Deeper Shallow Embeddings

Jacob Prinz
University of Maryland, College Park, USA

G. A. Kavvos
University of Bristol, UK

Leonidas Lampropoulos
University of Maryland, College Park, USA

Abstract

Deep and shallow embeddings are two popular techniques for embedding a language in a host language with complementary strengths and weaknesses. In a deep embedding, embedded constructs are defined as data in the host: this allows for syntax manipulation and facilitates metatheoretic reasoning, but is challenging to implement—especially in the case of dependently typed embedded languages. In a shallow embedding, by contrast, constructs are encoded using features of the host: this makes them quite straightforward to implement, but limits their use in practice.

In this paper, we attempt to bridge the gap between the two, by presenting a general technique for extending a shallow embedding of a type theory with a deep embedding of its typing derivations. Such embeddings are almost as straightforward to implement as shallow ones, but come with capabilities traditionally associated with deep ones. We demonstrate these increased capabilities in a number of case studies; including a DSL that only holds affine terms, and a dependently typed core language with computational beta reduction that leverages function extensionality.

2012 ACM Subject Classification Software and its engineering

Keywords and phrases type theory, shallow embedding, deep embedding, Agda

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.19

Acknowledgements This material is based upon work supported by NSF award #2107206, *Efficient and Trustworthy Proof Engineering* (any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF).

1 Introduction

Hosting a programming language inside another one is one of our favorite pastimes as programming language researchers. Such *embeddings* have proven useful in a variety of domains, ranging from particular applications (such as the design of hardware [9] or the writing of random property-based generators [10]), all the way to the mechanization of the metatheory of such applications (such as Kami [8] or Luck [19] respectively).

The complexity of such embeddings varies significantly depending on the features of both the *object language* and the *host language* involved. For example, embedding a simply-typed object language in a functional host language is a relatively straightforward task, yet one that has proved quite useful in practice, leading to a proliferation of libraries in mainstream ecosystems [26, 25, 17, 22]. On the other hand, embedding a dependently-typed language in another is a highly nontrivial task that gives rise to foundational challenges and that has received significant attention in recent years [1, 18].

Starting with Boulton et al. [5], language embeddings have been broadly classified as either *deep* or *shallow*. An embedding is deep when the terms of the object language are represented as inductive data in the host language. In a deep embedding terms may be arbitrarily manipulated and inspected via the usual mechanism of pattern matching. Deep

```
data Type : Set where
  _⇒_ : Type → Type → Type
  base : Type

data Ctx : Set where
  ∅ : Ctx
  _,_ : Ctx → Type → Ctx

data Var : (Γ : Ctx) → Type → Set where
  same : ∀{Γ T} → Var (Γ , T) T
  next : ∀{Γ T A} → Var Γ A → Var (Γ , T) A

data Exp : Ctx → Type → Set where
  var : ∀{Γ T} → Var Γ T → Exp Γ T
  lambda : ∀{Γ A B} → Exp (Γ , A) B
    → Exp Γ (A ⇒ B)
  app : ∀{Γ A B} → Exp Γ (A ⇒ B)
    → Exp Γ A → Exp Γ B
  tt : ∀{Γ} → Exp Γ base
```

Figure 1 A deep embedding of STLC in Agda

embeddings for simply-typed languages are reasonably straightforward to construct, e.g. using an intrinsically-typed representation of terms as an inductive family [3, §3].

By contrast, shallow embeddings directly expand the constructs of the object language in terms of constructs of the host language. In the language of semantics, if a deep embedding can be thought of as defining the *initial* syntactic model of the object language, then a shallow embedding can be thought of as an *arbitrary* semantic model of the object language, but expressed and manipulated in the host language instead of mathematics [5, §5].

Deep Embeddings

For concreteness, consider the standard (intrinsic) deep embedding for the simply-typed lambda calculus (STLC) in Figure 1. We focus on the fragment consisting of a single base type, and functions. Contexts are lists of types. Variables are positions in that list, viz. de Bruijn indices. Terms are parameterized by a type and context. Every constructor of Term represents an STLC typing rule. For example, the app constructor takes a term of type $A \Rightarrow B$ and a term of type A, and produces produce a term of type B—parametrically in any context Γ. Functions over the syntax of STLC terms can then be defined using pattern-matching/induction, allowing a plethora of operations (e.g. substitution, reductions, optimizations) and metatheoretic proofs (e.g. admissibility of substitution).

In practice, Agda's type inference system plays very nicely with intrinsically-typed DSLs. Because the type and context are parameters of Term, Agda can infer them in the same way that it would for Agda programs. For example, when given the definition

```
lambda (var same) : Term (base → base)
```

Agda is able to infer that it is well-typed without additional information. It is thus not necessary for the user to specify the type of any part of this expression (e.g. the variable), thereby greatly increasing the usability of the DSL.

However, the same is not true for dependently-typed object languages: because of the presence of the *type conversion* rule

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A = B \text{ type}}{\Gamma \vdash M : B}$$

there is considerable overhead in defining a deep embedding of the object language, as we often have to somehow transport terms across type equalities. In practice this overhead is

```
Ctx = Set
Type : Ctx → Set
Type Γ = Γ → Set

∅ : Ctx
∅ = ⊤
cons : (Γ : Ctx) → Type Γ → Ctx
cons Γ T = Σ Γ T

Var : (Γ : Ctx) → Type Γ → Set
Var Γ T = (γ : Γ) → T γ
same : ∀{Γ T} → Var (cons Γ T) (T ∘ proj₁)
same = λ (γ , t) → t
next : ∀{Γ T A} → Var Γ A
  → Var (cons Γ T) (A ∘ proj₁)
next x = λ (γ , t) → x γ

Term : (Γ : Ctx) → Type Γ → Set
Term Γ T = (γ : Γ) → T γ

Π : ∀{Γ}
  → (A : Type Γ) → Type (cons Γ A) → Type Γ
Π A B = λ γ → (a : A γ) → B (γ , a)

U : ∀{Γ} → Type Γ
U γ = Set

var : ∀{Γ T} → (icx : Var Γ T) → Term Γ T
var x = x

lambda : ∀{Γ A B} → Term (cons Γ A) B
  → Term Γ (Π A B)
lambda e = λ γ a → e (γ , a)

app : ∀{Γ A B} → Term Γ (Π A B)
  → (a : Term Γ A)→ Term Γ (λ γ → B (γ , a γ))
app e₁ e₂ = λ γ → (e₁ γ) (e₂ γ)
```

Figure 2 Shallow embedding of dependent type theory

prohibitive, regardless of whether we are using the dependently-typed object language or proving things about it.

A number of researchers have attempted such deep embeddings of dependently-typed languages with varying degrees of success and completeness; see for example [12, 7, 1]. Each of these attempts represents a complex feat of proof engineering, often using various techniques (such as setoids), or assuming certain advanced features in the host language (such as quotient inductive types). In contrast to the simply-typed case, a simple, practical deep embedding of dependent type theory appears impossible with current technology. This invites a search for an easier alternative.

Shallow Embeddings

In contrast, shallow embeddings do not represent terms of the object language as inductive data, but rather directly interpret them as values in the host language.

For example, consider the shallow embedding of dependent type theory shown in Figure 2. Like with the deep embedding, we have definitions of types, contexts, variables, and terms. However, unlike the deep embedding, these are not datatypes. Instead, Ctx is defined as Agda's universe (called Set), and Type is the type of families of types over a given context. Finally, given a context Γ and type T, terms and variables are then defined as dependent functions $(\gamma : \Gamma) \to T\,\gamma$ over the family T. Moreover, for each term constructor that we had in the deep embedding, we have a corresponding definition in the shallow embedding of the corresponding type. For example, the lambda definition is defined using an Agda λ expression, and the app definition is defined using standard Agda function application.

Thus, each artifact of the object language is interpreted by its counterpart in the host language. Owing to its similarity to the set-theoretic model [16, §3] this is sometimes referred to as the *standard model* [1, §4], or even the *metacircular interpretation* of type theory [18].

This shallow embedding of dependent type theory obviates many of the difficulties one

encounters when trying to define a deep embedding. The reason is that types themselves are no longer mere pieces of syntax, but mathematical objects that are subject to the definitional rules of Agda. Thus, the type equalities we had to transport over vanish [21]. As the shallow embedding inherits the definitional behavior of the host language, we find ourselves in a situation that enables quick and easy prototyping. Unfortunately, there is no free lunch: the fact that our terms are now semantic objects too means that we may no longer pattern-match on them.

A Middle Ground?

Given the complementary strengths and weaknesses of deep and shallow embeddings, it is natural to ask whether there is something in the middle: is there a form of embedding which is almost as easy to implement as a shallow embedding, but provides some of the extended, syntactic capabilities of a deep embedding?

In this paper we propose an answer to this question, which we call *deeper shallow embeddings*. To build a deeper shallow embedding we need a pre-existing shallow embedding of the object language. Then, the contexts and terms of the deeper shallow embedding are defined by 'wrapping' the contexts and terms of the shallow embedding in an inductive data type. Following the technique of McBride [21], the types remain shallowly embedded, so that the host language takes care of type conversion for us.

We claim that deeper shallow embeddings preserve the mathematical simplicity of shallow embeddings, yet add extra capabilities which are useful for writing DSLs. This claim is substantiated by demonstrating these capabilities in practice. For example, we use them to (1) restrict the terms allowed in the DSL, (2) add metadata to terms (and perform computations dependent on this metadata), and (3) do a limited form of pattern-matching (which becomes more powerful in the presence of function extensionality).

Concretely, we make the following contributions:

- We present a way of deepening any shallow embedding, preserving its mathematical simplicity while also gaining additional capabilities that one might expect would require a deep embedding (Section 2). We show it by example on three shallow embeddings: the standard model (Section 2), a standard model for affine terms (Section 3), and a shallow embedding built from an inductive-recursive universe construction (Section 5).
- We demonstrate the usefulness of deeper shallow embeddings through a series of case studies showcasing different features that they exhibit over standard ones: adding metadata, restricting terms, and performing induction over terms (Section 3).
- We consider syntactic transformations such as substitution (Section 4), and show how to further increase the expressive power of a deeper shallow embedding by assuming function extensionality. This gives us the power to define—and compute — β-reduction (Section 5).

All of these results are formalized in Agda, available at `https://github.com/jeprinz/Deeper-Shallow-Embeddings`). Finally, we discuss related work in Section 6 and conclude by discussing limitations and future directions in Section 7.

2 Deeper Shallow Embeddings

As discussed before, a shallow embedding consists of a set of definitions in the host language. The valid object language programs in this embedding are exactly the well-typed terms built by combining these definitions. Thus, given a term `t` in the host language (here, Agda), the statement "`t` is a term of the shallow embedding" cannot be easily expressed in the host

```
data Context : S.Ctx → Set where
  ∅ : Context S.∅
  _,_ : Context sΓ → (T : S.Type sΓ) → Context (S.cons sΓ T)

data Var : {sΓ : S.Ctx} → (Γ : Context sΓ) → (T : S.Type sΓ)→ (S.Term sΓ T) → Set where
  same : Var (Γ , T) (T ∘ proj₁) S.same
  next : Var {sΓ} Γ A s → Var (Γ , T) (T ∘ proj₁) (S.next s)

data Term : {sΓ : S.Ctx} → (Γ : Context sΓ) → (T : S.Type sΓ)→ S.Term sΓ T → Set where
  lambda : Term (Γ , A) B s → Term Γ (S.Π A B) (S.lambda s)
  var : Var Γ T s → Term Γ T s
  app : Term Γ (S.Π A B) s₁ → (x : Term Γ A s₂)→ Term Γ (λ γ → B (γ , s₂ γ)) (S.app s₁ s₂)
  Π : (A : Term Γ S.U s₁) → (B : Term (Γ , s₁) S.U s₂)→ Term Γ S.U (S.Π₀ s₁ s₂)
  U : Term Γ S.U S.U
```

Figure 3 Deepening the shallow embedding of dependent type theory. Each constructor is a wrapper around a shallow constructor (prefixed by S).

language—even if t is of a type that is evidently in the image of the shallow embedding. In fact, the ability to do this amounts to solving the *definability* problem in semantics.

By contrast, deeper shallow embeddings *internalize* this statement. By defining an inductive family of contexts and terms, we effectively *tag* the definable elements of the shallow embedding in a way that records their construction. This leaves us with something in between a deep and shallow embedding. On the one hand, our contexts, types, and terms carry elements of a shallow embedding. On the other hand, our typing derivations are deeply embedded in a datatype, so we may pattern-match on them.

To make the idea more concrete, let us revisit the shallow embedding of Figure 2. We will henceforth refer to this shallow embedding as S. By definition, to have a term of this shallow embedding means to have a shallow context Γ : Ctx, a shallow type T : Type Γ, and a dependent function (γ : Γ) → T γ. To deepen this embedding we must define a new datatype Term that is indexed by these three values. The constructors of the datatype encode the typing rules of the theory which it represents. The full definition may be found in Figure 3. All references to definitions in the shallow embedding start with the prefix S.

To understand this deepened term type better, we may for instance consider its lambda constructor. The λ constructor of the shallow embedding is referred to as S.lambda. If s is a term of the shallow embedding of the appropriate type and context, then S.lambda s represents the λ-abstraction of that term in the shallow embedding. Then, the lambda constructor of the deepened embedding inputs a term of a type parameterized by the shallow term s, and outputs an expression of type parameterized by the term S.lambda s. For clarity we write each deep constructor so that it refers to the corresponding shallow definition, but note that the code repetition could be eliminated by unfolding the definitions of the shallow embedding directly into the constructors of the datatype.

We can of course extend this idea to the construction of contexts and variables as well. The deepened context datatype Context : S.Ctx → Set is a family over the contexts S.Ctx of the shallow embedding. When we have an inhabitant of Context sΓ we know that (1) sΓ is a well-formed context in the shallow embedding, and (2) that sΓ is definable by starting from the empty context, and extending it by shallow types. Similarly, Var sΓ sT s is inhabited when s is a well-formed variable in the shallow embedding.

For a quick example of the deepened embedding, here is a definition of the identity

function on the universe:

```
idU : Term ∅ (S.Π S.U S.U) _
idU = lambda (var same)
```

Notice that Agda is able to infer the corresponding shallow term, which has been elided and replaced by "_". We are always able to extract the shallow term from a deepened embedding:

```
extract : ∀{sΓ Γ T t} → Term {sΓ} Γ T t → S.Term sΓ T
extract {sΓ} {Γ} {T} {t} e = t
```

Hence, anything that can be proven about shallow embeddings can also be proven about deeper ones. For example, we can prove consistency (relative to the ambient type theory); that is, we can show that if we have an inhabitant of the shallow empty type, then we have an inhabitant of the empty type in the host language:

```
consistency : ∀{t} → Term {S.∅} ∅ (λ _ → ⊥) t → ⊥
consistency e = (extract e) tt
```

Syntax and Universe Level Simplifications

To facilitate readability, we have omitted certian details present in the formalization when presenting Agda code. In particular, we mostly leave out universally quantified parameters when they can easily be inferred. More importantly, we have also omitted all traces of universe levels. For example, in the formalization, the U constructor of Term looks like:

$$\mathsf{U}_0 : \forall\{s\Gamma\ \Gamma\} \to \mathsf{Term}\ \{s\Gamma\}\ \Gamma\ \mathsf{S.U}_1\ \mathsf{S.U}_0$$

Each universe level included in the deeper embedding needs it's own constructor, so for example U_1 is a separate constructor. Additionally, Term (and its shallow counterpart) needs some constructors to deal with universe level cumulativity. The full definition has three additional constructors: Lift, which raises a type to the next level; lift, which raises a term to a lifted type; and lower, which lowers a term from a lifted type. The corresponding shallow embedding definitions are implemented with the Agda terms of the same name.

3 Advantages of Deeper Shallow Embeddings, by Example

In this section we show that deeper shallow embeddings have usability advantages vis-a-vis both shallow and deep embeddings. We do so by example: we present deeper shallow embeddings which enable features that are not supported otherwise. Our discussion is focused around three examples: metadata on terms, pattern matching, and term restriction.

3.1 Metadata: Named Variables

Most embeddings of dependently-typed languages in proof assistants such as Agda or Coq rely on de Bruijn indices for representing and accessing variables: see e.g. [21, 18]. Evidently, this state of affairs is undesirable from a usability perspective.

We show that a deeper shallow embedding can be used to define a DSL with named variables. This is achieved without changing the standard shallow embedding of dependent types. Instead, the deepened embedding carries *metadata* about terms like a deep embedding

could, in particular the names of variables. This way a user of the DSL is able to write lambda "x" (var "x") for the identity function, instead of the usual expression lambda (var same) .

To achieve this, we add a string along with each type in the deepened context:

```
data Context : S.Ctx → Set j where
  ∅ : Context S.∅
  _,_::_ : Context sΓ → String → (T : S.Type sΓ) → Context (S.cons sΓ T)
```

This allows us to write a function:

```
findVar : (Γ : Context sΓ) → String → Σ[ t ∈ Var Γ T t ]
```

which searches for a variable name in the context. If the variables exists, it returns the shallow type and term of the variable as the Var value corresponding to it.

Unlike de Bruijn indices, this does not preclude the user from attempting to write a nonsensical term like lambda "x" (var "y") in an empty context. To deal with that eventuality we introduce an error type and term

```
data Error : Set where
  error : Error
```

We can now implement functions which, given a variable name, search the context for the corresponding type and term. If the variable is not found, the Error type is used.

```
resultType : (Γ : Context sΓ) → String → S.Type sΓ
resultType Γ name with findVar Γ name
... | nothing = λ _ → Error
... | just ((T , t) , x) = T

resultTerm : (Γ : Context sΓ)
  → (name : String) → S.Term sΓ (resultType Γ name)
resultTerm Γ name with findVar Γ name
... | nothing = λ _ → error
... | just ((T , t) , x) = t
```

We can now implement Term. The var constructor takes a string as an argument, and uses the aforementioned functions to compute its own type:

```
var : (name : String) → Term Γ (resultType Γ name) (resultTerm Γ name)
```

Finally, we let the lambda constructor take a string argument as well:

```
lambda : (name : String) → Term (Γ , name :: A) B s → Term Γ (S.Π A B) (S.lambda s)
```

Putting all of this together, we can now write terms with named variables! For example, the identity function can be written as:

```
id : Term ∅ (λ _ → (X : Set) → X → X) _
id = lambda "X" (lambda "x" (var "x"))
```

So, what if we try to use an unbound variable? Suppose we enter the definition

```
id : Term ∅ (λ _ → (X : Set) → X → X) _
id = lambda "X" (lambda "x" (var "y"))
```

This definition will not typecheck in Agda! By writing this definition we have communicated to Agda our expected type; we have also fixed the context to be empty. In the process of type-checking, Agda introduces the two variables `"X"` and `"x"` in the context, and then looks for `"y"`. It fails to find it, so var constructs an element of type `Term Γ ( λ _ → Error ) ( λ _ → error )` which does not match the stated type, causing a type error.

It is important to note that this technique is possible exactly because (a) `λ _ → Error` is a perfectly acceptable type of the shallow embedding, yet (b) none of our Term constructors create elements of `Term Γ ( λ _ → Error ) ( λ _ → error )` .

3.2 Pattern Matching and Induction: Compilation

Another drawback of shallow embeddings is that they do not provide the ability to induct on their terms. Given an arbitrary element `t : S.Term nil T`, there is not much we can really do: T is an arbitrary family over the empty context, so essentially an arbitrary Agda type. Hence, there is very little we can do, either with the term t or the type T.

In contrast, terms in the deeper shallow embedding are given by an inductive data type. It is therefore possible to write functions that pattern-match on them. For example, the following function compiles a dependently-typed term to JavaScript.

```
compileToJs : Term Γ T s → String
compileToJs {Γ} (lambda e) =
   "function(x" ++ (show (len Γ)) ++ ")"++ "{" ++ compileToJs e ++ "}"
compileToJs {Γ} (var x) =
  "x" ++ (show ((len Γ) - (index x)))
compileToJs (app e₁ e₂) =
   "(" ++ (compileToJs e₁) ++ "" ++ (compileToJs e₂) ++ ")"
```

The compilation proceeds in a fairly obvious way: a lambda is compiled to an anonymous function; and variable names are determined by their position in the context, using len and index to compute the relevant indices. Because of the lack of an induction principle, writing this simple function over a shallow embedding is impossible.

3.3 Syntactic Restrictions: Affine Terms

A final advantage of deeper shallow embeddings is that they provide the ability to restrict the terms that appear in the DSL. This has two advantages: (1) on an engineering level, it limits the interface exposed to the DSL programmer to a safer fragment; and (2) on a design level, it enables us to 'carve out' a subset of interest of a shallow embedding.

When embedding a DSL in a shallow manner, the intention is that the user will build terms by composing the given definitions. However, shallow embeddings cannot stop a user from writing any term of the base language with a fortuitous type. For example, if one wants an element of `S.Term S.∅ (S.Π S.U S.U)` then instead of writing `S.lambda (S.var S.same)` , one could simply write `λ y x → x` , circumventing the intent of the DSL designer. This example is fairly innocuous, but it is not difficult to imagine that this might become problematic in more complicated languages.

Kaposi, Kovács, and Kraus [18] propose a solution to this that uses Agda records for data hiding. The idea is that we wrap the contraptions of the shallow embedding in unary record types whose fields are private, and import only the wrapped model. That way the user cannot 'access' the underlying representation. However, because of Agda's η-rule for record types, the wrapped model retains the definitional properties of the shallow embedding.

While this approach works, it relies heavily on features specific to Agda. If we use deeper shallow embeddings instead, only constructors of Term can be used to obtain terms. Therefore, the terms of the DSL can be restricted without making use of any data-hiding mechanism.

As an example, we can define a DSL which holds only affine terms, i.e. a DSL whose terms can use a variable in the context *at most once*. First, we need to augment contexts with some usage annotation. We define a family VarData : ContextsΓ → Set over deepened contexts, which stores a boolean flag for each type in the context. The flag records whether that variable has been used. Next, we inductively define a ternary relation

```
Check : VarData Γ → VarData Γ → VarData Γ → Set j
```

between flagged contexts, which holds if the first two contexts do not use the same variable, while the third context uses all variables used by either of the first two. This is reminiscent of ternary 'context split' relations often used with linear types, e.g. session types [24, §3].

```
data VarData : Context sΓ → Set where
  ∅ : VarData ∅
  _,_ : VarData Γ → Bool → VarData (Γ , T)

data Check : VarData Γ → VarData Γ → VarData Γ→ Set j where
∅ : Check ∅ ∅ ∅
consLeft : (T : S.Type sΓ) → Check Γ₁ Γ₂ Γ₃ → Check (Γ₁ , true) (Γ₂ , false) (Γ₃ , true)
consRight : (T : S.Type sΓ) → Check Γ₁ Γ₂ Γ₃ → Check (Γ₁ , false) (Γ₂ , true) (Γ₃ , true)
consNeither : (T : S.Type sΓ) → Check Γ₁ Γ₂ Γ₃ → Check (Γ₁ , false) (Γ₂ , false) (Γ₃ , false)
```

Next, we define AffineTerm, which is a deepened type of terms that only holds affine terms. It is defined in much the same way as the standard deepened term type, but it incorporates elements of Check and VarData as evidence that the embedded terms are affine. Specifically, this evidence is used in the app constructor to ensure that the two subterms do not both use the same variable.

```
data AffineTerm : (Γ : Context sΓ) → VarData Γ → (T : S.Type sΓ) → S.Term sΓ T → Set j where
app : AffineTerm Γ Γ₁ (S.Π A B) s₁ → (x : AffineTerm Γ Γ₂ A s₂) → Check Γ₁ Γ₂ Γ₃
    → AffineTerm Γ Γ₃ (λ γ → B (γ , s₂ γ)) (S.app s₁ s₂)
Π : AffineTerm Γ Γ₁ S.U s₁ → AffineTerm (Γ , s₁) (Γ₂ , b) S.U s₂ → Check Γ₁ Γ₂ Γ₃
  → AffineTerm Γ Γ₃ S.U (S.Π s₁ s₂)
- ...
```

Finally, we are at a point where one may clearly witness the power of deeper shallow embeddings and the ability to pattern-match on them. We are able to define a function:

```
checkAffine : Term Γ T t → Maybe (Σ (VarData Γ) (λ vd → AffineTerm Γ vd T t))
```

which checks whether a given Term is affine, and—if so—reconstructs it as an AffineTerm. To achieve that we will need some helper functions, whose definitions we omit. Amongst these the most important one is check, which inputs two elements of VarData, and—if they do not conflict with each other—returns their combination alongside an element of Check. The 'affinization' function itself recursively calculates the variables used by each expression. Whenever an app case is encountered, it checks that no variable is used twice.

4 Renamings and Substitutions

Having showcased some advantages of deeper shallow embeddings with small examples, we now turn to the more complicated operations of *renaming* and *substitution* that are central to dependent type theory.

As shallow embeddings are essentially semantic interpretations, substitutions are definable operations. In our running example of the 'standard' interpretation, the type of all substitutions is the type of all functions between contexts; the action of a substitution on types and terms is then determined by function application:

```
Sub : Ctx → Ctx → Set
Sub Γ₂ Γ₁ = Γ₂ → Γ₁

extend : Sub Γ₂ Γ₁ → Term Γ₁ T → Sub Γ₂ (cons Γ₁ T)
extend sub e γ₂ = sub γ₂ , e (sub γ₂)

subType : Sub Γ₂ Γ₁ → Type Γ₁ → Type Γ₂
subType sub T = λ γ₂ → T (sub γ₂)

lift : (sub : Sub Γ₂ Γ₁) → (T : Type Γ₁) → Sub (cons Γ₂ (subType sub T)) (cons Γ₁ T)
lift sub T (γ , t) = sub γ , t

subTerm : (sub : Sub Γ₂ Γ₁) → Term Γ₁ T → Term Γ₂ (subType sub T)
subTerm sub e = λ γ₂ → e (sub γ₂)
```

There is no evident non-inductive way to isolate the renamings amongst these.

In contrast, in deep embeddings substitutions are usually given in an algebraic style, and defined inductively from the empty substitution, the identity, composition, weakening, and extension [12, §3.5] [7, §2] [1, §3]. Renamings may also be defined by induction-recursion—at the same time as a recursive function that interprets them as full substitutions [2, §5].

How is one to bridge this gap for deeper shallow embeddings? As before, the answer lies exactly in the middle: we are able to define a data type Ren of renamings, which is indexed in shallowly-embedded substitutions. Like with shallow embeddings but unlike with deep embeddings, renamings here have inherent computational content: they are actual functions mapping variables to variables.

```
Ren : S.Sub sΓ₂ sΓ₁ → Context sΓ₂ → Context sΓ₁ → Set
Ren sub Γ₂ Γ₁ = Var Γ₁ T t → Var Γ₂ (S.subType sub T) (S.subTerm sub t)

lift : Ren sub Γ₂ Γ₁ → Ren (S.lift sub T) (Γ₂ , S.subType sub T) (Γ₁ , T)

renTerm : Ren sub Γ₂ Γ₁ → Term Γ₁ T t → Term Γ₂ (S.subType sub T) (S.subTerm sub t)
renTerm ren (lambda e) = lambda (renTerm (lift ren) e)
renTerm ren (var x) = var (ren x)
renTerm ren (app e₁ e₂) = app (renTerm ren e₁) (renTerm ren e₂)
renTerm ren (Π A B) = Π (renTerm ren A) (renTerm (lift ren) B)
renTerm ren U = U
```

In many ways, our definition resembles the traditional definition of renaming in simply-typed λ-calculus [14, §II.1.1], i.e. a map from variables to variables that respects types. However, making that definition dependent involves quite a bit of tricky indexing [11, §5]. Fortunately, we have no need for that: our deepened renamings are also 'tracked' by a corresponding substitution of the shallow embedding, so we can use that instead.

Substitutions work similarly: a deepened substitution Sub is parameterized by a shallow substitution, i.e. an element of S.Sub. Interestingly, renaming is used to define substitution.

```
Sub : S.Sub sΓ₂ sΓ₁ → Context sΓ₂ → Context sΓ₁ → Set₁
Sub sub Γ₂ Γ₁ = Var Γ₁ T t → Term Γ₂ (S.subType sub T) (S.subTerm sub t)

liftSub : Sub sub Γ₂ Γ₁ → Sub (S.lift sub) (Γ₂ , S.subType sub T) (Γ₁ , T)
liftSub sub same = var same
liftSub sub (next x) = renTerm next (sub x)

extend : Sub sub Γ₂ Γ₁ → Term Γ₁ T t → Sub (S.extend sub t) Γ₂ (Γ₁ , T)
extend sub e same = subTerm sub e
extend sub e (next x) = sub x

subTerm : Sub sub Γ₂ Γ₁ → Term Γ₁ T t → Term Γ₂ (S.subType sub T) (S.subTerm sub t)
subTerm sub (lambda e) = lambda (subTerm (liftSub sub) e)
subTerm sub (var x) = sub x
subTerm sub (app e₁ e₂) = app (subTerm sub e₁) (subTerm sub e₂)
subTerm sub (Π A B) = Π (subTerm sub A) (subTerm (liftSub sub) B)
subTerm sub U = U
```

To sum up, with deeper shallow embeddings we can perform complicated *syntactic* operations on embedded terms, such as substitution and renaming. Naturally, one wonders: how far can we take this? For example, could we encode a β-reduction step? We answer that in the next section.

5 β-reduction and Injectivity of Products

With induction under our belts, it might seem that we have all of the components needed to define one-step β-reduction. After all, we can certainly inductively traverse a term looking for a λ-abstraction to the left of an application, and—if we find one—apply substitution as described in the previous section. Here is a first attempt:

```
βreduce : Term Γ T t → Term Γ T t
βreduce (lambda e) = lambda (βreduce e)
βreduce (var x) = var x
βreduce (Π A B) = Π (βreduce A) (βreduce B)
βreduce U = U
βreduce (app e₁ e₂) = ?
```

In fact, the completed clauses of this definition are not one-step β-reduction. It will perform more reductions than necessary, as e.g. we recurse on both the left and right subtree of Π. We could rework this definition into a correct one, but this simpler variant suffices to illustrate the point.

Given our syntax, the only computationally non-trivial case is that of app. To complete it we would ideally like to check whether the expression is a redex, i.e. whether e_1 is of the appropriate form lambda e. If not we can mindlessly recurse like in all other cases; but if we find a λ-abstraction, we would like to perform the substitution. Unfortunately, we are not able to pattern-match on e_1. The reason is that its type is Term Γ (S.Π A B) s_1. As Term is an inductive family indexed in shallow types, induction is only allowed when we have a term of general type Term Γ T s_1 with all of Gamma, T and s_1 free.

Being more precise about the problem we are trying to solve, we would like a function

```
castLam : Term Γ (S.Π A B) t → Maybe (Term (Γ , A) B (λ (γ , a) → t γ a))
```

that checks if its argument is a λ-abstraction, and if so returns its body. We can make some progress towards defining such a function by generalizing the Π type to an arbitrary type T. We may then induct on it; if it happens to be a λ-abstraction, we return a term of type Term (Γ , A') B' t' where $\mathtt{A}'$, $\mathtt{B}'$, and $\mathtt{t}'$ have no particular relation to the input.

In fact, we can do better than that. We can define a fairly simple function castLamImpl which checks whether its argument is a λ-abstraction. If it is, it returns

- types A' and B', the latter depending on the former,
- the body $\mathtt{t}'$ of the λ-abstraction,
- a proof that T ≡ S.Π A' B', and
- a proof that t is equal to the shallow-λ-abstraction of $\mathtt{t}'$.

All of this is encoded in an ugly nested Σ type, so here we abbreviate the syntax to convey the meaning:

```
castLamImpl : Term Γ T t → Maybe ∑_{A,B,t'} λγ → ( T γ , t γ) ≡ λγ →
  ( ( S.Π A B ) γ , λ a → t' (γ, a) ) × Term (Γ , A) B t'
castLamImpl (lambda e) = just (_ , _ , _ , refl , e)
castLamImpl _ = nothing
```

One might expect that it would be easy to define castLam using the proofs of equality provided by castLamImpl. But that is not so! If we apply castLamImpl to something of type Term Γ (S.Π A B) t we would obtain, amongst other things, types A' and B' along with a proof of S.Π A B ≡ S.Π A' B'. Hence, at the very least, we would have to show that $\mathtt{p} : \mathtt{A} \equiv \mathtt{A}'$ and $\mathtt{B} \equiv_{\mathtt{p}} \mathtt{B}'$, where the $\equiv_{\mathtt{p}}$ means that the two types are equal 'over' the equality p of the types on which they depend. In more detail, the exact statement we need is

```
Π-injective :
  (λ γ → ((S^T.Π A B) γ , λ a → t (γ , a))) ≡ (λ γ → ((S^T.Π A' B') γ , λ a → t' (γ , a)))
  → (A , B , t) ≡ (A' , B' , t')
```

If we had a proof of this, it would be possible to derive castLam from castLamImpl:

```
castLam : Term Γ (S.Π A B) t → Maybe (Term (Γ , A) B (λ (γ , a) → t γ a))
castLam e with castLamImpl e
... | nothing = nothing
... | just (A , B , t' , p , e') with (Π-injective p)
... | refl = just e'
```

Using that we could complete the final case of our β-reduction function:

```
βreduce (app e₁ e₂) with castLam e₁
... | nothing = app (βreduce e₁) (βreduce e₂)
... | just e = subTerm (extend idSub e₂) e
```

Unfortunately, Π-injective is a very long way from being true. This is a well-known issue in the metatheory of dependent types, known as the *injectivity of type constructors*. To begin, notice that given any Agda types A, B, C, D, the following statement is not true:

$$(A \to B) \equiv (C \to D) \quad \to \quad A \equiv C \times B \equiv D$$

Taking A and C to be empty types, B to be empty, and D to be the unit type of one element, we see that in homotopy type theory [23] the antecedent is true by univalence, yet the consequent proves that the empty and the unit types are equal. Thus, neither this, nor the corresponding more general statement about dependent function types in Agda, can be true. But since S.Π was defined in terms of Agda function types, neither will Π-injective!

Additionally, our shallow types are functions from a context to an Agda type. In order to prove equality of shallow types, we will therefore also require function extensionality.

5.1 Type codes: an Inductive-Recursive Universe

The problem we faced above boils down to two facts: (1) the types of the shallow embedding S were elements of the Agda universe of types Set, and (2) Set itself is far too *open*. This means that we are not in general able to induct on elements of Set (i.e. Agda types) so as to prove the requisite injectivity lemma. This restriction is only natural: as Agda supports new data type definitions, which would change the induction principle of Set.

The solution is to change the shallow embedding S, so that its types come from a *closed* universe. This can be achieved using a classic construction by Martin-Löf [20], which is sometimes known as the *inductive-recursive universe* [13, §1]. The idea is simple: instead of having a universe of types, we construct a universe of *type codes*, i.e. an inductive data type whose elements represent types. At the same time we define a family over this universe, which interprets these codes as types of the host language.

The technique itself is best illustrated by example. In fact, as our object language contains a universe itself, we will generate an infinite hierarchy of universes by simply adjoining an additional $\mathbb{N}$ parameter. This family TypeCode : $\mathbb{N} \to$ Set of inductive-recursive universes is defined as follows. Technically, as written here this definition is not strictly positive. However, the full definition in our formalization includes the standard trick of performing induction on the $\mathbb{N}$ parameter so that it is admissible in Agda.

```
data TypeCode : ℕ → Set where
  'U : TypeCode (suc n)
  'Π : (A : TypeCode n) → (⟦ A ⟧ → TypeCode n) → TypeCode n
  'lift : TypeCode n → TypeCode (suc n)

⟦_⟧ : TypeCode n → Set
⟦ 'U ⟧ = TypeCode n
⟦ 'Π A B ⟧ = (a : ⟦ A ⟧) → ⟦ B a ⟧
⟦ 'lift T ⟧ = ⟦ T ⟧
```

Notice the distinctive use of induction-recursion in the constructor 'Π, which takes a type code and a family over the *interpretation* of that type code.

We can then build a new shallow embedding whose types are, in fact, type codes. The construction amounts to sprinkling the decoding function wherever it should be to turn type codes into bona-fide types:

```
Ctx = Set
Type : ℕ → Ctx → Set
Type n Γ = Γ → TypeCode n
Term : ∀{n} → (Γ : Ctx) → Type n Γ → Set
Term Γ T = (γ : Γ) → ⟦ T γ ⟧
```

```
U : ∀{n} → Type (suc n) Γ
U = λ _ → 'U

Π : (A : Type (suc n) Γ)
  → Type (suc n) (cons Γ A) → Type (suc n) Γ
Π A B = λ γ → 'Π (A γ) ((λ a → B (γ , a)))
```

The implementation of Type is now a function from a context to TypeCode, while the implementation of Term uses the decoding function. Moreover, notice that types and terms are now indexed by $\mathbb{N}$, so they live at various levels of the inductive-recursive universe hierarchy.

Finally, we can deepen this new shallow embedding, not forgetting to thread the new universe levels around in the process. The new definition of Context, Var, and Term looks nearly identical to our original one from Figure 3, except that the prefix S now refers to our new shallow embedding with type codes.

5.2 Completing the Definition

The benefit of inductive-recursive universes is that, as they are inductively defined, their constructors are injective. As a consequence we are now able to prove the injectivity of 'Π:

```
Π-injective-typecode : (('Π A B) , t) ≡ (('Π A' B') , t') → (A , B , t) ≡ (A' , B' , t')
Π-injective-typecode refl = refl
```

This is almost a proof of Π-injective, with the difference that we must somehow extract proofs S.Π A B ≡ S.Π A' B' and t ≡ t' from its premise. Looking at the premise more carefully, it roughly gives us what we want, but as an equality of functions:

```
(λ γ → ((S.Π A B) γ , λ a → t (γ , a))) ≡ (λ γ → ((S.Π A' B') γ , λ a → t' (γ , a)))
```

By post-composing these two functions with the first projection, we can obtain an equality of the form

```
(λ γ → (S.Π A B) γ a) ≡ (λ γ → (S.Π A' B') γ)
```

To turn that into the desired equality, we must also use the *function extensionality* axiom:

```
funExt : ((x : A) → f x ≡ g x) → f ≡ g
```

which is not natively available in Agda. However, it is well-known to be consistent with intentional Martin-Löf type theory [23, §2].

To make the behavior of funExt somewhat more computational, we add the following *rewrite rule* to Agda:

```
postulate
  funExtElim : funExt (λ x → refl) ≡ refl
{-# REWRITE funExtElim #-}
```

This definitional equality is known to hold in e.g. the set-theoretic model. With this machinery in place, we are able to complete the definition of β−reduction.

6 Related Work

Our approach to deepening shallow embeddings is closely related to McBride's 'Outrageous but Meaningful Coincidences' [21]. In that work, McBride introduces a deeper shallow embedding for dependent type theory by wrapping it in a datatype. The embedding differs from ours in that instead of indexing deepened terms by their shallow counterparts, McBride writes an evaluator that targets the shallow embedding using induction-recursion. While the

aesthetics of each approach are debatable, it is a fact that the inductive-recursive interpreter makes it difficult to define syntactic operations like renaming and substitution. This is because any operations on the terms must include a proof that they commute with the evaluation function. Our purely inductive definition makes such syntactic transformations much simpler to define. Our work focuses on the engineering aspects of this technique, and its various practical uses and advantages over traditional shallow embeddings.

Kaposi et al. [18] propose that shallow embeddings are a 'morally correct' alternative to deep embeddings. They consider how one can be sure that a shallow embedding corresponds to the desired type theory; how does one know that no extra equalities or terms have been introduced? They prove several such correctness results *externally* to Agda. By contrast, our deeper shallow embeddings can be seen as bringing some of this work back inside of Agda. Kaposi et. al. also present a technique for term restriction in shallow embeddings, which we describe and compare to our techniques in (Section 3.3)

Of course, one way to build a maximally-expressive embedding of type theory in type theory is to actually use a deep embedding. Perhaps the most technically accomplished work in that direction is by Altenkirch and Kaposi [1], who use *quotient inductive-inductive types* (QIITs) to present a deep embedding of type theory (with explicit substitutions) that is already quotiented under its own definitional equality. The power of this technique has been shown by proving normalization results [2].

Previous work by Danielsson [12] and Chapman [7] encodes type theory (with explicit substitutions) in type theory itself. Because it lacks the technology of QIITs, this approach has to explicitly transport the representations of terms along type equalities using the type conversion rule.

The functional programming community has also explored various forms of embedding. Gibbons and Wu [15] present a unified approach through polymorphism and folding. Carette et al. [6] design a system using Haskell typeclasses to build shallow embeddings which simultaneously allow users to add new terms at any time but also to define new interpreters. Another instance of an 'intermediate' embedding that is between deep and shallow can be found in the work of Augustsson [4], where the author hints at the possibility of so-called *neritic* embeddings.

7 Conclusion

In this paper we presented a technique for *deepening* a shallow embedding by storing it in a data type. This allows us to inspect, analyze, and manipulate terms in a way that is usually associated with deep embeddings, while retaining the automation afforded by the shallow embedding. We demonstrated the practical uses of this technique in a series of small case studies, and showed how—assuming function extensionality—we can recover syntactic transformations, such as β-reduction.

One application of domain-specific languages is metaprogramming. In Lisp-like languages, code can be manipulated as data, quoted, and unquoted. Building a typed DSL can be seen as implementing a typed version of the sort of metaprogramming that Lisp can do. For example, quoting can be seen as simply writing an element of the DSL, and unquoting as corresponding to our extract functions. One could even imagine that the syntax of the DSL could look like the syntax of the host language, and therefore the host language would interpret code as either host code or DSL code depending on the type. Our hope is that ultimately dependently-typed embedding techniques will yield a typed (and therefore less error-prone) version of the sort of metaprogramming that one can do in Lisp.

Moreover, we explored the idea that there is more than just deep embeddings (initial models) and shallow embeddings (arbitrary models) in the design space. Unexplored constructions of shallow models harbor additional power, which might be of significant interest for particular applications. In the future, we are hoping to further explore this spectrum of 'hybrid' embeddings, with the ultimate goal of a practical approach for encoding a dependent type theory inside a dependent type theory.

Of course, the technique of deepening a shallow embedding has limitations. For example, deeper shallow embeddings allow induction over terms, but not over types. Various steps can be taken to improve that; for example, switching to an inductive-recursive universe allowed us to prove that Π is injective. However, the same trick does not allow us to prove e.g. that U and Π are unequal.[1]

Finally, it is often the case that we would like to compute syntactic transformations on the terms of an embedded language—like the β-reduction function given in §5—as they might prove useful in compilation, optimizations, etc. Thus, the question of expressivity arises: which syntactic transformations can be expressed over deeper shallow embeddings? We believe that most transformations that can be performed on a shallow embedding can be lifted to its deepened version: as shallow embeddings often 'quotient away' numerous differences between terms, any transformation on that level is likely easy to extend to deepened terms, but may require some effort. For example, when we defined renaming and substitution in Section 4, we had to define substitution on the shallow embedding first; when defining β-reduction in Section 5, we had to find a shallow embedding which satisfied certain equalities first, which required some ingenuity. In short, the transformations that can be expressed are not quite as expressive as those over deep embeddings, and may also require additional work.

Because of these restrictions, we view the main benefits of a deeper shallow embedding to be for metaprogramming, rather than for studying the metatheory of type theory.

References

1 Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 18–29. Association for Computing Machinery, 2016. doi:10.1145/2837614.2837638.

2 Thorsten Altenkirch and Ambrus Kaposi. Normalisation by Evaluation for Type Theory, in Type Theory. *Logical Methods in Computer Science*, 13(4), 2017. doi:10.23638/LMCS-13(4:1)2017.

3 Thorsten Altenkirch and Bernhard Reus. Monadic Presentations of Lambda Terms Using Generalized Inductive Types. In Jörg Flum and Mario Rodriguez-Artalejo, editors, *Computer Science Logic*, volume 1683 of *Lecture Notes in Computer Science*, pages 453–468, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. doi:10.1007/3-540-48168-0_32.

4 Lennart Augustsson. Making edsls fly. http://vimeo.com/73223479, 2012.

5 Richard J. Boulton, Andrew D. Gordon, Michael J. C. Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In Victoria Stavridou, Thomas F. Melham, and Raymond T. Boute, editors, *Theorem Provers in Circuit Design, Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience, Nijmegen, The Netherlands, 22-24 June 1992, Proceedings*, volume A-10 of *IFIP Transactions*, pages 129–156. North-Holland, 1992. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/11/EmbeddingPaper.pdf.

[1] As types are families of type codes over a context Γ, by funExt they are equal when $\Gamma = \bot$.

6 Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, 2009. doi:10.1017/S0956796809007205.

7 James Chapman. Type Theory Should Eat Itself. *Electronic Notes in Theoretical Computer Science*, 228:21–36, 2009. doi:10.1016/j.entcs.2008.12.114.

8 Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: A platform for high-level parametric hardware specification and its modular verification. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017. doi:10.1145/3110268.

9 Koen Claessen. *Embedded Languages for Describing and Verifying Hardware*. PhD thesis, Chalmers University of Technology, Gothenburg, Sweden, 2001. URL: http://publications.lib.chalmers.se/publication/636-embedded-languages-for-describing-and-verifying-hardware.

10 Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP*, 46, 01 2000. doi:10.1145/1988042.1988046.

11 Thierry Coquand. Canonicity and normalization for dependent type theory. *Theoretical Computer Science*, 777:184–191, 2019. doi:10.1016/j.tcs.2019.01.015.

12 Nils Anders Danielsson. A formalisation of a dependently typed language as an inductive-recursive family. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, volume 4502 of *Lecture Notes in Computer Science*, pages 93–109. Springer, 2006. doi:10.1007/978-3-540-74464-1_7.

13 Peter Dybjer and Anton Setzer. Indexed induction–recursion. *The Journal of Logic and Algebraic Programming*, 66(1):1–49, 2006. doi:10.1016/j.jlap.2005.07.001.

14 Marcelo Fiore. Semantic analysis of normalisation by evaluation for typed lambda calculus. In *Proceedings of the 4th ACM SIGPLAN international conference on Principles and Practice of Declarative Programming - PPDP '02*, pages 26–37. ACM Press, 2002. doi:10.1145/571157.571161.

15 Jeremy Gibbons and Nicolas Wu. Folding domain-specific languages: deep and shallow embeddings (functional pearl). In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 339–347. ACM, 2014. doi:10.1145/2628136.2628138.

16 Martin Hofmann. Syntax and Semantics of Dependent Types. In Andrew M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*, pages 79–130. Cambridge University Press, 1997. URL: https://www.tcs.ifi.lmu.de/mitarbeiter/martin-hofmann/pdfs/syntaxandsemanticsof-dependenttypes.pdf, doi:10.1017/CBO9780511526619.004.

17 Paul Hudak and Donya Quick. *The Haskell School of Music: From Signals to Symphonies*. Cambridge University Press, 2018. doi:10.1017/9781108241861.

18 Ambrus Kaposi, András Kovács, and Nicolai Kraus. Shallow Embedding of Type Theory is Morally Correct. In Graham Hutton, editor, *Mathematics of Program Construction*, volume 11825 of *Lecture Notes in Computer Science*, pages 329–365, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-33636-3_12.

19 Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. Beginner's Luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 114–129, 2017. URL: http://dl.acm.org/citation.cfm?id=3009868.

20 Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory*. Bibliopolis, 1984.

21 Conor McBride. Outrageous but meaningful coincidences: Dependent type-safe syntax and evaluation. In *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming*,

706 WGP '10, page 1–12, New York, NY, USA, 2010. Association for Computing Machinery.
707 doi:10.1145/1863495.1863497.
708 **22** Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and
709 Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality,
710 and recomputation in image processing pipelines. *SIGPLAN Not.*, 48(6):519–530, June 2013.
711 doi:10.1145/2499370.2462176.
712 **23** The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of*
713 *Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.
714 **24** Vasco T. Vasconcelos. Fundamentals of session types. *Information and Computation*, 217:52–70,
715 2012. doi:10.1016/j.ic.2012.05.002.
716 **25** Brent Yorgey. Diagrams: A declarative dsl for creating vector graphics. https://diagrams.
717 github.io/doc/manual.html, 2013.
718 **26** Christina Zeller and Ivan Perez. Mobile game programming in haskell. In Donya Quick and
719 Daniel Winograd-Cort, editors, *Proceedings of the 7th ACM SIGPLAN International Workshop*
720 *on Functional Art, Music, Modeling, and Design, FARM@ICFP 2019, Berlin, Germany,*
721 *August 18-23, 2019*, pages 37–48. ACM, 2019. doi:10.1145/3331543.3342580.