Scoped and Typed Staging by Evaluation

Guillaume Allais
guillaume.allais@strath.ac.uk
University of Strathclyde
Glasgow, United Kingdom

Abstract

Using a dependently typed host language, we give a well scoped-and-typed by construction presentation of a minimal two level simply typed calculus with a static and a dynamic stage. The staging function partially evaluating the part of a term that are static is obtained by a model construction inspired by normalisation by evaluation.

We then go on to demonstrate how this minimal language can be extended to provide additional metaprogramming capabilities, and to define a higher order functional language evaluating to digital circuit descriptions.

Keywords: Staging, Dependent Types, Two Level Type Theory, Normalisation by Evaluation, Hardware Descriptions , Agda

ACM Reference Format:
Guillaume Allais. 2024. Scoped and Typed Staging by Evaluation. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '24).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Staged compilation, by running arbitrary programs at compile time in order to generate code, is a way to offer users metaprogramming facilities. Kovács demonstrated that the notion of two level type theories, originally introduced in homotopy theory, can be repurposed to describe layered languages equipped with a staging operation partially evaluating the terms in the upper layer [18].

In order to enable the mechanised study of such systems, we give an intrinsically scoped-and-typed treatment of various two level simply typed calculi and their corresponding staging operations evaluating away all of the static subterms. We obtain these staging operations by performing type-directed model constructions reminiscent of the ones used for normalisation by evaluation, hence the title of this paper.

PEPM '24, January 16, 2024, London, UK

ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00
https://doi.org/XXXXXXX.XXXXXXX

This culminates in a system that takes seriously Kovács' remark that the static and dynamic layers do not need to have exactly the same features. Its static layer is a higher order functional language while the dynamic one corresponds to digital circuit descriptions. This casts existing work on high level languages for quantum circuit descriptions into a new light as two level theories.

This work has been fully formalised using Agda [21] as our host language (but any implementation of Martin-Löf type theory [19] with inductive families [14] would do).

2 A Primer on Intrinsically Typed λ-Calculi

Let us start with a quick primer on intrinsically scoped-and-typed λ-calculi defined in a dependently typed host language. The interested reader can refer to ACMM [3] for a more in-depth presentation of this approach.

2.1 Object Types and Contexts

We first give an inductive definition of object types. We call it Type and its own type is Set, the type of all small types in Agda. It has two constructors presented in generalised algebraic datatype fashion. We use ‘α as our base type, and (A ‘⇒ B) is the type of functions from A to B.

```
data Type : Set where
  ‘α      : Type
  _‘⇒_ : (A B : Type) → Type
```

Agda-ism (Syntax Highlighting)**.** *All of the code snippets in this paper are semantically highlighed: keywords are orange, definitions and types are blue, data constructors are green, bound variables are slanted, and comments are brown.*

Agda-ism (Implicit Prenex Polymorphism)**.** *We extensively use Agda's variable mechanism: all of the seemingly unbound names will in fact have been automatically quantified over in a prenex position provided that they have been declared beforehand.*

The following block for instance announces that from now on unbound As, Bs, and Cs stand for implicitly bound Type variables.

```
variable A B C : Type
```

Next, we form contexts as left-nested lists of types using constructor names similar to the ones typically used in type judgments. Contexts may be the empty context ε or a compound context (Γ , A) obtained by extending an existing context Γ on the right with a newly bound (nameless) variable of type A.

```
data Context : Set where
  ε   : Context
  _,_ : Context → Type → Context

variable Γ Δ Θ : Context
variable P Q : Context → Set
```

2.2 Manipulating Indexed Types

In this paper we are going to conform to the convention of only mentioning context *extensions* when presenting judgements. That is to say we will write the application and λ-abstraction rules as they are in the right column rather than the left one where the ambient context Γ is explicitly threaded.

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash t : A}{\Gamma \vdash f\ t : B} \qquad\qquad \frac{f : A \to B \qquad t : A}{f\ t : B}$$

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x.b : A \to B} \qquad\qquad \frac{x : A \vdash b : B}{\lambda x.b : A \to B}$$

To do so, we need to introduce a small set of combinators to manipulate indexed definitions. These are commonplace and already present in Agda's standard library. First, ∀[_] universally quantifies over its argument's index; it is meant to be used to surround a complex expression built up using the other combinators.

```
∀[_] : (I → Set) → Set
∀[ P ] = ∀ {i} → P i
```

Second, the suggestively named _⊢_ allows us to modify the index; it will be useful to extend a context with freshly bound variables.

```
_⊢_ : (I → J) → (J → Set) → (I → Set)
(f ⊢ P) i = P (f i)
```

Third, we can form index-respecting function spaces.

```
_⇒_ : (P Q : I → Set) → (I → Set)
(P ⇒ Q) i = P i → Q i
```

Finally, the pointwise lifting of pairing is called _∩_; it will only come into play in Section 5.2.

```
_∩_ : (P Q : I → Set) → (I → Set)
(P ∩ Q) i = P i × Q i
```

We include below an artificial example of a type written using the combinators together with its full expansion using explicit context-passing.

```
∀[ (_, A) ⊢ (P ∩ Q ⇒ Q ∩ P) ]
∀ {Γ} → (P (Γ , A) × Q (Γ , A)) → (Q (Γ , A) × P (Γ , A))
```

2.3 Intrinsically Typed Variables

Our first inductive family [14] Var formalises what it means for a variable of type A to be present in context Γ. It is indexed over said type and context. We present it side by side with the corresponding inference rules for the typing judgement for variables denoted $(\cdot :_v \cdot)$. The first constructor (here) states that in a non-empty context where the most local variable has type A we can indeed obtain a variable of type A. The second one (there) states that if a variable of type A is present in a context then it also is present in the same context extended with a freshly bound variable of type B.

```
data Var : Type → Context → Set where
  here  : ∀[         (_, A) ⊢ Var A ]
  there : ∀[ Var A ⇒ (_, B) ⊢ Var A ]
```

$$\frac{}{x : A \vdash x :_v A} \qquad\qquad \frac{x :_v A}{y : B \vdash x :_v A}$$

This is a standard definition corresponding to a scoped-and-typed variant of De Bruijn indices [5, 6, 8, 13]: here corresponds to zero, and there to successor.

2.4 Intrinsically Typed Terms

We are now ready to give the type of intrinsically typed terms. It is once again an inductive family indexed over a type and a context; its declaration is as follows.

```
data Term : Type → Context → Set where
```

We will introduce constructors in turn, each paralleled by its counterpart as an inference rule. We start with the variable rule: a variable of type A forms a valid term of type A. As you can see below, we use a line lexed as a comment (----) to suggestively typeset the constructor's type like the corresponding rule.

```
'var : ∀[  Var A ⇒
          --------
           Term A ]
```

$$\frac{x :_v A}{x : A}$$

Next we have the constructor for applications. It states that by combining a term whose type is a function type from A to B and a term of type A, we obtain a term of type B.

```
'app : ∀[  Term (A '⇒ B) ⇒ Term A ⇒
          ------------------------
           Term B ]
```

$$\frac{f : A \to B \qquad t : A}{f\ t : B}$$

Last but not least, the rule for λ-abstraction is the only rule with a premise mentioning a context extension. It states that we can build a term for a function from A to B by building the function's body of type B in a context extended by a freshly bound variable of type A.

```
'lam : ∀[  (_, A) ⊢ Term B ⇒
        - - - - - - - - - - - - - - - -
           Term (A '⇒ B) ]
```

$$\frac{x : A \vdash b : B}{\lambda x.b : A \to B}$$

Putting it all together, we obtain the following inductive family.

```
data Term : Type → Context → Set where
  'var : ∀[ Var A ⇒ Term A ]
  'app : ∀[ Term (A '⇒ B) ⇒ Term A ⇒ Term B ]
  'lam : ∀[ (_, A) ⊢ Term B ⇒ Term (A '⇒ B) ]
```

This gives us the intrinsically scoped-and-typed syntax for the simply typed lambda calculus. And we give our first example: the identity function ($\lambda.0$ in de Bruijn notation).

```
'id : ∀[ Term (A '⇒ A) ]
'id = 'lam ('var here)
```

As any well behaved syntax should, it is stable under weakening as we are going to see shortly.

2.5 Weakening

Following Altenkirch, Hofmann, and Streicher [4] we start by defining the category of weakenings with contexts as objects and the following inductive family as morphims.

```
data _≤_ : Context → Context → Set where
  done : ε ≤ ε
  keep : Γ ≤ Δ → Γ , A ≤ Δ , A
  drop : Γ ≤ Δ → Γ     ≤ Δ , A
```

This relation on contexts, also known as order-preserving embeddings in the literature, is a first order description of order-preserving injections: done is the trivial injection of the empty context into itself; keep extends an existing injection into one that preserves the most local variable; and drop records that the most local variable of the target context does not have a pre-image via the injection.

We can define identity and composition of these morphisms (we leave the definitions out but they are available in the accompanying material).

```
≤-refl : Γ ≤ Γ              ≤-trans : Γ ≤ Δ → Δ ≤ Θ → Γ ≤ Θ
```

These order-preserving embeddings all have an action on suitably well behaved scoped families. We will call these actions weakening principles, and introduce the following type synonym to describe them.

```
Weaken : (Context → Set) → Set
Weaken P = ∀ {Γ Δ} → Γ ≤ Δ → P Γ → P Δ
```

The action on variables is given by the following wkVar definition. It is defined by induction over the renaming and case analysis on the de Bruijn index if the most local variable happens to be in both contexts.

```
wkVar : Weaken (Var A)
wkVar (drop σ) v          = there (wkVar σ v)
wkVar (keep σ) here       = here
wkVar (keep σ) (there v) = there (wkVar σ v)
```

The action on terms is purely structural, with the caveat that the weakening needs to be amended when going under a binder: the most recently bound variable is present in both the source and target contexts and so we use keep to mark it as retained.

```
wkTerm : Weaken (Term A)
wkTerm σ ('var v)    = 'var (wkVar σ v)
wkTerm σ ('app f t) = 'app (wkTerm σ f) (wkTerm σ t)
wkTerm σ ('lam b)   = 'lam (wkTerm (keep σ) b)
```

Using these results, we can define function composition as a pseudo constructor: provided g and f, we form $\lambda x.g\ f\ x$ i.e. we use g and f in a context extended with x hence the need for weakening.

```
_'∘_ : ∀[ Term (B '⇒ C) ⇒ Term (A '⇒ B) ⇒ Term (A '⇒ C) ]
g '∘ f = let Γ≤Γ,A = drop ≤-refl in
        'lam ('app (wkTerm Γ≤Γ,A g)
                   ('app (wkTerm Γ≤Γ,A f) ('var here)))
```

Agda-ism (Lexing of Identifiers)**.** *Ignoring details about reserved characters for now: any space-free string of unicode characters is considered a single identifier. Correspondingly, in the example above* Γ≤Γ,A *is a single identifier named like this to document for the human reader what its type looks like.*

2.6 Normalisation by Evaluation

It is now time to define an evaluation function for this syntax. By the end of this section, we will have a function eval turning terms into Kripke-style values, provided that we have an environment assigning values to each of the term's free variables. It will have the following type.

```
eval : Env Γ Δ → Term A Γ → Value A Δ
```

2.6.1 Kripke Function Spaces. This whole process is based on Kripke semantics for intuitionistic logic [20]. As a consequence one of the central concepts is closure under future worlds, here context extensions. This idea is captured by the definition of the □ record: we can inhabit (□ A Γ) whenever for any extension Δ of Γ we are able to construct an (A Δ).

```
record □ (A : Context → Set) (Γ : Context) : Set where
  constructor mk□
  field run□ : ∀[ (Γ ≤_) ⇒ A ]
```

For more information on □ and its properties, see Allais, Atkey, Chapman, McBride, and McKinna [2, Section 3.1]. We will only use the fact that it is a comonad, that is to say that we can define extract and duplicate thanks to the fact that the embedding relation is a preorder.

```
extract : ∀[ □ P ⇒ P ]
extract p = p .run□ ≤-refl
```

```
duplicate : ∀[ □ P ⇒ □ (□ P) ]
duplicate p .run□ σ .run□ = p .run□ ∘ ≤-trans σ
```

Agda-ism (Copattern matching)**.** *The definition of* duplicate *proceeds by copattern-matching [1]. This allows us to define values of a record type (here* □*) by defining the result of taking each of its projections (here the unique* run□ *one). In this instance it is particularly useful because we have a type involving nested records and each projection takes additional arguments: an (implicit) context and a weakening into that context.*

Kripke function spaces then correspond to functions inside a box, hence the following definition.

```
Kripke : (P Q : Context → Set) → (Context → Set)
Kripke P Q = □ (P ⇒ Q)
```

The comonadic structure of □ additionally ensures we can define semantic application (_$$_) and weakening of Kripke function spaces.

```
_$$_ : ∀[ Kripke P Q ⇒ P ⇒ Q ]
_$$_ = extract

wkKripke : Weaken (Kripke P Q)
wkKripke σ f = duplicate f .run□ σ
```

Finally, we introduce a notation to hide away □-related notions when building Kripke functions. After the following declarations we can write λλ[σ , v] b to implement a function of type (Kripke A B Γ).

```
syntax mk□ (λ σ x → b) = λλ[ σ , x ] b
```

Agda-ism (Syntax Declarations)**.** *A* syntax *declaration introduces syntactic sugar that is allowed to perform variable binding, or take arguments in a seemingly out-of-order manner. In the above declaration the left hand side describes the actual term and the right hand side its newly introduced sugared form.*

We now have all of the ingredients necessary to perform the model construction allowing us to implement a normaliser.

2.6.2 Model Construction. This step follows standard techniques for normalisation by evaluation [7, 10, 11]. The family of values is defined by induction on the value's type. Values of a base type are neutral terms (this is not enforced here and we are happy to simply reuse Term) while values of a function type are Kripke function spaces between values of the domain and values of the codomain.

```
Value : Type → Context → Set
Value ‘α        = Term ‘α
Value (A ‘⇒ B) = Kripke (Value A) (Value B)
```

We prove that values can be weakened by using the fact they are defined in terms of families already known to be amenable to weakenings.

```
wkValue : (A : Type) → Weaken (Value A)
wkValue ‘α        σ v = wkTerm σ v
wkValue (A ‘⇒ B) σ v = wkKripke σ v
```

Environments are functions associating a Value to each Variable in scope.

```
record Env (Γ Δ : Context) : Set where
  field get : ∀ {A} → Var A Γ → Value A Δ
```

In the upcoming definition of the evaluation function, environments will in general simply be threaded through. They will only need to be modified when going under a binder. This binder, interpreted as a Kripke function space, will provide a context weakening and a value living in that context. The environment will have to be extended with the value while its existing content will need to be transported, along the weakening, into the bigger context. The extend definition combines these two operations into a single one. It is defined in copattern style: .run□ builds a box while .get builds the returned environment. The definition proceeds by case analysis on the variable to be mapped to a value: if it is the newly bound one, we immediately return the value we just obtained, and otherwise we look up the associated value in the old environment and use σ to appropriately weaken it.

```
extend : ∀[ Env Γ ⇒ □ (Value A ⇒ Env (Γ , A)) ]
extend ρ .run□ σ v .get here      = v
extend ρ .run□ σ v .get (there x) = wkValue _ σ (ρ .get x)
```

The evaluation function maps terms to values provided that an environment assigns a value to every free variable in scope. It is defined by induction on the term and maps every construct to its semantical counterpart: variables become environment lookups, application become Kripke applications, and λ-abstractions become Kripke functions.

```
eval : Env Γ Δ → Term A Γ → Value A Δ
eval ρ (‘var v)    = ρ .get v
eval ρ (‘app f t) = eval ρ f $$ eval ρ t
eval ρ (‘lam b)    = λλ[ σ , v ] eval (extend ρ .run□ σ v) b
```

A typical normalisation by evaluation presentation would conclude with the definition of a reification function extracting a term from a value in a type-directed manner before defining normalisation as the composition of evaluation and reification. This last step will however not be useful for our study of two level calculi and so we leave it out. It can be found in details in Catarina Coquand's work on normalisation by evaluation for a simply typed λ-calculus with explicit substitutions [10].

Now that we have seen how to define a small well scoped-and-typed language and construct an evaluation function by performing a model construction, we can start looking at a extending it to two level language.

3 Minimal Intrinsically Typed Two Level Type Theory

We start with the smallest two level calculus we can possibly define by extending the simply typed λ-calculus as defined in the previous section with quotes (‘⟨_⟩) and splices (‘∼_).

This will enable us to write and stage simple programs such as the following.

$$`\alpha \; `\!\Rightarrow \; `\alpha \ni `\mathsf{app}\; `\mathsf{id}^d \; (`\!\sim `\mathsf{app}\; `\mathsf{id}^s \; `\langle `\mathsf{id}^d \rangle) \rightsquigarrow `\mathsf{app}\; `\mathsf{id}^d \; `\mathsf{id}^d$$

The three-place relation ($A \ni s \rightsquigarrow t$) states that staging a term s at type A yields the term t. Here, ‘id^d is a dynamic identity function while ‘id^s is a static one, ‘⟨_⟩ quotes a static term inside a dynamic one, and ‘∼_ splices a dynamic term in a static one. Correspondingly, staging will partially evaluate the call to ‘id^s as well as all the quotes and splices while leaving the rest of the term intact. Hence the result: the call to the static identity function has fully reduced but the call to the dynamic one has been preserved.

3.1 Phases, Stages, and Types

We start by defining a sum type of phases denoting whether we are currently writing src code or inspecting stg code that has already been partially evaluated.

```
data Phase : Set where
  src stg : Phase

variable ph : Phase
```

Additionally, our notion of types is going to be explicitly indexed by the stage they live in. These stages are themselves indexed over the phase they are allowed to appear in. The static (sta) stage is only available in the src phase: once code has been staged, all of its static parts will be gone. The dynamic (dyn) stage however will be available in both phases, hence the unconstrained index ph.

```
data Stage : Phase → Set where
  sta : Stage src
  dyn : Stage ph

variable st : Stage ph
```

We can now define our inductive family of simple types indexed by their stage.

```
data Type : Stage ph → Set where
  ‘α    : Type st
  ‘⇑_   : Type {src} dyn → Type sta
  _‘⇒_  : (A B : Type st) → Type st

variable A B C : Type st
```

We have both static and dynamic terms of base type, hence the unconstrained indices ph and st for the constructor ‘α. The constructor ‘⇑_ allows us to embed dynamic types into static ones; (‘⇑ A) is effectively the type of *programs* that will compute a value of type A at runtime. This is only available in the src phase. Function types are available in both layers provided that they are homogeneous: both the domain and codomain need to live in the same layer.

Purely dynamic types in the source phase have a direct counterpart in the staged one. We demonstrate this by implementing the following asStaged function.

```
asStaged : Type {src} dyn → Type {stg} dyn
asStaged ‘α         = ‘α
asStaged (A ‘⇒ B) = asStaged A ‘⇒ asStaged B
```

It is essentially the identity function except for the fact that its domain and codomain have different indices.

3.2 Intrinsically Scoped and Typed Syntax

We skip over the definition of contexts and variables: they are essentially the same as the ones we gave in Section 2.

Our type of term is indexed by a phase, a stage, a type at that stage, and a context.

```
data Term : (ph : Phase) (st : Stage ph) →
            Type st → Context → Set where
```

The first constructors are familiar: they are exactly the ones seen in the previous section. These constructs are available at both levels and both before and after staging hence the fact that the phase and stage indices are polymorphic here.

```
‘var  : ∀[ Var A ⇒ Term ph st A ]
‘app : ∀[ Term ph st (A ‘⇒ B) ⇒ Term ph st A ⇒ Term ph st B ]
‘lam  : ∀[ (_, A) ⊢ Term ph st B ⇒ Term ph st (A ‘⇒ B) ]
```

Next we have the constructs specific to the two level calculus: quotes (‘⟨_⟩) let users insert dynamic terms into static expressions while splices (‘∼_) allow static terms to be inserted in dynamic ones. Staging will, by definition, eliminate these and so their phase index is constrained to be src.

```
‘⟨_⟩ : ∀[ Term src dyn A ⇒ Term src sta (‘⇑ A) ]
‘∼_ : ∀[ Term src sta (‘⇑ A) ⇒ Term src dyn A ]
```

Putting it all together, we obtain the following inductive family representing a minimal intrinsically typed two-level calculus.

```
data Term : (ph : Phase) (st : Stage ph) →
            Type st → Context → Set where
  ‘var  : ∀[ Var A ⇒ Term ph st A ]
  ‘app : ∀[ Term ph st (A ‘⇒ B) ⇒ Term ph st A ⇒ Term ph st B ]
  ‘lam  : ∀[ (_, A) ⊢ Term ph st B ⇒ Term ph st (A ‘⇒ B) ]
  ‘⟨_⟩  : ∀[ Term src dyn A ⇒ Term src sta (‘⇑ A) ]
  ‘∼_   : ∀[ Term src sta (‘⇑ A) ⇒ Term src dyn A ]
```

We can readily write examples such as the following definitions of a purely dynamic and a purely static identity function. The dynamic function will survive staging even if it is applied to a dynamic argument while the static one can only exist in the source phase and will be fully evaluated during staging.

```
‘id^d : ∀[ Term ph dyn (A ‘⇒ A) ]    ‘id^s : ∀[ Term src sta (A ‘⇒ A) ]
‘id^d = ‘lam (‘var here)              ‘id^s = ‘lam (‘var here)
```

Now that we have a syntax, we can start building the machinery that will actually perform its partial evaluation.

4 Staging by Evaluation

The goal of this section is to define a type of Values as well as an evaluation function which computes the value associated to each term, provided that we have an appropriate environment to interpret the term's free variables. This will once again yield a function eval of the following type.

```
eval : Env Γ Δ → Term src st A Γ → Value st A Δ
```

As a corollary we will obtain a staging function that takes a closed dynamic term and gets rid of all of the quotes and splices by fully evaluating all of its static parts.

```
stage : Term src dyn A ε → Term stg dyn (asStaged A) ε
```

We start with the model construction describing precisely the type of values.

4.1 Model Construction

The type of values is defined by case analysis on the stage. Static values are given a static meaning (defined below) while dynamic values are given a meaning as staged terms i.e. terms guaranteed not to contain any static subterm.

```
Value : (st : Stage src) → Type st → Context → Set
Value sta  = Static
Value dyn = Term stg dyn ∘ asStaged
```

The family of static values is defined by induction on the value's type. It is fairly similar to the standard normalisation by evaluation construction except that static values at a base types cannot possibly be neutral terms.

```
Static : Type sta → Context → Set
Static ‘α          = const ⊥
Static (‘⇑ A)      = Value dyn A
Static (A ‘⇒ B) = Kripke (Static A) (Static B)
```

There are no static values of type ‘α as this base type does not have any associated constructors and so we return the empty type ⊥; values of type (‘⇑ A) are dynamic values of type A i.e. staged terms of type A; functions from A to B are interpreted using Kripke function spaces from static values of type A to static values of type B.

4.2 Evaluation

We can now explain what the meaning of each term constructor is. In every instance we will proceed by case analysis on the stage the meaning is being used at, essentially using a meaning inspired by normalisation by evaluation for the static part and one inspired by substitution for the dynamic one.

Application is interpreted as the semantic application defined for Kripke function spaces in the static case, and the syntactic ‘app constructor in the dynamic one.

```
app : (st : Stage src) {A B : Type st} →
      Value st (A ‘⇒ B) Γ → Value st A Γ → Value st B Γ
app sta  = _$$_
app dyn = ‘app
```

Lambda-abstraction are mapped to Kripke λs for static values and to syntactic ones for the dynamic ones.

```
lam : (st : Stage src) {A B : Type st} →
      Kripke (Value st A) (Value st B) Γ →
      Value st (A ‘⇒ B) Γ
lam sta  b = λλ[ σ , v ] b .run□ σ v
lam dyn b = ‘lam (b .run□ (drop ≤-refl) (‘var here))
```

Putting it all together, we obtain the following definition of the evaluation function. Note that by virtue of the model construction the interpretation of both ‘∼_ and ‘⟨_⟩ is the identity function: static values of type (‘⇑ A) and staged terms of type A are interchangeable.

```
eval : Env Γ Δ → Term src st A Γ → Value st A Δ
eval ρ (‘var v)             = ρ .get v
eval ρ (‘app {st = st} f t) = app st (eval ρ f) (eval ρ t)
eval ρ (‘lam {st = st} b)   = lam st (body ρ b)
eval ρ ‘⟨ t ⟩               = eval ρ t
eval ρ (‘∼ v)               = eval ρ v
```

The function eval is mutually defined with an auxiliary function describing its behaviour on the body of a λ-abstraction. It is defined using semantics lambdas and extend.

```
body : Env Γ Δ → Term src st B (Γ , A) →
       Kripke (Value st A) (Value st B) Δ
body ρ b = λλ[ σ , v ] eval (extend ρ .run□ σ v) b
```

We finally obtain the stage function by calling eval with an empty environment.

```
stage : Term src dyn A ε → Term stg dyn (asStaged A) ε
stage = eval (λ where .get ())
```

Agda-ism ((Co)Pattern-Matching Lambda)**.** *The keyword (λ where) is analogous to Haskell's* `\case`*: it introduces a pattern-matching lambda. In this instance, it is a copattern-matching one: we define the environment of type (Env ε ε) by copattern-matching on .get which allows us to bind an argument of type (Var A ε) that can in turn be immediately dismissed as uninhabited using the empty pattern ().*

5 A More Practical Two Level Calculus

We are now going to extend the minimal calculus we used so far to show a more realistic example of a two level calculus.

First we are going to add natural numbers and their eliminator. These will be available at both stages and we will see how we can transfer a static natural number to the dynamic phase by defining a static ‘reify term.

Second, based on Kovács' observation that the static and dynamic language do not need to have exactly the same features, we are going to add a type of static pairs. These pairs and their projections can be used in arbitrary static code but will be guaranteed to be evaluated away during staging. We will demonstrate this by giving a static term 'fib implementing a standard linear (ignoring the cost of addition) algorithm for the Fibonacci function. This will allow us to obtain e.g.

```
'ℕ ∋  'app ('app 'add 'zero) ('~ 'app ('reify '∘ 'fib) (fromℕ 8))
  ⇝ 'app ('app 'add 'zero) (fromℕ 21)
```

where fromℕ is a helper function turning Agda literals into Terms built using 'zero and 'succ, and 'add is a dynamic addition function. Note that the dynamic call to addition was not evaluated away during staging.

5.1 Adding natural numbers

Our first extension adds the inductive type of Peano-style natural numbers, its two constructors, and the appropriate eliminator for it.

5.1.1 Types and Terms. First we extend the definition of Type with a new constructor 'ℕ. Natural numbers will be present at both stages and so we allow the index to be polymorphic.

```
'ℕ : Type st
```

We then add Term constructors for the two Peano-style constructors ('zero and 'succ) as well as an eliminator ('iter) which turns a natural number into its Church encoding [9, Chapter 3].

```
'zero : ∀[ Term ph st 'ℕ ]
'succ : ∀[ Term ph st 'ℕ ⇒ Term ph st 'ℕ ]
'iter  : ∀[ Term ph st ('ℕ '⇒ (A '⇒ A) '⇒ A '⇒ A) ]
```

Our first program example is the function 'reify that turns its static natural number argument into a dynamic encoding. It does so by iterating over its input and replacing static 'zeros and 'succs by dynamic ones.

```
'reify : ∀[ Term src sta ('ℕ '⇒ '⇑ 'ℕ) ]
'reify = 'lam ('app ('app ('app 'iter ('var here))
  ('lam '⟨ 'succ ('~ 'var here) ⟩))
  '⟨ 'zero ⟩)
```

We can also naturally define addition as iterated calls to 'succ. This definition is valid at both stages hence the polymorphic phase and stage indices.

```
'add : ∀[ Term ph st ('ℕ '⇒ 'ℕ '⇒ 'ℕ) ]
'add = 'lam ('app ('app 'iter ('var here)) ('lam ('succ ('var here))))
```

Let us now see how to evaluate the newly added constructs.

5.1.2 Staging by Evaluation. We extend the definition of Static with a new clause decreeing that values of type 'ℕ are constant natural numbers.

```
Static 'ℕ = const ℕ
```

We can then describe the semantical counterparts of the newly added constructors. The term constructor 'zero is either interpreted by the natural number 0 or by the term constructor itself depending on whether it is used in a static or dynamic manner.

```
zero : (st : Stage src) → Value st 'ℕ Γ
zero sta  = 0
zero dyn = 'zero
```

Similarly 'succ is interpreted either as (1 +_) if it used in a static manner or by the term constructor itself for dynamic uses.

```
succ : (st : Stage src) → Value st 'ℕ Γ → Value st 'ℕ Γ
succ sta  = 1 +_
succ dyn = 'succ
```

The meaning of 'iter in the static layer is defined in terms of the iterate function defined by pattern-matching in the host language and turning a natural number into its Church encoding. Note that we need to use wkKripke to bring the *succ* argument into the wider scope the *zero* one lives int.

```
iterate : {ty : Set} → (ty → ty) → ty → ℕ → ty
iterate s z 0        = z
iterate s z (suc n) = s (iterate s z n)
```

```
iter : ∀ st {A} → Value st ('ℕ '⇒ (A '⇒ A) '⇒ (A '⇒ A)) Γ
iter dyn = 'iter
iter sta  = λλ[ _ , m ] λλ[ _ , succ ] λλ[ σ , zero ]
              iterate (weak-Kripke σ succ $$_) zero m
```

We can readily compute with these numbers. Reifying the static result obtained by adding 7 to 35 will for instance return 42 (here fromℕ once again stands for a helper function turning Agda literals into Term numbers).

```
'ℕ ∋  '~ 'app 'reify ('app ('app 'add (fromℕ 7)) (fromℕ 35))
  ⇝ fromℕ 42
```

Let us now look at an example of the fact, highlighted in Kovács' original paper, that static datatypes do not need to have a counterpart at runtime.

5.2 Adding static pairs

We now want to add pairs that are only available in the static layer and ensure that all traces of pairs and their projections will have completely disappeared after staging.

5.2.1 Types and Terms. We first extend the inductive definition of object types with a new construct for pair types. It is explicitly marked as sta only.

```
_'×_ : (A B : Type sta) → Type sta
```

We then extend the inductive family of term constructs with a constructor for pairs (_',_) and two constructors for the first ('fst) and second ('snd) projection respectively.

```
_',_  : ∀[ Term src sta A ⇒ Term src sta B ⇒
          Term src sta (A '× B) ]
'fst   : ∀[ Term src sta ((A '× B) '⇒ A) ]
'snd : ∀[ Term src sta ((A '× B) '⇒ B) ]
```

This enables us to implement in the static layer the classic linear definition of the Fibonacci function which internally uses a pair of the current Fibonacci number and its successor.

It is obtained by taking the first projection of the result of iterating the invariant-respecting *step* function over the valid *base* case.

```
'fib : Term src sta ('ℕ '⇒ 'ℕ) ε
'fib = 'fst '∘ 'lam ('app ('app ('app
  -- this implements n ↦ (fib n, fib (1 + n))
  'iter ('var here))
      {- step -} ('lam
        let fibₙ   = 'app 'fst ('var here)
            fib1ₙ = 'app 'snd ('var here)
            fib2ₙ = 'app ('app 'add fibₙ) fib1ₙ
        in (fib1ₙ ', fib2ₙ)))
      {- base -} ('zero ', 'succ 'zero))
```

This definition uses _'∘_ defined in Section 2.5, and 'add defined in Section 5.1.1.

5.2.2 Staging by Evaluation. The amendment to the model construction and the definition of the constructors' semantical counterparts is easy. First, static pairs are pairs of static values.

```
Static (A '× B) = Static A ∩ Static B
```

Second, pair constructors are mapped to pair constructors in the host language, and the same for projections.

```
eval ρ (s ', t) = eval ρ s , eval ρ t
eval ρ 'fst     = λλ[ _ , v ] Prod.proj₁ v
eval ρ 'snd     = λλ[ _ , v ] Prod.proj₂ v
```

These definition now allow us to evaluate static calls to the Fibonacci function such as the one presented in this section's introduction.

```
'ℕ ∋  'app ('app 'add 'zero) ('~ 'app ('reify '∘ 'fib) (fromℕ 8))
   ↝ 'app ('app 'add 'zero) (fromℕ 21)
```

While this addition of static pairs may seem interesting but anecdotal, the same techniques can be used to work on defining a much more applicable two level language.

6 Application: Circuit Generation

This section's content is inspired by Quipper, a functional programming language to describe quantum computations introduced by Green, Lumsdaine, Ross, Selinger, and Valiron [16] and related formal treatments such as Rennela and Staton's categorical models [23]. This strand of research gives us a good example of a setting in which we have two very distinct layers: a static layer with a full-fledged functional language, and a dynamic layer of quantum circuits obtained by partially evaluating the source.

In our proof of concept, we study a minimal language of classical circuits inspired by Π-ware a formal hardware description and verification language proposed by Flor, Swierstra, and Sijsling [15]. This allows us to focus on the two-level aspect instead of having to deal with linearity and unitary operators which are specific to the Quantum setting.

6.1 Types and Terms

Our definition of types should now be mostly unsurprising. We have function spaces (this time confined to the static layer), a lifting construct allowing the embedding of dynamic types in the static layer at the source stage, and finally a type of circuits ⟨ *i* | *o* ⟩ characterised by their input (i) and output (o) arities, each represented by a natural number in the host language.

```
data Type : Stage ph → Set where
  _'⇒_ : (A B : Type sta) → Type sta
  '⇑_   : Type {src} dyn → Type sta
  '⟨_|_⟩ : (i o : ℕ) → Type {ph} dyn
```

Next, we extend the basic simply typed lambda calculus with quotes and splices with term constructors for circuit descriptions. They will all belong to the dynamic stage. Our first constructor gives us the universal nand gate. Its type records the fact it takes two inputs and returns a single output.

```
'nand : ∀[ Term ph dyn '⟨ 2 | 1 ⟩ ]
```

Next, we have a constructor for the parallel composition of existing circuits. The input and output arities of the resulting circuit are obtained by adding up the respective input and output arities of each of the components.

```
'par : ∀[ Term ph dyn '⟨ i₁      | o₁      ⟩ ⇒
          Term ph dyn '⟨      i₂ |      o₂ ⟩ ⇒
          Term ph dyn '⟨ i₁ + i₂ | o₁ + o₂ ⟩ ]
```

We can also compose circuits sequentially, provided that the output arity of the first circuit matches the input arity of the second.

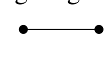
```
'seq : ∀[ Term ph dyn '⟨ i | m ⟩ ⇒
          Term ph dyn '⟨ m | o ⟩ ⇒
          Term ph dyn '⟨ i | o ⟩ ]
```

Finally, we follow the Π-ware [15] approach and offer a general rewiring component. A 'mix' of i inputs returning o outputs is defined by a vector (i.e. a list of known length) of size o containing finite numbers between 0 and i corresponding to the input the output is connected to. This allows arbitrary duplications and deletions of inputs.

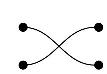
```
'mix : Vec (Fin i) o → ∀[ Term ph dyn '⟨ i | o ⟩ ]
```

Typical examples include 'id₂ (the *id*entity circuit on two inputs), 'swap (the circuit *swap*ping its two inputs), and 'dup (the circuit *dup*licating its single input). We present them below together with the corresponding wiring diagrams.

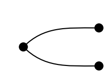
```
'id₂ : ∀[ Term ph dyn '⟨ 2 | 2 ⟩ ]
'id₂ = 'mix (0 :: 1 :: [])
```



```
'swap : ∀[ Term ph dyn '⟨ 2 | 2 ⟩ ]
'swap = 'mix (1 :: 0 :: [])
```

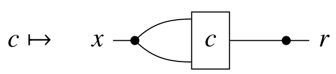


```
'dup : ∀[ Term ph dyn '⟨ 1 | 2 ⟩ ]
'dup = 'mix (0 :: 0 :: [])
```



We can then define our first real example: 'diag, a static program taking a circuit with two inputs and one output and returning a circuit with one input and one output. It does so by first duplicating the one input using 'dup and then feeding it to both of the argument's ports. We present it below together with the corresponding circuit diagram.

```
'diag : ∀[ Term src sta ('⇑ '⟨ 2 | 1 ⟩ '⇒ '⇑ '⟨ 1 | 1 ⟩) ]
'diag = 'lam '⟨ 'seq 'dup ('∼ 'var here) ⟩
```



We can then obtain the 'not gate by taking the diagonal of the 'nand built-in gate.

```
'not : ∀[ Term src dyn '⟨ 1 | 1 ⟩ ]
'not = '∼ 'app 'diag '⟨ 'nand ⟩
```

Staging this definition does evaluate away all of the function calls to yield a simple circuit obtained by sequentially composing 'dup and 'nand as shown below.

```
'⟨ 1 | 1 ⟩ ∋ 'not ⇝ 'seq 'dup 'nand
```

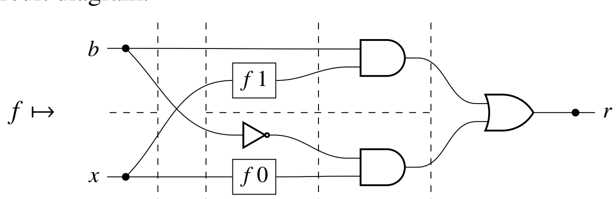
Using standard constructions, we can define 'and and 'or in terms of the universal 'nand gate.

```
'and : ∀[ Term src dyn '⟨ 2 | 1 ⟩ ]     'or : ∀[ Term src dyn '⟨ 2 | 1 ⟩ ]
'and = 'seq 'nand 'not                  'or = 'seq ('par 'not 'not) 'nand
```

Going back to a slightly more complex setting, adding booleans in the static layer lets us once again define more interesting terms. For instance, the following 'tab circuit *tab*ulating its input: given a function that takes a boolean and computes a one-input one-output circuit, it returns a circuit with two inputs and one output that has the same behaviour.

```
'tab : ∀[ Term src sta (('Bool '⇒ '⇑ '⟨ 1 | 1 ⟩) '⇒ '⇑ '⟨ 2 | 1 ⟩) ]
'tab = 'lam '⟨ 'seq ('seq ('seq
        ('par 'dup 'dup)
        ('mix (0 :: 2 :: 1 :: 3 :: [])))
        ('par ('seq ('par 'id₁ ('∼ 'app ('var here) 'true)) 'and)
              ('seq ('par 'not ('∼ 'app ('var here) 'false)) 'and)))
        'or ⟩
```

Using dashed lines to separate the different constituting parts of the circuit as defined above, we obtain the following circuit diagram.



This term is not in and of itself particularly useful but its generalisation to one that could take a function computing an $\langle\, i \mid o \,\rangle$ circuit and return an equivalent $\langle\, 1 + i \mid o \,\rangle$ circuit would allow us to build arbitrarily complex circuits by tabulating static n-ary boolean functions.

This would however require a setting where the static layer is dependently typed like in Kovács' original work, something out of scope for this paper.

7 Related work

Prior work on partial evaluation and metaprogramming abounds so we will only focus on the very most relevant works involving strong types.

Quantum Circuits Generation. As already mentioned in Section 6, such two level systems occur naturally when defining high level languages for (quantum) circuit descriptions. Rennela and Staton's EWire language is itself the categorical treatment of a minor generalisation of Paykin, Rand, and Zdancewic's QWire [22], a clear invariant-enforcing improvement over the weakly typed Haskell embedded domain specific language Quipper [16]. EWire is an ad-hoc construction which, although not worded explicitly in terms of a two level type theory, effectively is one: quotes and splices are called boxing and unboxing, and a QWire-inspired partial normalisation procedure proven to be semantics-preserving is defined.

SMT Constraints Generation. In their work on compiling higher order specifications to SMT constraints [12], Daggit, Atkey, Kokke, Komendantskaya, and Arnaboldi designed a cunning 'translation by evaluation' to partially evaluate specifications written in a full featured high level functional language (without recursion) into first order SMT constraints. This is not explicitly designed as a two level system and so the sucess of the partial evaluation comes from a careful but ultimately ad-hoc design rather than a systematic approach. Unlike ours, their system however comes with a proof of correctness: the generated formula is proven to be logically equivalent to the high level specification. This is an obvious avenue for future work on our part.

Typed Metaprogramming. Jang, Gélineau, Monnier, and Pientka's Mœbius [17] defines a type theory with a built in notion of quasiquotations that can be used to generate programs in a type-safe manner. The language lets metaprograms inspect the code fragments they are passed as arguments thus allowing e.g. the implementation of optimisation passes post-processing the result of a prior metaprogram. This is extremely powerful, at the cost of a more complex underlying theory. In Mœbius the meta and object language are essentially the same but it does not seem to be a necessary restriction.

8 Future Work

Soundness and Completeness. We focused here on the intrinsically typed language description, the corresponding model construction, and the acquisition of a staging-by-evaluation function as a corollary. Following Catarina Coquand's work on formalising normalisation by evaluation [10] we could additionally introduce the appropriate logical relations to prove that this process is sound and complete with respect to a small step semantics for the static layer.

Dependently Typed Circuit Description Language. Our undergraduates are already being taught digital logic using a functional-style circuit description language. Extending it with a dependently typed meta-programming layer would allow them to structure their understanding of the generic construction of arithmetic circuits for arbitrarily large inputs.

Generic Two Level Construction. Even though we have seen that having two wildly different language layers can be extremely useful, a two-level construction with exactly the same features is still very useful: it lets programmers use their language of choice as its own metaprogramming facilities. Correspondingly, giving a generic treatment of the construction taking a language and returning its standard two-level version is an important endeavour. A promising approach involves defining such a transformation by induction over a universe of language descriptions [2].

Acknowledgments

We would like to thank Bob Atkey for his suggestion to index stages by a phase, thus allowing us to ensure that a staged term does not have any static subterm.

References

[1] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: programming infinite structures by observations. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 27--38. https://doi.org/10.1145/2429069.2429075

[2] Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. 2021. A type- and scope-safe universe of syntaxes with binding: their semantics and proofs. *J. Funct. Program.* 31 (2021), e22. https://doi.org/10.1017/S0956796820000076

[3] Guillaume Allais, James Chapman, Conor McBride, and James McKinna. 2017. Type-and-scope safe programs and their proofs. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, Yves Bertot and Viktor Vafeiadis (Eds.). ACM, 195--207. https://doi.org/10.1145/3018610.3018613

[4] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. 1995. Categorical Reconstruction of a Reduction Free Normalization Proof. In *Category Theory and Computer Science, 6th International Conference, CTCS '95, Cambridge, UK, August 7-11, 1995, Proceedings (Lecture Notes in Computer Science, Vol. 953)*, David H. Pitt, David E. Rydeheard, and Peter T. Johnstone (Eds.). Springer, 182--199. https://doi.org/10.1007/3-540-60164-3_27

[5] Thorsten Altenkirch and Bernhard Reus. 1999. Monadic Presentations of Lambda Terms Using Generalized Inductive Types. In *Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1683)*, Jörg Flum and Mario Rodrıguez-Artalejo (Eds.). Springer, 453--468. https://doi.org/10.1007/3-540-48168-0_32

[6] Françoise Bellegarde and James Hook. 1994. Substitution: A Formal Methods Case Study Using Monads and Transformations. *Sci. Comput. Program.* 23, 2-3 (1994), 287--311. https://doi.org/10.1016/0167-6423(94)00022-0

[7] Ulrich Berger and Helmut Schwichtenberg. 1991. An Inverse of the Evaluation Functional for Typed lambda-calculus. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*. IEEE Computer Society, 203--211. https://doi.org/10.1109/LICS.1991.151645

[8] Richard S. Bird and Ross Paterson. 1999. De Bruijn Notation as a Nested Datatype. *J. Funct. Program.* 9, 1 (1999), 77--91. https://doi.org/10.1017/s0956796899003366

[9] Alonzo Church. 1941. *The calculi of lambda-conversion*. Number 6 in Annals of Mathematics Studies. Princeton University Press.

[10] Catarina Coquand. 2002. A Formalised Proof of the Soundness and Completeness of a Simply Typed Lambda-Calculus with Explicit Substitutions. *High. Order Symb. Comput.* 15, 1 (2002), 57--90. https://doi.org/10.1023/A:1019964114625

[11] Thierry Coquand and Peter Dybjer. 1997. Intuitionistic Model Constructions and Normalization Proofs. *Math. Struct. Comput. Sci.* 7, 1 (1997), 75--94. https://doi.org/10.1017/S0960129596002150

[12] Matthew L. Daggitt, Robert Atkey, Wen Kokke, Ekaterina Komendantskaya, and Luca Arnaboldi. 2023. Compiling Higher-Order Specifications to SMT Solvers: How to Deal with Rejection Constructively. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*, Robbert Krebbers, Dmitriy Traytel, Brigitte Pientka, and Steve Zdancewic (Eds.). ACM, 102--120. https://doi.org/10.1145/3573105.3575674

[13] Nicolaas Govert de Bruijn. 1972. Lambda Calculus Notation with Nameless Dummies. In *Indagationes Mathematicae*, Vol. 75. Elsevier, 381--392.

[14] Peter Dybjer. 1994. Inductive Families. *Formal Aspects Comput.* 6, 4 (1994), 440--465. https://doi.org/10.1007/BF01211308

[15] João Paulo Pizani Flor, Wouter Swierstra, and Yorick Sijsling. 2015. Pi-Ware: Hardware Description and Verification in Agda. In *21st International Conference on Types for Proofs and Programs, TYPES 2015, May 18-21, 2015, Tallinn, Estonia (LIPIcs, Vol. 69)*, Tarmo Uustalu (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1--9:27. https://doi.org/10.4230/LIPIcs.TYPES.2015.9

[16] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. An Introduction to Quantum Programming in Quipper. In *Reversible Computation - 5th International Conference, RC 2013, Victoria, BC, Canada, July 4-5, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7948)*, Gerhard W. Dueck and D. Michael Miller (Eds.). Springer, 110--124. https://doi.org/10.1007/978-3-642-38986-3_10

[17] Junyoung Jang, Samuel Gélineau, Stefan Monnier, and Brigitte Pientka. 2022. Mœbius: metaprogramming using contextual types: the stage where system f can pattern match on itself. *Proc. ACM Program. Lang.* 6, POPL (2022), 1--27. https://doi.org/10.1145/3498700

[18] András Kovács. 2022. Staged compilation with two-level type theory. *Proc. ACM Program. Lang.* 6, ICFP (2022), 540--569. https://doi.org/10.1145/3547641

[19] Per Martin-Löf. 1984. *Intuitionistic type theory*. Studies in proof theory, Vol. 1. Bibliopolis.

[20] John C. Mitchell and Eugenio Moggi. 1991. Kripke-Style Models for Typed lambda Calculus. *Ann. Pure Appl. Log.* 51, 1-2 (1991), 99--124. https://doi.org/10.1016/0168-0072(91)90067-V

[21] Ulf Norell. 2008. Dependently Typed Programming in Agda. In *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures (Lecture Notes in Computer Science, Vol. 5832)*, Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra (Eds.). Springer, 230--266.

https://doi.org/10.1007/978-3-642-04652-0_5
[22] Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: a core language for quantum circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 846--858. https://doi.org/10.1145/3009837.3009894
[23] Mathys Rennela and Sam Staton. 2020. Classical Control, Quantum Circuits and Linear Logic in Enriched Category Theory. *Log. Methods Comput. Sci.* 16, 1 (2020). https://doi.org/10.23638/LMCS-16(1:30)2020

Received 18 October 2023; revised TBD; accepted TBD