```
EnvCR : ∀ {Φ Ψ} → (η η' : Env Φ Ψ) → Set
EnvCR η η' = ∀{K}(α : _ ∋⋆ K) → CR K (η α) (η' α)
```

Before defining the fundamental theorem of logical relations which is analogous to eval we define an identity extension lemma which is used to bootstrap the fundamental theorem. It states that if we evaluate a single term in related environments we get related results. Semantic renaming commutes with eval, and we prove this simultaneously with identity extension:

```
idext       : ∀{Φ Ψ K}{η η' : Env Φ Ψ} → EnvCR η η' → (A : Φ ⊢⋆ K)
  → CR K (eval A η) (eval A η')
ren⊨-eval : ∀{Φ Ψ Θ K}(A : Ψ ⊢⋆ K){η η' : Env Ψ Φ}(p : EnvCR η η')
  → (ρ : Ren⋆ Φ Θ ) → CR K (ren⊨ ρ (eval A η)) (eval A (ren⊨ ρ ∘ η'))
```

We have proved that semantic renaming commutes with evaluation. We also require that syntactic renaming commutes with evaluation: that we can either rename before evaluation or evaluate in a renamed environment:

```
ren-eval : ∀{Φ Ψ Θ K}(A : Θ ⊢⋆ K){η η' : Env Ψ Φ}(p : EnvCR η η')(ρ : Ren⋆ Θ Ψ)
  → CR K (eval (ren⋆ ρ A) η) (eval A (η' ∘ ρ))
```

As in our previous renaming lemma we require that we can either substitute and then evaluate or, equivalently, evaluate the underlying term in an environment constructed by evaluating everything in the substitution. This is the usual *substitution lemma* from denotational semantics and also one of the laws of an algebra for a relative monad (the other one holds definitionally):

```
subst-eval : ∀{Φ Ψ Θ K}(A : Θ ⊢⋆ K){η η' : Env Ψ Φ}
  → (p : EnvCR η η')(σ : Sub⋆ Θ Ψ)
  → CR K (eval (sub⋆ σ A) η) (eval A (λ α → eval (σ α) η'))
```

We can now prove the fundamental theorem of logical relations for CR. It is defined by recursion on the β-equality proof:

```
fund : ∀{Φ Ψ K}{η η' : Env Φ Ψ}{A A' : Φ ⊢⋆ K}
  → EnvCR η η' → A ≡β A' → CR K (eval A η) (eval A' η')
```

As for the ordinary identity environment, the proof that the identity environment is related to itself relies on reflection:

```
idCR : ∀{Φ} → EnvCR (idEnv Φ) (idEnv Φ)
idCR x = reflectCR refl
```

Given all these components we can prove the completeness result by running the fundamental theorem in the identity environment and then applying reification. Thus, our normalisation algorithm takes β-equal types to identical normal forms.

```
completeness : ∀ {K Φ} {A B : Φ ⊢⋆ K} → A ≡β B → nf A ≡ nf B
completeness p = reifyCR (fund idCR p)
```

Complications due to omitting the η-rule and the requirement to avoid extensionality were the main challenges in this section.

3.4 Soundness of Type Normalisation

The soundness property states that terms are β-equal to their normal forms which means that normalisation has preserved the meaning. i.e. that the unique representatives chosen by normalisation are actually in the equivalence class.

We proceed in a similar fashion to the completeness proof by defining a logical relation, reify/reflect, fundamental theorem, identity environment, and then plugging it all together to get the required result.

To state the soundness property which relates syntactic types to normal forms we need to convert normal forms back into syntactic types:

```
embNf  : ∀{Γ K} → Γ ⊢Nf⋆ K  → Γ ⊢⋆ K
embNe : ∀{Γ K} → Γ ⊢Ne⋆ K → Γ ⊢⋆ K
```

The soundness property is a Kripke Logical relation as before, defined as a Set-valued function by recursion on kinds. But this time it relates syntactic types and semantic values. In the first two cases the semantic values are normal or neutral forms and we can state the property we require easily. In the last case where we have a semantic function, we would like to state that sound functions take sound arguments to sound results (modulo the usual Kripke extension). Indeed, when doing this proof for a version of the system with $\beta\eta$-equality this was what we needed. Here, we have only β-equality for types and we were unable to get the proof to go through with the same definition. To solve this problem we added an additional requirement to the semantic function case: we require that our syntactic type of function kind A is β-equal to a λ-expression. Note this holds trivially if we have the η-rule.

```
SR : ∀{Φ} K → Φ ⊢⋆ K → Φ ⊨ K → Set
SR * A V                = A ≡β embNf V
SR (K ⇒ J) A (inl A') = A ≡β embNe A'
SR (K ⇒ J) A (inr F) = Σ (_ ,⋆ K ⊢⋆ J) λ A' → (A ≡β ƛ A') ×
  ∀{Ψ}(ρ : Ren⋆ _ Ψ){B V}
    → SR K B V → SR J (ren⋆ ρ (ƛ A') · B) (ren⊨ ρ (inr F) ·V V)
```

As before we have a notion of reify and reflect for soundness. Reflect takes soundness results about neutral terms to soundness results about semantic values and reify takes soundness results about semantic values to soundness results about normal forms:

```
reflectSR : ∀{K Φ}{A : Φ ⊢⋆ K}{A' : Φ ⊢Ne⋆ K}
  → A ≡β embNe A' → SR K A (reflect A')
reifySR : ∀{K Φ}{A : Φ ⊢⋆ K}{V : Φ ⊨ K}
  → SR K A V → A ≡β embNf (reify V)
```

We need a notion of environment for soundness, which will be used in the fundamental theorem. Here it is a lifting of the relation SR which relates syntactic types to semantic values to a relation which relates type substitutions to type environments:

```
SREnv : ∀{Φ Ψ} → Sub⋆ Φ Ψ → Env Φ Ψ → Set
SREnv {Φ} σ η = ∀{K}(α : Φ ∋⋆ K) → SR K (σ α) (η α)
```

The fundamental Theorem of Logical Relations for SR states that, for any type, if we have a related substitution and environment then the action of the substitution and environment on the type will also be related.

```
evalSR : ∀{Φ Ψ K}(A : Φ ⊢⋆ K){σ : Sub⋆ Φ Ψ}{η : Env Φ Ψ}
  → SREnv σ η → SR K (sub⋆ σ A) (eval A η)
```

The identity substitution is related to the identity environment:

```
idSR : ∀{Φ} → SREnv ‘ (idEnv Φ)
idSR = reflectSR ∘ refl≡β ∘ ‘
```

Soundness result: all types are β-equal to their normal forms.

```
soundness : ∀ {Φ J} → (A : Φ ⊢⋆ J) → A ≡β embNf (nf A)
soundness A = subst (_≡β embNf (nf A)) (sub⋆-id A) (reifySR (evalSR A idSR))
```

Complications in the definition of SR due to omitting the η-rule were the biggest challenge in this section.

3.5 Stability of Type Normalisation

The normalisation algorithm is stable: renormalising a normal form will not change it.

This property is often omitted from treatments of normalisation. For us it is crucial as in the substitution algorithm we define in the next section and in term level definitions we renormalise types.

Stability for normal forms is defined mutually with an auxiliary property for neutral types:

```
stability    : ∀{K Φ}(A : Φ ⊢Nf⋆ K) → nf (embNf A) ≡ A
stabilityNe : ∀{K Φ}(A : Φ ⊢Ne⋆ K) → eval (embNe A) (idEnv Φ) ≡ reflect A
```

We omit the proofs which are a simple simultaneous induction on normal forms and neutral terms. The most challenging part for us was getting the right statement of the stability property for neutral terms.

Stability is quite a strong property. It guarantees both that embNf ∘ nf is idempotent and that nf is surjective:

```
idempotent : ∀{Φ K}(A : Φ ⊢⋆ K)
  → (embNf ∘ nf ∘ embNf ∘ nf) A ≡ (embNf ∘ nf) A
idempotent A = cong embNf (stability (nf A))

surjective : ∀{Φ K}(A : Φ ⊢Nf⋆ K) → Σ (Φ ⊢⋆ K) λ B → nf B ≡ A
surjective A = embNf A ,, stability A
```

Note we use double comma ,, for Agda pairs as we used single comma for contexts.

3.6 Normality preserving Type Substitution

In the previous subsections we defined a normaliser. In this subsection we will combine the normaliser with our syntactic substitution operation on types to yield a normality preserving substitution. This will be used in later sections to define intrinsically typed terms with normal types. We proceed by working with similar interface as we did for ordinary substitutions.

Normality preserving substitutions are functions from type variables to normal forms:

```
SubNf⋆ : Ctx⋆ → Ctx⋆ → Set
SubNf⋆ Φ Ψ = ∀ {J} → Φ ∋⋆ J → Ψ ⊢Nf⋆ J
```

We can lift a substitution over a new bound variable as before. This is needed for going under binders.

```
liftsNf⋆ : ∀ {Φ Ψ}→ SubNf⋆ Φ Ψ → ∀{K} → SubNf⋆ (Φ ,⋆ K) (Ψ ,⋆ K)
liftsNf⋆ σ Z     = ne (` Z)
liftsNf⋆ σ (S α) = weakenNf⋆ (σ α)
```

We can extend a substitution by an additional normal type analogously to 'cons' for lists:

```
extendNf⋆ : ∀{Φ Ψ} → SubNf⋆ Φ Ψ → ∀{J}(A : Ψ ⊢Nf⋆ J) → SubNf⋆ (Φ ,⋆ J) Ψ
extendNf⋆ σ A Z = A
extendNf⋆ σ A (S α) = σ α
```

We define the action of substitutions on normal types as follows: first we embed the normal type to be acted on into a syntactic type, and compose the normalising substitution with embedding into syntactic types to turn it into an ordinary substitution, and then use our syntactic substitution operation from section 2.6. This gives us a syntactic type which we normalise using the normalisation algorithm from section 3.2. This is not efficient. It has to traverse the normal type to convert it back to a syntactic type and it may run the normalisation algorithm on things that contain no redexes. However as this is a formalisation primarily, efficiency is not a priority, correctness is.

```
subNf⋆ : ∀{Φ Ψ} → SubNf⋆ Φ Ψ → ∀ {J} → Φ ⊢Nf⋆ J → Ψ ⊢Nf⋆ J
subNf⋆ ρ n = nf (sub⋆ (embNf ∘ ρ) (embNf n))
```

We verify the same correctness properties of normalising substitution as we did for ordinary substitution: namely the relative monad laws. Note that the second law subNf⋆-∋ doesn't hold definitionally this time.

```
subNf⋆-id   : ∀{Φ J}(A : Φ ⊢Nf⋆ J) → subNf⋆ (ne ∘ ‘) A ≡ A
subNf⋆-var  : ∀{Φ Ψ J}(σ : SubNf⋆ Φ Ψ)(α : Φ ∋⋆ J)
  → subNf⋆ σ (ne (‘ α)) ≡ σ α
subNf⋆-comp : ∀{Φ Ψ Θ}(σ : SubNf⋆ Φ Ψ)(σ' : SubNf⋆ Ψ Θ){J}(A : Φ ⊢Nf⋆ J)
  → subNf⋆ (subNf⋆ σ' ∘ σ) A ≡ subNf⋆ σ' (subNf⋆ σ A)
```

These properties and the definitions that follow rely on properties of normalisation and often corresponding properties of ordinary substitution. E.g. the first law subNf⋆-id follows from stability and sub⋆-id, the second law follows directly from stability (the corresponding property holds definitionally in the ordinary case), and the third law follows from soundness, various components of completeness and sub⋆-comp.
Finally, we define the special case for single type variable substitution that will be needed in the definition of terms in the next section:

```
_[_]Nf⋆ : ∀{Φ J K} → Φ ,⋆ K ⊢Nf⋆ J → Φ ⊢Nf⋆ K → Φ ⊢Nf⋆ J
A [ B ]Nf⋆ = subNf⋆ (extendNf⋆ (ne ∘ ‘) B) A
```

The development in this section was straightforward. The most significant hurdle was that we require a complete normalisation proof and correctness properties of ordinary substitution to prove correctness properties of substitution on normal forms. The substitution algorithm in this section is essentially a rather indirect implementation of hereditary substitution.

Before moving on we list special case auxiliary lemmas that we will need when defining renaming and substitution for terms with normal types in section 5.

```
ren[]Nf⋆ : ∀{Φ Θ J K}(ρ : Ren⋆ Φ Θ)(A : Φ ,⋆ K ⊢Nf⋆ J) (B : Φ ⊢Nf⋆ K )
  → renNf⋆ ρ (A [ B ]Nf⋆) ≡ renNf⋆ (lift⋆ ρ) A [ renNf⋆ ρ B ]Nf⋆

weakenNf⋆-subNf⋆ : ∀{Φ Ψ}(σ : SubNf⋆ Φ Ψ){K}(A : Φ ⊢Nf⋆ ⋆)
  → weakenNf⋆ (subNf⋆ σ A) ≡ subNf⋆ (liftsNf⋆ σ {K = K}) (weakenNf⋆ A)

subNf⋆-liftNf⋆ : ∀{Φ Ψ}(σ : SubNf⋆ Φ Ψ){K}(B : Φ ,⋆ K ⊢Nf⋆ ⋆)
  → subNf⋆ (liftsNf⋆ σ) B
    ≡
    eval (sub⋆ (lifts⋆ (embNf ∘ σ)) (embNf B)) (lifte (idEnv Ψ))

subNf⋆-[]Nf⋆ : ∀{Φ Ψ K}(σ : SubNf⋆ Φ Ψ)(A : Φ ⊢Nf⋆ K)(B : Φ ,⋆ K ⊢Nf⋆ ⋆)
  → subNf⋆ σ (B [ A ]Nf⋆)
    ≡
    eval (sub⋆ (lifts⋆ (embNf ∘ σ)) (embNf B)) (lifte (idEnv Ψ))
    [ subNf⋆ σ A ]Nf⋆
```

3.7 Terms with normal types

We are now ready to define the algorithmic syntax where terms have normal types and the problematic conversion rule is not needed.

The definition is largely identical except wherever a syntactic type appeared before, we have a normal type, wherever an operation on syntactic types appeared before we have the corresponding operation on normal types. Note that the kind level remains the same, so we reuse Ctx⋆ for example.

Term Contexts Term level contexts are indexed by their type level contexts.

```
data CtxNf : Ctx⋆ → Set where
  ∅   :                                  CtxNf ∅
  _,⋆_ : ∀{Φ} → CtxNf Φ → ∀ J       → CtxNf (Φ ,⋆ J)
  _,_  : ∀{Φ} → CtxNf Φ → Φ ⊢Nf⋆ ⋆ → CtxNf Φ
```

Let Γ, Δ range over contexts.

Term Variables Note that in the T case, we are required to weaken (normal) types.

```
data _∋Nf_ : ∀ {Φ} → CtxNf Φ → Φ ⊢Nf⋆ ⋆ → Set where
  Z : ∀{Φ Γ}{A : Φ ⊢Nf⋆ ⋆}                  → Γ , A ∋Nf A
  S : ∀{Φ Γ}{A : Φ ⊢Nf⋆ ⋆}{B : Φ ⊢Nf⋆ ⋆} → Γ ∋Nf A → Γ , B ∋Nf A
  T : ∀{Φ Γ}{A : Φ ⊢Nf⋆ ⋆}{K}            → Γ ∋Nf A → Γ ,⋆ K ∋Nf weakenNf⋆ A
```

Let x, y range over variables.

Terms Note the absence of the conversion rule. The types of terms are unique so it is not possible to coerce a term into a different type.

```
data _⊢Nf_ {Φ} Γ : Φ ⊢Nf⋆ ⋆ → Set where
  `       : ∀{A}     → Γ ∋Nf A                    → Γ ⊢Nf A
  ƛ       : ∀{A B}   → Γ , A ⊢Nf B                → Γ ⊢Nf A ⇒ B
  _·_     : ∀{A B}   → Γ ⊢Nf A ⇒ B → Γ ⊢Nf A      → Γ ⊢Nf B
  Λ       : ∀{K B}   → Γ ,⋆ K ⊢Nf B               → Γ ⊢Nf Π B
  _·⋆_    : ∀{K B}   → Γ ⊢Nf Π B → (A : Φ ⊢Nf⋆ K) → Γ ⊢Nf B [ A ]Nf⋆
  wrap    : ∀ A      → Γ ⊢Nf A [ μ A ]Nf⋆          → Γ ⊢Nf μ A
  unwrap  : ∀{A}     → Γ ⊢Nf μ A                   → Γ ⊢Nf A [ μ A ]Nf⋆
```

Let L, M range over terms.

We now have an intrinsically typed definition of terms with types that are guaranteed to be normal. By side-stepping the conversion problem we can define an operational semantics for this syntax which we will do in section 5. In the next section we will reflect on the correspondence between this syntax and the syntax with conversion presented in section 2.

We define two special cases of subst which allow us to substitute the types of variables or terms by propositionally equal types. While it is the case that types are now represented uniquely we still want or need to to prove that two types are equal, especially in the presence of (Agda) variables, cf., while the natural number 7 has a unique representation in Agda we still might want to prove that for any natural numbers m and n, $m + n \equiv n + m$.

```
conv∋Nf : ∀ {Φ Γ}{A A' : Φ ⊢Nf⋆ *} → A ≡ A' → (Γ ∋Nf A) → Γ ∋Nf A'
conv∋Nf refl α = α

conv⊢ : ∀ {Φ Γ}{A : Φ ⊢⋆ *}{A' : Φ ⊢⋆ *} → A ≡ A' → Γ ⊢ A → Γ ⊢ A'
conv⊢ refl α = α
```

We see these operations in in use in section 5.

4 Correspondence between declarative and algorithmic type systems

We now have two versions of the syntax/typing rules. Should we just throw away the old one and use the new one? No. The first version is the standard textbook version and the second version is an algorithmic version suitable for implementation. To reconcile the two we prove the second version is sound and complete with respect to the first. This is analogous to proving that a typechecker is sound and complete with respect to the typing rules. Additionally, we prove that before and after normalising the type, terms erase to the same untyped terms. The constructions in this section became significantly simpler and easier after switching from inductive-recursive term contexts to indexed term contexts.

There is an interesting parallel here with the metatheory of Twelf[4]. In Twelf, hereditary substitution are central to the metatheory and the semantics is defined on a version of the syntax where both types and terms are canonical (i.e. they are normalised). In our setting only the types are normalised (viz. canonical). But, the situation is similar: there are two versions of the syntax, one with a semantics (the canonical system), and one without (the ordinary system). Martens and Crary[28] make the case that the ordinary version is the programmer's interface, or the external language in compiler terminology, and the canonical version is the internal language in compiler terminology. In their setting the payoff is also the same: by moving from a language with type equivalence to one where types are uniquely represented, the semantics and metatheory become much simpler.

There is also a parallel with how type checking algorithms are described in the literature: they are often presented an alternative set of typing rules and then they are proved sound and complete with respect to the original typing rules. We will draw on this analogy in the rest of this section as our syntaxes are also type systems.

[4] We thank an anonymous reviewer for bringing this to our attention.

4.1 Soundness of Typing

From a typing point of view, soundness states that anything typeable in the new type system is also typeable in the old one. From our syntactic point of view this corresponds to taking an algorithmic term and embedding it back into a declarative term.

We have already defined an operation to embed normal types into syntactic types. But, we need an additional operation here: term contexts contain types so we must embed term contexts with normal type into term contexts with syntactic types.

```
embCtx : ∀{Φ} → CtxNf Φ → Ctx Φ
embCtx ∅          = ∅
embCtx (Γ ,⋆ K)   = embCtx Γ ,⋆ K
embCtx (Γ , A)    = embCtx Γ , embNf A
```

Embedding for terms takes a term with a normal type and produces a term with a syntactic type.

```
embTy : ∀{Φ Γ}{A : Φ ⊢Nf⋆ ⋆} → Γ ⊢Nf A → embCtx Γ ⊢ embNf A
```

Soundness of typing is a direct corollary of embTy:

```
soundnessT : ∀{Φ Γ}{A : Φ ⊢Nf⋆ ⋆} → Γ ⊢Nf A → embCtx Γ ⊢ embNf A
soundnessT = embTy
```

Soundness gives us one direction of the correspondence between systems. The other direction is given by completeness.

4.2 Completeness of Typing

Completeness of typing states that anything typeable by the original declarative system is typeable by the new system, i.e. we do not lose any well typed programs by moving to the new system. From our syntactic point of view, it states that we can take any declarative term of a given type and normalise its type to produce an algorithmic term with a type that is β-equal to the type we started with.

We have already defined normalisation for types. Again, we must provide an operation that normalises a context:

```
nfCtx : ∀{Φ} → Ctx Φ → CtxNf Φ
nfCtx ∅          = ∅
nfCtx (Γ ,⋆ K)   = nfCtx Γ ,⋆ K
nfCtx (Γ , A)    = nfCtx Γ , nf A
```

We observe at this point (just before we use it) that conversion is derivable for the algorithmic syntax. It computes:

```
convNf : ∀ {Φ Γ}{A A' : Φ ⊢Nf⋆ ⋆} → A ≡ A' → Γ ⊢Nf A → Γ ⊢Nf A'
convNf refl L = L
```

The operation that normalises the types of terms takes a declarative term and produces an algorithmic term. We omit the majority of the definition, but include the case for a conversion. In this case we have a term t of type $\Gamma \vdash A$ and a proof p that $A \equiv_\beta B$. We require a term of of type $\Gamma \vdash\mathsf{Nf}\ \mathsf{nf}\ B$. By inductive hypothesis/recursive call $\mathsf{nfType}\ t : \Gamma \vdash\mathsf{Nf}\ \mathsf{nf}\ A$. But, via completeness of normalisation we know that if $A \equiv_\beta B$ then $\mathsf{nf}\ B \equiv \mathsf{nf}\ A$, so we invoke the conversion function $\mathsf{conv}\vdash\mathsf{Nf}$ with the completeness proof and and the recursive call as arguments:

```
nfType : ∀{Φ Γ}{A : Φ ⊢⋆ *} → Γ ⊢ A → nfCtx Γ ⊢Nf nf A
nfType (conv p t) = conv⊢Nf (completeness p) (nfType t)
```

⋮ (remaining cases omitted)

The operation nfType is not quite the same as completeness. Additionally we need that the original type is β-equal to the new type. This follows from soundness of normalisation.

```
completenessT : ∀{Φ Γ}{A : Φ ⊢⋆ *} → Γ ⊢ A
  → nfCtx Γ ⊢Nf nf A × (A ≡β embNf (nf A))
completenessT {A = A} t = nfType t ,, soundness A
```

4.3 Erasure

We have two version of terms, and we can convert from one to the other. But, how do we know that after conversion, we still have the *same* term? One answer is to show that that the term before conversion and the term after conversion both erase to the same untyped term. First, we define untyped (but intrinsically scoped) λ-terms:

```
data _⊢ : ℕ → Set where
  `   : ∀{n} → Fin n → n ⊢
  ƛ   : ∀{n} → suc n ⊢ → n ⊢
  _·_ : ∀{n} → n ⊢ → n ⊢ → n ⊢
```

Following the pattern of the soundness and completeness proofs we deal in turn with contexts, variables, and then terms. In this case erasing a context corresponds to counting the number of term variables in the context:

```
len : ∀{Φ} → Ctx Φ → ℕ
len ∅       = 0
len (Γ ,⋆ K) = len Γ
len (Γ , A)  = suc (len Γ)
```

Erasure for variables converts them to elements of Fin:

```
eraseVar : ∀{Φ Γ}{A : Φ ⊢⋆ *} → Γ ∋ A → Fin (len Γ)
eraseVar Z     = zero
```

```
eraseVar (S α) = suc (eraseVar α)
eraseVar (T α) = eraseVar α
```

Erasure for terms is straightforward:

```
erase : ∀{Φ Γ}{A : Φ ⊢⋆ *} → Γ ⊢ A → len Γ ⊢
erase (` α)          = ` (eraseVar α)
erase (ƛ L)          = ƛ (erase L)
erase (L · M)        = erase L · erase M
erase (Λ L)          = erase L
erase (L ·⋆ A)       = erase L
erase (wrap A L)     = erase L
erase (unwrap L)     = erase L
erase (conv p L)     = erase L
```

Note that we drop wrap and unwrap when erasing as these special type casts merely indicate at which isomorphic type we want the term to considered. Without types wrap and unwrap serve no purpose.

Erasure from algorithmic terms proceeds in the same way as declarative terms. The only difference is the that there is no case for conv:

```
lenNf       : ∀{Φ} → CtxNf Φ → ℕ
eraseVarNf : ∀{Φ Γ}{A : Φ ⊢Nf⋆ *} → Γ ∋Nf A → Fin (lenNf Γ)
eraseNf     : ∀{Φ Γ}{A : Φ ⊢Nf⋆ *} → Γ ⊢Nf A → lenNf Γ ⊢
```

Having defined erasure for both term representations we proceed with the proof that normalising types preserves meaning of terms. We deal with contexts first, then variables, and then terms. Normalising types in the context preserves the number of term variables in the context:

```
sameLen : ∀ {Φ}(Γ : Ctx Φ) → lenNf (nfCtx Γ) ≡ len Γ
```

The main complication in the proofs about variables and terms below is that sameLen appears in the types. It complicates each case as the subst prevents things from computing when its proof argument is not refl. This can be worked around using Agda's *with* feature which allows us to abstract over additional arguments such as those which are stuck. However in this case we would need to abstract over so many arguments that the proof becomes unreadable. Instead we prove a simple lemma for each case which achieves the same as using *with*. We show the simplest instance lemzero for the Z variable which abstracts over proof of sameLen and replaces it with an arbitrary proof p that we can pattern match on.

```
lemzero : ∀{n n'}(p : suc n ≡ suc n') → zero ≡ subst Fin p zero
lemzero refl = refl

sameVar : ∀{Φ Γ}{A : Φ ⊢⋆ *}(x : Γ ∋ A)
  → eraseVar x ≡ subst Fin (sameLen Γ) (eraseVarNf (nfTyVar x))
sameVar {Γ = Γ , _} Z = lemzero (cong suc (sameLen Γ))
```

```
⋮  (remaining cases omitted)
```

```
same : ∀{Φ Γ}{A : Φ ⊢⋆ *}(t : Γ ⊢ A)
  → erase t ≡ subst _⊢ (sameLen Γ) (eraseNf (nfType t))
```

This result indicates that when normalising the type of a term we preserve the meaning of the term where the *meaning* of a term is taken to be the underlying untyped term.

A similar result holds for embedding terms with normal types back into terms with ordinary type but we omit it here.

5 Operational Semantics

We will define the operational semantics on the algorithmic syntax. Indeed, this was the motivation for introducing the algorithmic syntax: to provide a straightforward way to define the semantics without having to deal with type equality coercions. The operational semantics is defined as a call-by-value small-step reduction relation. The relation is typed so it is not necessary to prove preservation as it holds intrinsically. We prove progress for this relation which shows that programs cannot get stuck. As the reduction relation contains β-rules we need to implement substitution for algorithmic terms before proceeding. As we did for types, we define renaming first and then use it to define substitution.

5.1 Renaming for terms

We index term level renamings/substitutions by their type level counter parts.

Renamings are functions from term variables to terms. The type of the output variable is the type of the input variable renamed by the type level renaming.

```
RenNf : ∀ {Φ Ψ} Γ Δ → Ren⋆ Φ Ψ → Set
RenNf Γ Δ ρ = {A : _ ⊢Nf⋆ *} → Γ ∋Nf A → Δ ∋Nf renNf⋆ ρ A
```

We can lift a renaming both over a new term variable and over a new type variable. These operations are needed to push renamings under binders (λ and Λ respectively).

```
liftNf : ∀{Φ Ψ Γ Δ}{ρ⋆ : Ren⋆ Φ Ψ} → RenNf Γ Δ ρ⋆
  → {B : Φ ⊢Nf⋆ *} → RenNf (Γ , B) (Δ , renNf⋆ ρ⋆ B) ρ⋆
liftNf ρ Z = Z
liftNf ρ (S x) = S (ρ x)

⋆liftNf : ∀{Φ Ψ Γ Δ}{ρ⋆ : Ren⋆ Φ Ψ} → RenNf Γ Δ ρ⋆
  → (∀ {K} → RenNf (Γ ,⋆ K) (Δ ,⋆ K) (lift⋆ ρ⋆))
⋆liftNf ρ (T x) = conv∋Nf (trans (sym (renNf⋆-comp _)) (renNf⋆-comp _)) (T (ρ x))
```

Next we define the functorial action of renaming on terms. In the type instantiation, wrap, unwrap cases we need a proof as this is where substitutions appear in types.

```
renNf : ∀ {Φ Ψ Γ Δ}{ρ⋆ : Ren⋆ Φ Ψ} → RenNf Γ Δ ρ⋆
  → ({A : Φ ⊢Nf⋆ *} → Γ ⊢Nf A → Δ ⊢Nf renNf⋆ ρ⋆ A )
renNf ρ (` x)                = ` (ρ x)
renNf ρ (ƛ N)                = ƛ (renNf (liftNf ρ) N)
renNf ρ (L · M)              = renNf ρ L · renNf ρ M
renNf ρ (Λ N)                = Λ (renNf (⋆liftNf ρ) N)
renNf ρ (_·⋆_ {B = B} t A)   =
  conv⊢Nf (sym (ren[]Nf⋆ _ B A)) (renNf ρ t ·⋆ renNf⋆ _ A)
renNf ρ (wrap A L)           =
  wrap _ (conv⊢Nf (ren[]Nf⋆ _ A (μ A)) (renNf ρ L))
renNf ρ (unwrap {A = A} L) =
  conv⊢Nf (sym (ren[]Nf⋆ _ A (μ A))) (unwrap (renNf ρ L))
```

Weakening by a type is a special case. Another proof is needed here.

```
weakenNf : ∀ {Φ Γ}{A : Φ ⊢Nf⋆ *}{B : Φ ⊢Nf⋆ *} → Γ ⊢Nf A → Γ , B ⊢Nf A
weakenNf {A = A} x =
  conv⊢Nf (renNf⋆-id A) (renNf (conv∋Nf (sym (renNf⋆-id _)) ∘ S) x)
```

We can also weaken by a kind:

```
⋆weakenNf : ∀ {Φ Γ}{A : Φ ⊢Nf⋆ *}{K} → Γ ⊢Nf A → Γ ,⋆ K ⊢Nf weakenNf⋆ A
⋆weakenNf x = renNf T x
```

5.2 Substitution

Substitutions are defined as functions from type variables to terms. Like renamings they are indexed by their type level counterpart, which is used in the return type.

```
SubNf : ∀ {Φ Ψ} Γ Δ → SubNf⋆ Φ Ψ → Set
SubNf Γ Δ ρ = {A : _ ⊢Nf⋆ *} → Γ ∋Nf A → Δ ⊢Nf subNf⋆ ρ A
```

We define lifting of a substitution over a type and a kind so that we can push substitutions under binders. Agda is not able to infer the type level normalising substitution in many cases so we include it explicitly.

```
liftsNf : ∀{Φ Ψ Γ Δ}(σ⋆ : SubNf⋆ Φ Ψ) → SubNf Γ Δ σ⋆
  → {B : _ ⊢Nf⋆ *} → SubNf (Γ , B) (Δ , subNf⋆ σ⋆ B) σ⋆
liftsNf _ σ Z = ` Z
liftsNf _ σ (S x) = weakenNf (σ x)

⋆liftsNf : ∀{Φ Ψ Γ Δ}(σ⋆ : SubNf⋆ Φ Ψ) → SubNf Γ Δ σ⋆
  → ∀ {K} → SubNf (Γ ,⋆ K) (Δ ,⋆ K) (liftsNf⋆ σ⋆)
⋆liftsNf σ⋆ σ (T {A = A} x) =
  conv⊢Nf (weakenNf⋆-subNf⋆ σ⋆ A) (⋆weakenNf (σ x))
```

Having defined lifting we are now ready to define substitution on terms:

```
subNf : ∀{Φ Ψ Γ Δ}(σ⋆ : SubNf⋆ Φ Ψ) → SubNf Γ Δ σ⋆
  → ({A : Φ ⊢Nf⋆ ⋆} → Γ ⊢Nf A → Δ ⊢Nf subNf⋆ σ⋆ A)
subNf σ⋆ σ (` k)                = σ k
subNf σ⋆ σ (ƛ N)                = ƛ (subNf σ⋆ (liftsNf σ⋆ σ) N)
subNf σ⋆ σ (L · M)              = subNf σ⋆ σ L · subNf σ⋆ σ M
subNf σ⋆ σ (Λ {B = B} N)        =
  Λ (conv⊢Nf (subNf⋆-liftNf⋆ σ⋆ B) (subNf (liftsNf⋆ σ⋆) (⋆liftsNf σ⋆ σ) N))
subNf σ⋆ σ (_·⋆_ {B = B} L M)   =
  conv⊢Nf (sym (subNf⋆-[]Nf⋆ σ⋆ M B)) (subNf σ⋆ σ L ·⋆ subNf⋆ σ⋆ M)
subNf σ⋆ σ (wrap A L)           =
  wrap _ (conv⊢Nf (subNf⋆-[]Nf⋆ σ⋆ (μ A) A) (subNf σ⋆ σ L))
subNf σ⋆ σ (unwrap {A = A} L) =
  conv⊢Nf (sym (subNf⋆-[]Nf⋆ σ⋆ (μ A) A)) (unwrap (subNf σ⋆ σ L))
```

We define special cases for single type and term variable substitution into a term, but omit their long winded and not very informative definitions.

```
_[_]Nf   : ∀{Φ Γ}{A B : Φ ⊢Nf⋆ ⋆} → Γ , B ⊢Nf A → Γ ⊢Nf B → Γ ⊢Nf A
_⋆[_]Nf : ∀{Φ Γ K}{B : Φ ,⋆ K ⊢Nf⋆ ⋆}
  → Γ ,⋆ K ⊢Nf B → (A : Φ ⊢Nf⋆ K) → Γ ⊢Nf B [ A ]Nf⋆
```

We now have all the equipment we need to specify small-step reduction.

5.3 Reduction

Before defining the reduction relation we define a value predicate on terms that captures which terms cannot be reduced any further. We do not wish to perform unnecessary computation so we do not compute under the binder in the ƛ case. However, we do want to have the property that when you erase a value it remains a value. A typed value, after erasure, should not require any further reduction to become an untyped value. This gives us a close correspondence between the typed and untyped operational semantics. So, it is essential in the Λ and wrap cases that the bodies are values as both of these constructors are removed by erasure.

```
data Value {Φ}{Γ} : {A : Φ ⊢Nf⋆ ⋆} → Γ ⊢Nf A → Set where
  V-ƛ    : ∀{A B}{L : Γ , A ⊢Nf B}                → Value (ƛ L)
  V-Λ    : ∀{K B}{L : Γ ,⋆ K ⊢Nf B}   → Value L → Value (Λ L)
  V-wrap : ∀{A}{L : Γ ⊢Nf A [ μ A ]Nf⋆} → Value L → Value (wrap A L)
```

We give the dynamics of the term language as a small-step reduction relation. The relation is is typed and terms on the left and right hand side have the same type so it is impossible to violate preservation. We have two congruence (xi) rules for application and only one for type application, types are unique so the type argument cannot reduce. Indeed, no reduction of types is either possible or needed. There are three computation (beta) rules, one for application, one for type application and one for recursive types. We allow reduction in almost any term argument in the xi rules except under a ƛ. Allowing reduction under Λ and wrap is required to ensure that their bodies become values. The

value condition on the function term in rule $\xi\text{-}\cdot_2$ ensures that, in an application, we reduce the function before the argument. The value condition on the argument in rule $\beta\text{-}\lambda$ ensures that the our semantics is call-by-value.

```
data _⟶_ {Φ}{Γ} : {A : Φ ⊢Nf⋆ ⋆} → (Γ ⊢Nf A) → (Γ ⊢Nf A) → Set where
  ξ-·1      : ∀{A B}{L L' : Γ ⊢Nf A ⇒ B}{M : Γ ⊢Nf A}
    → L ⟶ L' → L · M ⟶ L' · M
  ξ-·2      : ∀{A B}{V : Γ ⊢Nf A ⇒ B}{M M' : Γ ⊢Nf A}
    → Value V → M ⟶ M' → V · M ⟶ V · M'
  ξ-Λ       : ∀{K B}{L L' : Γ ,⋆ K ⊢Nf B}
    → L ⟶ L' → Λ L ⟶ Λ L'
  ξ-·⋆      : ∀{K B}{L L' : Γ ⊢Nf Π B}{A : Φ ⊢Nf⋆ K}
    → L ⟶ L' → L ·⋆ A ⟶ L' ·⋆ A
  ξ-unwrap : ∀{A}{L L' : Γ ⊢Nf μ A}
    → L ⟶ L' → unwrap L ⟶ unwrap L'
  ξ-wrap : {A : Φ ,⋆ ⋆ ⊢Nf⋆ ⋆}{L L' : Γ ⊢Nf A [ μ A ]Nf⋆}
    → L ⟶ L' → wrap A L ⟶ wrap A L'
  β-ƛ       : ∀{A B}{L : Γ , A ⊢Nf B}{M : Γ ⊢Nf A}
    → Value M → ƛ L · M ⟶ L [ M ]Nf
  β-Λ       : ∀{K B}{L : Γ ,⋆ K ⊢Nf B}{A : Φ ⊢Nf⋆ K}
    → Value L
    → Λ L ·⋆ A ⟶ L ⋆[ A ]Nf
  β-wrap    : ∀{A}{L : Γ ⊢Nf A [ μ A ]Nf⋆} → Value L
    → unwrap (wrap A L) ⟶ L
```

5.4 Progress and preservation

The reduction relation is typed. The definition guarantees that the terms before and after reduction will have the same type. Therefore it is not necessary to prove type preservation.

Progress captures the property that reduction of terms should not get stuck, either a term is already a value or it can make a reduction step. Progress requires proof. We show the proof in complete detail. In an earlier version of this work when we did not reduce under Λ and we proved progress directly for closed terms, i.e. for terms in the empty context. Reducing under the Λ binder means that we need to reduce in non-empty contexts so our previous simple approach no longer works.

There are several approaches to solving this including: (1) modifying term syntax to ensure that the bodies of Λ-expressions are already in fully reduced form (the so-called *value restriction*). This means that we need only make progress in the empty context as no further progress is necessary when we are in a non-empty context. This has the downside of changing the language slightly but keeps progress simple; (2) defining neutral terms (terms whose reduction is blocked by a variable), proving a version of progress for open terms, observing that there are no neutral terms in the empty context and deriving progress for closed terms as a corollary. This has the disadvantage of having to introduce neutral terms only to rule them out and complicating the progress

proof; (3) observe that Λ only binds type variables and not term variables and only term variables can block progress, prove progress for terms in contexts that contain no term variables and derive closed progress as a simple corollary. We choose option 3 here as the language remains the same and the progress proof is relatively unchanged, it just requires an extra condition on the context. The only cost is an additional predicate on contexts and an additional lemma.

Before starting the progress proof we need to capture the property of a context not containing any term variables. Our term contexts are indexed by type contexts, if we wanted to rule out type variables we could talk about term contexts indexed by the empty type context, but we cannot use the same trick for ruling out term variables. So, we use a recursive predicate on contexts NoVar. The empty context satisfies it, a context extended by (the kind of) a type variable does if the underlying context does, and a context containing (the type of) a term variable does not.

```
NoVar : ∀{Φ} → CtxNf Φ → Set
NoVar ∅ = ⊤
NoVar (Γ ,⋆ J) = NoVar Γ
NoVar (Γ , A) = ⊥
```

We can prove easily that it is impossible to have term variable in a context containing no term variables. There is only one case and the property follows by induction on variables:

```
noVar : ∀{Φ Γ} → NoVar Γ → {A : Φ ⊢Nf⋆ ⋆}(x : Γ ∋Nf A) → ⊥
noVar p (T x) = noVar p x
```

We can now prove progress. The proof is the same as the one for closed terms, except for the extra argument p : NoVar Γ.

```
progress : ∀{Φ}{Γ} → NoVar Γ → {A : Φ ⊢Nf⋆ ⋆}(L : Γ ⊢Nf A)
  → Value L ⊎ Σ (Γ ⊢Nf A) λ L' → L —→ L'
```

The variable case is impossible.

```
progress p (` x) = ⊥-elim (noVar p x)
```

Any λ-expression is a value as we do not reduce under the binder.

```
progress p (ƛ L) = inl (V-ƛ L)
```

In the application case we first examine the result of the recursive call on the function term, if it is a value, it must be a λ-expression, so we examine the recursive call on the argument term. If this is a value then we perform β-reduction. Otherwise we make the appropriate ξ-step.

```
progress p (L · M)    with progress p L
progress p (L · M)    — inl V        with progress p M
progress p (ƛ L · M) — inl (V-ƛ L) — inl V          = inr (L [ M ]Nf ,, β-ƛ V)
```

```
progress p (L · M)    — inl V        — inr (M' ,, q) = inr (L · M' ,, ξ-·₂ V q)
progress p (L · M)    — inr (L' ,, q) = inr (L' · M ,, ξ-·1 q)
```

As we must reduce under Λ and wrap in both cases we make a recursive call on their bodies and proceed accordingly. Notice that the argument p is unchanged in the recursive call to the body of a Λ as NoVar (Γ ,⋆ K) = NoVar Γ.

```
progress p (Λ L) with progress p L
...                  — inl V        = inl (V-Λ V)
...                  — inr (L' ,, q) = inr (Λ L' ,, ξ-Λ q)
progress p (wrap A L) with progress p L
...                       — inl V        = inl (V-wrap V)
...                       — inr (L' ,, q) = inr (wrap A L' ,, ξ-wrap q)
```

In the type application case we first examine the result of recursive call on the type function argument. If it is a value it must be a Λ-expression and we perform β-reduction. Otherwise we perform a ξ-step.

```
progress p (L ·⋆ A)    with progress p L
progress p (Λ L ·⋆ A) — inl (V-Λ V) = inr (L ⋆[ A ]Nf ,, (β-Λ V))
progress p (L ·⋆ A)    — inr (L' ,, q) = inr (L' ·⋆ A ,, ξ-·⋆ q)
```

In the unwrap case we examine the result of the recursive call on the body. If it is a value it must be a wrap and we perform β-reduction or a ξ-step otherwise. That completes the proof.

```
progress p (unwrap L)             with progress p L
progress p (unwrap (wrap A L)) — inl (V-wrap V) = inr (L ,, β-wrap V)
progress p (unwrap L)             — inr (L' ,, q)     = inr (unwrap L' ,, ξ-unwrap q)
```

Progress in the empty context progress∅ is a simple corollary. The empty context trivially satisfies NoVar as NoVar ∅ = ⊤:

```
progress∅ : ∀{A}(L : ∅ ⊢Nf A) → Value L ⊎ Σ (∅ ⊢Nf A) λ L' → L —→ L'
progress∅ = progress tt
```

5.5 Erasure

We can extend our treatment of erasure from syntax to (operational) semantics. Indeed, when defining values were careful to ensure this was possible.
To define the β-rule we need need to be able to perform substitution on one variable only. As for syntaxes in earlier sections we define parallel renaming and substitution first and get substitution on one variable as a special case. We omit the details here which are analogous to earlier sections.

```
_[_]U : ∀{n} → suc n ⊢ → n ⊢ → n ⊢
```

When erasing reduction steps below we will require two properties about pushing erasure through a normalising single variable substitution. These properties follow from properties of parallel renaming and substitution:

```
eraseNf-*[]Nf : ∀{Φ}{Γ : CtxNf Φ}{K B}(L : Γ ,* K ⊢Nf B)(A : Φ ⊢Nf* K)
  → eraseNf L ≡ eraseNf (L *[ A ]Nf)
eraseNf-[]Nf : ∀{Φ}{Γ : CtxNf Φ}{A B}(L : Γ , A ⊢Nf B)(M : Γ ⊢Nf A)
  → eraseNf L [ eraseNf M ]U ≡ eraseNf (L [ M ]Nf)
```

There is only one value in untyped lambda calculus: lambda.

```
data UValue {n} : n ⊢ → Set where
  V-ƛ : (t : suc n ⊢) → UValue (ƛ t)
```

We define a call-by-value small-step reduction relation that is intrinsically scoped.

```
data _U—→_ {n} : n ⊢ → n ⊢ → Set where
  ξ-·1 : {L L' M : n ⊢} → L U—→ L' → L · M U—→ L' · M
  ξ-· 2 : {L M M' : n ⊢} → UValue L → M U—→ M' → L · M U—→ L · M'
  β-ƛ : {L : suc n ⊢}{M : n ⊢} → UValue M → ƛ L · M U—→ L [ M ]U
```

Erasing values is straightforward. The only tricky part is to ensure that in values the subterms of the values for wrap and Λ are also values as discussed earlier. This ensures that after when we erase a typed value we will always get an untyped value:

```
eraseVal : ∀{Φ A}{Γ : CtxNf Φ}{t : Γ ⊢Nf A} → Value t → UValue (eraseNf t)
eraseVal (V-ƛ t)  = V-ƛ (eraseNf t)
eraseVal (V-Λ v) = eraseVal v
eraseVal (V-wrap v) = eraseVal v
```

Erasing a reduction step is more subtle as we may either get a typed reduction step (e.g., β-ƛ) or the step may disappear (e.g., β-wrap). In the latter case the erasure of the terms before and after reduction will be identical:

```
erase—→ : ∀{Φ A}{Γ : CtxNf Φ}{t t' : Γ ⊢Nf A}
  → t —→ t' → eraseNf t U—→ eraseNf t' ⊎ eraseNf t ≡ eraseNf t'
```

In the congruence cases for application what we need to do depends on the result of erasing the underlying reduction step. We make use of map for Sum types for this purpose, the first argument explains what to do if the underlying step corresponds to a untyped reduction step (we create an untyped congruence reducing step) and the second argument explains what to do if the underlying step disappears (we create an equality proof):

```
erase—→ (ξ-·1 {M = M} p)  =
  Sum.map ξ-·1 (cong (_· eraseNf M)) (erase—→ p)
erase—→ (ξ-· 2 {V = V} p q) =
  Sum.map (ξ-· 2 (eraseVal p)) (cong (eraseNf V ·_)) (erase—→ q)
```

In the following cases the outer reduction step is removed:

```
erase—→ (ξ-·⋆ p)       = erase—→ p
erase—→ (ξ-Λ p)        = erase—→ p
erase—→ (ξ-unwrap p) = erase—→ p
erase—→ (ξ-wrap p)     = erase—→ p
```

In the case of β-reduction for an ordinary application we always produce a corresponding untyped β-reduction step:

```
erase—→ (β-ƛ {L = L}{M = M} V) = inl (subst
  (ƛ (eraseNf L) · eraseNf M U—→_)
  (eraseNf-[]Nf L M)
  (_U—→_.β-ƛ {L = eraseNf L}{M = eraseNf M} (eraseVal V)))
```

In the other two β-reduction cases the step is always removed, e.g., unwrap (wrap A L) $\longrightarrow$ L becomes $L \equiv L$:

```
erase—→ (β-Λ {L = L}{A = A} V) = inr (eraseNf-⋆[]Nf L A)
erase—→ (β-wrap _)             = inr refl
```

That concludes the proof: either a typed reduction step corresponds to an untyped one or no step at all.

We can combine erasure of values and reduction steps to get a progress like result for untyped terms via erasure. Via typed progress we either arrive immediately at an untyped value, or a typed reduction step must exist and it will corr respond to an untyped step, or the step disappears:

```
erase-progress∅ : ∀{A : ∅ ⊢Nf⋆ *}(L : ∅ ⊢Nf A)
  → UValue (eraseNf L)
  ⊎ Σ (∅ ⊢Nf A) λ L' → (L —→ L')
    × (eraseNf L U—→ eraseNf L' ⊎ eraseNf L ≡ eraseNf L')
erase-progress∅ L =
  Sum.map eraseVal (λ {(L' ,, p) → L' ,, p ,, (erase—→ p)}) (progress∅ L)
```

6 Execution

We can iterate progress an arbitrary number of times to run programs. First, we define the reflexive transitive closure of reduction. We will use this to represent traces of execution:

```
data _—→*_ {Φ Γ} : {A : Φ ⊢Nf⋆ *} → Γ ⊢Nf A → Γ ⊢Nf A → Set where
  refl—→ : ∀{A}{M : Γ ⊢Nf A} → M —→* M
  trans—→ : ∀{A}{M M' M" : Γ ⊢Nf A}
    → M —→ M' → M' —→* M" → M —→* M"
```

The run function takes a number of allowed steps and a term. It returns another term, a proof that the original term reduces to the new term in zero or more steps and possibly a proof that the new term is a value. If no value proof is returned this indicates that we did not reach a value in the allowed number of steps.

If we are allowed zero more steps we return failure immediately. If we are allowed more steps then we call progress to make one. If we get a value back we return straight away with a value. If we have not yet reached a value we call run recursively having spent a step. We then prepend our step to the sequence of steps returned by run and return:

```
run : ∀ {A : ∅ ⊢Nf⋆ *} → ℕ → (M : ∅ ⊢Nf A)
  → Σ (∅ ⊢Nf A) λ M' → (M —→⋆ M') × Maybe (Value M')
run zero M = M ,, refl—→ ,, nothing
run (suc n) M with progress∅ M
run (suc n) M — inl V = M ,, refl—→ ,, just V
run (suc n) M — inr (M' ,, p) with run n M'
... — M'' ,, q ,, mV = M'' ,, trans—→ p q ,, mV
```

6.1 Erasure

Given that the evaluator run produces a trace of reduction that (if it doesn't run out of allowed steps) leads to a value we can erase the trace and value to yield a trace of untyped execution leading to an untyped value. Note that the untyped trace may be shorter as some steps may disappear.

We define the the reflexive transitive closure of untyped reduction analogously to the typed version:

```
data _U—→⋆_ {n} : n ⊢ → n ⊢ → Set where
  reflU—→  : {M : n ⊢} → M U—→⋆ M
  transU—→ : {M M' M'' : n ⊢}
    → M U—→ M' → M' U—→⋆ M'' → M U—→⋆ M''
```

We can erase a typed trace to yield an untyped trace. The reflexive case is straightforwards. In the transitive case, we may have a step p that corresponds to an untyped or it may disappear. We use case [_,_] instead of map this time. It is like map but instead of producing another sum it (in the non-dependent case that we are in) produces a result of an the same type in each case (in our case erase M —→ erase M''). In the first case we get an untyped *step* and rest of the trace is handled by the recursive call. In the second case eq is an equation erase M ≡ erase M' which we use to coerce the recursive call of type erase M' —→ erase M'' into type erase M —→ erase M'' and the length of the trace is reduced:

```
erase—→⋆ : ∀{Φ}{A : Φ ⊢Nf⋆ *}{Γ : CtxNf Φ}{t t' : Γ ⊢Nf A}
  → t —→⋆ t' → eraseNf t U—→⋆ eraseNf t'
erase—→⋆ refl—→ = reflU—→
erase—→⋆ (trans—→ {M'' = M''} p q) =
```

```
  [ (λ step → transU—→ step (erase—→* q))
  , (λ eq → subst (_U—→* eraseNf M'') (sym eq) (erase—→* q))
  ] (erase—→ p)
```

Finally we can use run to get an untyped trace leading to a value, allowed steps permitting.

```
erase-run : ∀ {A : ∅ ⊢Nf⋆ *} → ℕ → (M : ∅ ⊢Nf A)
  → Σ (0 ⊢) λ M' → (eraseNf M U—→* M') × Maybe (UValue M')
erase-run n M with run n M
... | M' ,, p ,, mv = eraseNf M' ,, erase—→* p ,, Maybe.map eraseVal mv
```

7 Examples

Using only the facilities of System F without the extensions of type functions and recursive types we can define natural numbers as Church Numerals:

```
ℕᶜ : ∀{Φ} → Φ ⊢Nf⋆ *
ℕᶜ = Π ((ne (` Z)) ⇒ (ne (` Z) ⇒ ne (` Z)) ⇒ (ne (` Z)))

Zeroᶜ : ∀{Φ}{Γ : CtxNf Φ} → Γ ⊢Nf ℕᶜ
Zeroᶜ = Λ (ƛ (ƛ (` (S Z))))

Succᶜ : ∀{Φ}{Γ : CtxNf Φ} → Γ ⊢Nf ℕᶜ ⇒ ℕᶜ
Succᶜ = ƛ (Λ (ƛ (ƛ (` Z · ((` (S (S (T Z)))) ·⋆ (ne (` Z)) · (` (S Z)) · (` Z)))))))

Twoᶜ : ∀{Φ}{Γ : CtxNf Φ} → Γ ⊢Nf ℕᶜ
Twoᶜ = Succᶜ · (Succᶜ · Zeroᶜ)

Fourᶜ : ∀{Φ}{Γ : CtxNf Φ} → Γ ⊢Nf ℕᶜ
Fourᶜ = Succᶜ · (Succᶜ · (Succᶜ · (Succᶜ · Zeroᶜ)))

TwoPlusTwoᶜ : ∀{Φ}{Γ : CtxNf Φ} → Γ ⊢Nf ℕᶜ
TwoPlusTwoᶜ = Twoᶜ ·⋆ ℕᶜ · Twoᶜ · Succᶜ
```

Using the full facilities of System $F_{\omega\mu}$ we can define natural numbers as Scott Numerals [1]. We the Z combinator instead of the Y combinator as it works for both lazy and strict languages.

```
G : ∀{Φ} → Φ ,⋆ * ⊢Nf⋆ *
G = Π (ne (` Z) ⇒ (ne (` (S Z)) ⇒ ne (` Z)) ⇒ ne (` Z))

M : ∀{Φ} → Φ ⊢Nf⋆ *
M = μ G

N : ∀{Φ} → Φ ⊢Nf⋆ *
N = G [ M ]Nf⋆
```

```
Zero : ∀{Φ}{Γ : CtxNf Φ} → Γ ⊢Nf N
Zero = Λ (ƛ (ƛ (` (S (Z )))))

Succ : ∀{Φ}{Γ : CtxNf Φ} → Γ ⊢Nf N ⇒ N
Succ = ƛ (Λ (ƛ (ƛ (` Z · wrap _ (` (S (S (T Z)))))))))

Two : ∀{Φ}{Γ : CtxNf Φ} → Γ ⊢Nf N
Two = Succ · (Succ · Zero)

Four : ∀{Φ}{Γ : CtxNf Φ} → Γ ⊢Nf N
Four = Succ · (Succ · (Succ · (Succ · Zero)))

case : ∀{Φ}{Γ : CtxNf Φ}
  → Γ ⊢Nf N ⇒ (Π (ne (` Z) ⇒ (N ⇒ ne (` Z)) ⇒ ne (` Z)))
case = ƛ (Λ (ƛ (ƛ (
  (` (S (S (T Z)))) ·* ne (` Z) · (` (S Z)) · (ƛ (` (S Z) · unwrap (` Z))))))))

Z-comb : ∀{Φ}{Γ : CtxNf Φ} →
  Γ ⊢Nf Π (Π (((ne (` (S Z)) ⇒ ne (` Z)) ⇒ ne (` (S Z)) ⇒ ne (` Z))
    ⇒ ne (` (S Z)) ⇒ ne (` Z)))
Z-comb = Λ (Λ (ƛ (ƛ (` (S Z) · ƛ (unwrap (` (S Z)) · ` (S Z) · ` Z))
  · wrap (ne (` Z) ⇒ ne (` (S (S Z))) ⇒ ne (` (S Z)))
        (ƛ (` (S Z) · ƛ (unwrap (` (S Z)) · ` (S Z) · ` Z))))))

Plus : ∀{Φ}{Γ : CtxNf Φ} → Γ ⊢Nf N ⇒ N ⇒ N
Plus = ƛ (ƛ ((Z-comb ·* N) ·* N · (ƛ (ƛ ((((case · ` Z) ·* N)
  · ` (S (S (S Z)))) · (ƛ (Succ · (` (S (S Z)) · ` Z))))) · ` (S Z)))

TwoPlusTwo : ∀{Φ}{Γ : CtxNf Φ} → Γ ⊢Nf N
TwoPlusTwo = (Plus · Two) · Two
```

8 Scaling up from System $F_{\omega\mu}$ to Plutus Core

This formalisation forms the basis of a formalisation of Plutus Core. There are two key extensions.

8.1 Higher kinded recursive types

In this paper we used $\mu : (* \to *) \to *$. This is easy to understand and makes it possible to express simple examples directly. This corresponds to the version of recursive types one might use in ordinary System F. In System F_ω we have a greater degree of freedom. We have settled on an indexed version of $\mu : ((k \to *) \to k \to *) \to k \to *$ that supports the encoding of mutually defined datatypes. This extension is straightforward in iso-recursive types, in equi-recursive it is not. We chose to present the restricted version in this paper as it is simpler and sufficient to present our examples. See the accompanying paper [26] for a more detailed discussion of higher kinded recursive types.

8.2 Integers, bytestrings and cryptographic operations

In Plutus Core we also extend System $F_{\omega\mu}$ with integers and bytestrings and some cryptographic operations such as checking signatures. Before thinking about how to add these features to our language, there is a choice to be made when modelling integers and bytestrings and cryptographic operations in Agda about whether we consider them internal or external to our model. We are modelling the Haskell implementation of Plutus Core which uses the Haskell bytestring library. We chose to model the Plutus Core implementation alone and consider bytestrings as an external black box. We assume (i.e. postulate in Agda) an interface given as a type for bytestrings and various operations such as take, drop, append etc. We can also make clear our expectations of this interface by assuming (postulating) some properties such as that append is associative. Using pragmas in Agda we can ensure that when we compile our Agda program to Haskell these opaque bytestring operations are compiled to the real operations of the Haskell bytestring library. We have taken a slightly different approach with integers as Agda and Haskell have native support for integers and Agda integers are already compiled to Haskell integers by default so we just make use of this builtin support. Arguably this brings integers inside our model. One could also treat integers as a blackbox. We treat cryptographic operations as a blackbox as we do with bytestrings.

To add integers and bytestrings to the System $F_{\omega\mu}$ we add type constants as types and term constants as terms. The type of a term constant is a type constant. This ensures that we can have term variables whose type is type constant but not term constants whose type is a type variable. To support the operations for integers and bytestrings we add a builtin constructor to the term language, signatures for each operation, and a semantics for builtins that applies the appropriate underlying function to its arguments. The underlying function is postulated in Agda and when compiled to Haskell it runs the appropriate native Haskell function or library function. Note that the cryptographic functions are operations on bytestrings.

Adding this functionality did not pose any particular formalisation challenges except for the fact it was quite a lot of work. However, compiling our implementation of builtins to Haskell did trigger several bugs in Agda's GHC backend which were rapidly diagnosed and fixed by the Agda developers.

8.3 Using our implementation for testing

As we can compile our Agda Plutus Core interpreter to Haskell we can test the production Haskell Plutus Core interpreter against it. We make use of the production system's parser and pretty printer which we import as a Haskell library and use the same libraries for bytestrings and cryptographic functions. The parser produces intrinsically typed terms which we scope check and convert to a representation with de Bruijn indices. We cannot currently use the intrinsically typed implementation we describe in this paper directly as we must type check terms first and formalising a type checker is future work. Instead we have implemented a separate extrinsically typed version that we use for testing. After evaluation we convert the de Bruijn syntax back to a named syntax and pretty print the output. We have proven that for any well-typed term the intrinsic and extrinsic versions give the same results after erasure.

Acknowledgements

We thank the anonymous reviewers for their helpful comments and insightful constructive criticism. We thank IOHK for their support of this work. We thank our colleagues Marko Dimjašević, Kenneth MacKenzie, and Michael Peyton Jones for helpful comments on an multiple drafts. The first author would like to James McKinna for spending an afternoon explaining pure type systems and Guillaume Allais, Apostolis Xekoukoulotakis and Ulf Norell for help with diagnosing and fixing bugs that we encountered in Agda's GHC backend in the course of writing this paper.

References

1. Abadi, M., Cardelli, L., Plotkin, G.: Types for the Scott numerals (1993)
2. Allais, G., Chapman, J., McBride, C., McKinna, J.: Type-and-Scope Safe Programs and Their Proofs. In: Bertot, Y., Vafeiadis, V. (eds.) Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2017). pp. 195–207. ACM, New York, NY, USA (2017). https://doi.org/10.1145/3018610.3018613
3. Allais, G., McBride, C., Boutillier, P.: New Equations for Neutral Terms. In: Weirich, S. (ed.) Proceedings of the 2013 ACM SIGPLAN Workshop on Dependently-typed Programming (DTP '13). pp. 13–24. ACM, New York, NY, USA (2013). https://doi.org/10.1145/2502409.2502411
4. Altenkirch, T., Chapman, J., Uustalu, T.: Relative Monads Formalised. Journal of Formalized Reasoning **7**(1), 1–43 (2014). https://doi.org/10.6092/issn.1972-5787/4389
5. Altenkirch, T., Chapman, J., Uustalu, T.: Monads need not be endofunctors. Logical Methods in Computer Science **11**(1), 1–40 (2015). https://doi.org/10.2168/LMCS-11(1:3)2015
6. Altenkirch, T., Reus, B.: Monadic Presentations of Lambda Terms Using Generalized Inductive Types. In: Flum, J., Rodriguez-Artalejo, M. (eds.) Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL. LNCS, vol. 1683, pp. 453–468. Springer-Verlag, Berlin, Heidelberg (1999). https://doi.org/10.1007/3-540-48168-0_32
7. Amani, S., Bégel, M., Bortin, M., Staples, M.: Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. In: Andronick, J., Felty, A. (eds.) Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2018). pp. 66–77. ACM, New York, NY, USA (2018). https://doi.org/10.1145/3167084
8. Aydemir, B.E., Bohannon, A., Fairbairn, M., Foster, J.N., Pierce, B.C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., Zdancewic, S.: Mechanized Metatheory for the Masses: The POPLmark Challenge. In: J., H., T.F., M. (eds.) Proceedings of the International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2015). LNCS, vol. 3603, pp. 50–65. Springer-Verlag, Berlin, Heidelberg (2005). https://doi.org/10.1007/11541868_4
9. Berger, U., Schwichtenberg, H.: An inverse of the evaluation functional for typed lambda-calculus. In: Kahn, G. (ed.) Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91). pp. 203–211. IEEE Computer Society Press (1991). https://doi.org/10.1109/LICS.1991.151645
10. Brown, M., Palsberg, J.: Breaking Through the Normalization Barrier: A Self-Interpreter for F-omega. In: Majumdar, R. (ed.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '16. pp. 5–17. ACM, New York, NY, USA (2016). https://doi.org/10.1145/2837614.2837623

11. Cai, Y., Giarrusso, P.G., Ostermann, K.: System F-omega with Equirecursive Types for Datatype-Generic Programming. In: Majumdar, R. (ed.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16). pp. 30–43. ACM, New York, NY, USA (2016). https://doi.org/10.1145/2837614.2837660
12. Chakravarty, M., Kireev, R., MacKenzie, K., McHale, V., Müller, J., Nemish, A., Nester, C., Peyton Jones, M., Thompson, S., Valentine, R., Wadler, P.: Functional Blockchain Contracts. Tech. rep., IOHK (2019), https://iohk.io/research/papers/#KQL88VAR
13. Chapman, J.: Type checking and normalisation. Ph.D. thesis, University of Nottingham, UK (2009), http://eprints.nottingham.ac.uk/10824/
14. Chapman, J., Kireev, R., Nester, C., Wadler, P.: Literate Agda source of MPC 2019 paper. https://github.com/input-output-hk/plutus/blob/f9f7aef94d9614b67c037337079ad89329889ffa/papers/system-f-in-agda/paper.lagda (2019)
15. Coquand, C.: A Formalised Proof of the Soundness and Completeness of a Simply Typed Lambda-Calculus with Explicit Substitutions. Higher-Order and Symbolic Computation **15**(1), 57–90 (2002). https://doi.org/10.1023/A:1019964114625
16. Danielsson, N.A.: A Formalisation of a Dependently Typed Language as an Inductive-Recursive Family. In: Altenkirch, T., McBride, C. (eds.) Proceedings of Types for Proofs and Programs, International Workshop, (TYPES 2006). LNCS, vol. 4502, pp. 93–109 (2006). https://doi.org/10.1007/978-3-540-74464-1_7
17. van Doorn, F., Geuvers, H., Wiedijk, F.: Explicit convertibility proofs in pure type systems. In: Proceedings of the Eighth ACM SIGPLAN International Workshop on Logical Frameworks & Meta-languages: Theory & Practice, (LFMTP 2013). pp. 25–36. ACM, New York, NY, USA (2013). https://doi.org/10.1145/2503887.2503890
18. Dreyer, D.: Understanding and Evolving the ML Module System. Ph.D. thesis, Carnegie Mellon University (2005)
19. Dreyer, D.: A Type System for Recursive Modules. In: Ramsey, N. (ed.) Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07). pp. 289–302. ACM, New York, NY, USA (2007). https://doi.org/10.1145/1291220.1291196
20. Dybjer, P.: A General Formulation of Simultaneous Inductive-Recursive Definitions in Type Theory. The Journal of Symbolic Logic **65**(2), 525–549 (2000), http://www.jstor.org/stable/2586554
21. Grishchenko, I., Maffei, M., Schneidewind, C.: A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In: Bauer, L., Küsters, R. (eds.) Principles of Security and Trust (POST 2018). LNCS, vol. 10804, pp. 243–269. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89722-6_10
22. Harz, D., Knottenbelt, W.J.: Towards Safer Smart Contracts: A Survey of Languages and Verification Methods. https://arxiv.org/abs/1809.09805 (2018)
23. Hindley, J.R., Seldin, J.P.: Lambda-Calculus and Combinators: An Introduction. Cambridge University Press (2008)
24. Hirai, Y.: Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In: Bracciali, A., Sala, M. (eds.) Proceedings of the 1st Workshop on Trusted Smart Contracts, International Conference on Financial Cryptography and Data Security (WTSC '17). LNCS, vol. 10323. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70278-0_33
25. Jung, A., Tiuryn, J.: A New Characterization of Lambda Definability. In: Proceedings of the International Conference on Typed Lambda Calculi and Applications, (TLCA '93). LNCS, vol. 664, pp. 245–257. Springer, Berlin, Heidelberg (1993). https://doi.org/10.1007/BFb0037110

26. Kireev, R., Nester, C., Peyton Jones, M., Wadler, P., Gkoumas, V., MacKenzie, K.: Unraveling recursion: compiling an IR with recursion to System F. In: Hutton, G. (ed.) Proceedings of Mathematics of Program Construction - 13th International Conference (MPC 2019) (2019). https://doi.org/10.1007/978-3-030-33636-3_15
27. Kovács, A.: System F Omega, https://github.com/AndrasKovacs/system-f-omega
28. Martens, C., Crary, K.: LF in LF: Mechanizing the Metatheories of LF in Twelf. In: Proceedings of the Seventh International Workshop on Logical Frameworks and Meta-languages, Theory and Practice (LFMTP '12). pp. 23–32. ACM, New York, NY, USA (2012). https://doi.org/10.1145/2364406.2364410
29. McBride, C.: Datatypes of Datatypes. In: Summer School on Generic and Effectful Programming, St Anne's College, Oxford, Lecture Notes (2015), https://www.cs.ox.ac.uk/projects/utgp/school/conor.pdf
30. Nomadic Labs: Michelson in Coq. Git Repository, https://gitlab.com/nomadic-labs/mi-cho-coq/
31. O'Connor, R.: Simplicity: A New Language for Blockchains. In: Bielova, N., Gaboardi, M. (eds.) Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security (PLAS '17). pp. 107–120. ACM, New York, NY, USA (2017). https://doi.org/10.1145/3139337.3139340
32. Park, D., Zhang, Y., Saxena, M., Daian, P., Roşu, G.: A Formal Verification Tool for Ethereum VM Bytecode. In: Garcia, A., Păsăreanu, C.S. (eds.) Proceedings of the 2018 26th ACM Join Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018). pp. 912–915. ACM, New York, NY, USA (2018). https://doi.org/10.1145/3236024.3264591
33. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
34. Pollack, R., Poll, E.: Typechecking in Pure Type Systems. In: Informal proceedings of Logical Frameworks '92. pp. 271–288 (1992)
35. Reynolds, J.C.: What Do Types Mean? – from Intrinsic to Extrinsic Semantics. In: McIver, A., Morgan, C. (eds.) Programming Methodology, pp. 309–327. Monographs in Computer Science, Springer, New York, NY (2003). https://doi.org/10.1007/978-0-387-21798-7_15
36. Wadler, P.: Programming Language Foundations in Agda. In: Massoni, T., Mousavi, M. (eds.) Formal Methods: Foundations and Applications. SBMF 2018. LNCS, vol. 11254, pp. 56–73. Springer, Cham (2018). https://doi.org/0.1007/978-3-030-03044-5_5
37. Wadler, P., Kokke, W.: Programming Language Foundations in Agda. https://plfa.github.io/
38. Watkins, K., Cervesato, I., Pfenning, F., Walker, D.: Specifying Properties of Concurrent Computations in CLF. In: C., S. (ed.) Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM 2004). ENTCS, vol. 199, pp. 67–87 (2008). https://doi.org/10.1016/j.entcs.2007.11.013