Contextual Embeddings: Implementing Bound Variables Through Instance Resolution

SAMANTHA FROHLICH, University of Bristol, United Kingdom
JESSICA FOSTER, University of Bristol, United Kingdom
G. A. KAVVOS, University of Bristol, United Kingdom
MENG WANG, University of Bristol, United Kingdom

Representing bound variables in embedded languages is a challenging problem, often requiring painful trade-offs between expressivity and usability. On the one hand, first-order representations using de Bruijn indices have many nice properties, but quickly become difficult to read and write. On the other hand, higher-order representations can piggy-back on the host language's binders to offer a more ergonomic interface, at a variety of costs depending on the technique. The current state-of-the-art is unembedding, i.e. a translation from the higher-order representation to the first-order and back again to get the best of both worlds. Unfortunately, the fact that this translation is type-safe relies on external metatheoretic arguments, holding unembedding back from its true potential. We solve this problem with a new embedding technique that uses instance resolution to define a context-directed isomorphism between an ergonomic higher-order interface and a first-order representation. Unlike previous techniques, this also applies to embedded languages with modal and substructural (e.g. linear) type systems, making unembedding relevant for modern languages.

CCS Concepts: • **Software and its engineering** → **Functional languages**; *Domain specific languages*; **Language features**; **Formal software verification**.

Additional Key Words and Phrases: embedded domain specific languages, binding, higher-order abstract syntax, interactive proof assistants

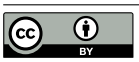
ACM Reference Format:
Samantha Frohlich, Jessica Foster, G. A. Kavvos, and Meng Wang. 2026. Contextual Embeddings: Implementing Bound Variables Through Instance Resolution. *Proc. ACM Program. Lang.* 10, PLDI, Article 191 (June 2026), 27 pages. https://doi.org/10.1145/3808269

1 Introduction

Programming languages are the tool we reach for to solve computational problems. We take pride in designing languages specialised to different tasks and joy in finding the level of abstraction that's just right for the task at hand. However, the implementation of a new language—no matter how small—is always a tiresome task: lexing, parsing, and interpreting can be an impediment to rapid prototyping. A well-known and effective solution to this problem is to *embed* the new (*guest* or *object*) language into an existing one (the *host* or *meta* language) [Hudak 1996].

One common difficulty is the embedding of *binding* constructs (those that abstract over variables, for example `let` or a λ-abstraction, increasing the language's expressivity). Implementing binders

Authors' Contact Information: Samantha Frohlich, University of Bristol, Bristol, United Kingdom, samantha.frohlich@bristol.ac.uk; Jessica Foster, University of Bristol, Bristol, United Kingdom, jess.foster@bristol.ac.uk; G. A. Kavvos, University of Bristol, Bristol, United Kingdom, alex.kavvos@bristol.ac.uk; Meng Wang, University of Bristol, Bristol, United Kingdom, meng.wang@bristol.ac.uk.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/6-ART191
https://doi.org/10.1145/3808269

from scratch can be cumbersome, due to fiddly issues such as *scope*, *name duplication*, and *capture-avoiding substitution* [Pfenning and Elliott 1988]. As such, embedding languages with binders is a well-explored area [Atkey 2009; Carette et al. 2009; Chlipala 2008; Despeyroux, Felty, et al. 1995; Despeyroux and Hirschowitz 1994; Despeyroux, Pfenning, et al. 1997; Huet and Lang 1978; Matsuda, Frohlich, et al. 2023; Washburn and Weirich 2008].

The problem of implementing binders for embedded languages is further complicated as interest extends beyond languages with *structural*—which support weakening, contraction, and exchange—to type systems to those where restrictions are placed on the usage of bound variables. In modal type systems, the types can specify when a variable can be used. This is useful for guiding the correct programming of staged computation [Davies and Pfenning 2001] and ensuring that data does not flow in an undesirable manner [Kavvos 2019; Tune et al. 2026]. In modal and substructural type systems, the types can specify how many times a variable can be used. Such systems are already used in influential languages like Rust [Matsakis and Klock 2014], which uses affine typing to control the use of mutable variables, and Haskell, which now has linear functions that ensure their input is used exactly once [Bernardy, Boespflug, et al. 2018]. Linear type systems in particular have become increasingly relevant as quantum computing develops, with qubits being non-clonable and non-discardable resources [Heunen and Vicary 2019, §4]. Hence, a modern embedding technique is required to support modal and substructural type systems.

The simplest approach to implement binders is a *first-order embedding* that explicitly represents binding, often using de Bruijn [1972] indices. For example, in Haskell we would write:

```
tt = Lambda (Lambda (Var One))  -- \xy.x, Church encoding of 'true'
ff = Lambda (Lambda (Var Zero)) -- \xy.y, Church encoding of 'false'
```

Unfortunately, de Bruijn indices (and first-order representations in general) are not particularly user-friendly. Explicit representations of binding are less intuitive to programmers, and error-prone. For example, consider how easy it would be to accidentally define `tt` instead of `ff`.

To solve this, functional programmers follow the ethos of reuse by developing Higher-Order Abstract Syntax (HOAS) [Huet and Lang 1978; Pfenning and Elliott 1988], which uses the host language's binders to implement guest binders. This technique is central to *logical frameworks* [Avron et al. 1987; Harper et al. 1993; Miller and Nadathur 2012].

Using HOAS we may encode the previous examples in a more ergonomic way:

```
data HOAS = Lam (HOAS -> HOAS) | App HOAS HOAS
tt = Lam (\x -> Lam (\y -> x)) :: HOAS  -- \xy.x, Church encoding of 'true'
ff = Lam (\x -> Lam (\y -> y)) :: HOAS  -- \xy.y, Church encoding of 'false'
```

However, HOAS can introduce a number of issues. For example, the simple implementation above admits *exotic terms*, i.e. elements that do not correspond to an embedded program. Additionally, HOAS directly supports only program transformations that can be defined as folds. However, in the back-end of the embedded language we often wish to express more complex program transformations, such as closure conversion or shrinking reduction optimisations, which are not easily written as folds [Atkey et al. 2009]. This prompts us to transform programs to a first-order representation, in a process called *unembedding* [Atkey et al. 2009]. Thus, the current state-of-the-art amounts to using HOAS in the front-end, which is then unembedded to obtain a more versatile first-order representation in the back-end.

The importance of unembedding as an implementation technique was highlighted by Matsuda, Frohlich, et al. [2023], who systematically applied it to languages with more complex semantics. For example, they used unembedding to embed HoBiT [Matsuda and Wang 2018], a bidirectional lens language that uses expressive binders instead of combinators. Previous implementations were challenging, relying on techniques developed specifically for each domain.

Implementing unembedding in a type-safe manner is surprisingly hard, even in languages with cutting-edge type systems [Grenrus 2025]. Current implementations [Atkey et al. 2009; Keuchel and Jeuring 2012; Matsuda, Frohlich, et al. 2023; McDonell et al. 2013] rely on externally justified operations (`fromJust`, `postulate`, `unsafeCoerce`, and a dynamic lookup, respectively) whose safety cannot be verified by the host language's type system. Atkey [2009] dedicates a whole paper to this proof, which is reused / adapted carefully by Matsuda, Frohlich, et al. [2023], McDonell et al. [2013], and Keuchel and Jeuring [2012]. Chlipala [2008, §3.1] encounters a similar problem in the setting of theorem provers with his work on Parametric Higher-Order Abstract Syntax (PHOAS): he finds it necessary to assert an additional axiom—an instance of *internal parametricity*—which is not readily available in current theorem provers.

This paper advances the state-of-the-art by presenting a new embedding technique which is evidently type-safe, allows unembedding, supports modal/substructural type systems, and incurs reasonable compile time and run time overheads. In summary:

- We present *Contextual Embeddings* (§3), a technique for embedding languages with binders that combines the precision of first-order representations (§2) with an ergonomic interface.
- We demonstrate the core ideas by embedding the Simply-Typed λ-Calculus (STLC) (§3.2).
- We justify the embedding with a proof that it is isomorphic to its first-order representation via a type-safe conversion, without requiring parametricity (§3.3).
- We argue that this is a flexible and applicable technique in any language with GADTs, type-level data (e.g. dependent types or Haskell DataKinds), and a suitable type class system (§3.5).
- We explore its performance (§3.6) and usability (§3.7), finding that it incurs reasonable costs, composes well, and does not require explicit type annotations.
- We argue that the technique is general by presenting embeddings of structural (§3.2), modal (§4), and substructural (§5) languages, *without* requiring modal or substructural features in the host language. To our knowledge, this is the first technique to encompass all of these categories.
- We develop a simplified version of our technique that can uniformly tackle all three categories of type systems (§6, §7, §8). The cost of this simplification is slightly reduced canonicity of the first-order embedding through the addition of some redundant type information, which we argue is a reasonable engineering solution.

2 Background: First-Order Embeddings

Contextual embeddings build upon first-order embeddings, so we begin with their description, trade-offs, and importance as a target for unembedding.

First-Order Embedding of the STLC. Fig. 1 shows a first-order embedding of the STLC[1], which uses a Generalised Algebraic Data Type (GADT) [Altenkirch and Reus 1999]. `Ty` represents the types of the guest language—in this case, a unit type (`Unit`) and a function type (`:->`). `Ctx` represents the guest language's typing context, which is a "snoc" list of types. Both of these are lifted to the type-level using Haskell's `DataKinds` extension to be used as parameters to our embedding.

The `Index ts t` type represents de Bruijn indices: the `Top` and `Pop` constructors are analogous to the `Zero` and `Succ` constructors of natural numbers. The constructor names refer to the type-level work they do, viz. ensuring that an `Index` represents a valid type `t` within the context `ts`. `Top` stands for the most recently bound variable, found at the 'top' of the context. `Pop` constructs an index that refers to a variable found deeper in the context by 'popping' off the most recently bound variable `t'`. For example: `Pop (Top :: Index (ts :/: a) a) :: Index (ts :/: a :/: b) a`

[1]The presentation of this paper will be in Haskell; see the artifact for examples in Lean and Agda.

```
data Ty = Unit | Ty :-> Ty
data Ctx = Empty | Ctx :/: Ty
data Index :: Ctx -> Ty -> Type where
  Top :: Index (ts :/: t) t
  Pop :: Index ts t
      -> Index (ts :/: t') t
```

```
data STLC :: Ctx -> Ty -> Type where
  Var :: Index ts t -> STLC ts t
  Lam  :: STLC (ts :/: t1) t2
       -> STLC ts (t1 :-> t2)
  App  :: STLC ts (t1 :-> t2)
       -> STLC ts t1
       -> STLC ts t2
  Star :: STLC ts Unit
```

Fig. 1. The Simply-Typed λ-Calculus (STLC), as a Generalised Algebraic Data Type (GADT)

`STLC` is a first-order embedding of the standard STLC with a unit type.[2] A term in context `ts` with type `t` can either be: a variable (`Var` holding an `Index`), a λ-abstraction (`Lam`), or an application (`App`). We also add `Star`, which introduces a term of type `Unit`. Following the standard type system of the STLC, `Lam` abstracts over an extended context to introduce a function, and `App` eliminates a function by applying it to its argument.

This first-order embedding is appealing for many reasons: it is canonical, *intrinsically* typed, supports "super-fold" semantics (semantics / intensional analyses including those that cannot be defined as folds); and can also be defined in dependent type theory.

Canonical and Intrinsically Typed. This embedding is one that contains exactly the information needed, no more and no less. Fig. 1 is exactly the STLC with no redundant information. Part of this arises from the fact it is an *intrinsically-typed embedding* [Reynolds 2003], i.e. one where the host's typing system only allows typed guest terms. This is desirable due to the benefits of static types and the embedding mantra of reusing host features.

Supports 'Super-fold' Semantics. A benefit of having an embedded language is that we can perform *metaprogramming*, i.e. use the host language to inspect, construct, optimise, and analyse programs of the guest language [Lilis and Savidis 2019]. Many such transformations cannot be easily expressed as a fold, e.g. shrinking reductions [Atkey et al. 2009].

Definable in Type Theory. A significant amount of work in programming language uses *dependent type theories*, either as a programming language (e.g. Idris [Brady 2013]) or as a *proof assistant* (e.g. Rocq [Coquand and Huet 1988], Lean [De Moura et al. 2015], and Agda [Norrell 2007]). A modern embedding technique should be compatible with dependent type theories. However, many of them only allow *strictly positive* inductive types; this first-order embedding is indeed one.

However, this embedding still isn't user-friendly; for example, the `tt` / `ff` confusion persists:

```
tt = Lam (Lam (Var (Pop Top))) -- \xy.x
ff = Lam (Lam (Var Top))       -- \xy.y
```

Dealbreaking Trade-offs. This trade-off between these compelling benefits and usability is not new. In fact, in his original paper de Bruijn [1972] gives three design criteria for embeddings:

(1) "easy to write and easy to read for the human reader"
(2) "easy to handle in metalingual discussion" (nowadays called *metatheory*)
(3) "easy for the computer and for the computer programmer"

He argues that this representation satisfies (2) and (3), but not (1). Guillemette and Monnier [2007] also discuss the trade-off of de Bruijn's representation, especially in comparison with a more user-friendly HOAS embedding with the goal of achieving a type-preserving closure conversion.

[2]We only have `Unit` here to keep the example simple; please see the artifact for an extended example with sum types.

Their experience concurs with the criteria of de Bruijn [1972] highlighting the tension between (1) and (3). The first-order embedding is simply best for performing closure conversion (or other global transformations / optimisations), but it compromises usability for embedded programming.

Rise of the Unembed. Unembedding [Atkey et al. 2009] is the act of translating from some embedding, typically a user-friendly one, into a first-order embedding. Its core is given by a function `unembed`, which performs this translation:

```
unembed :: Embedding -> STLC ts t
```

There is normally also an inverse to this function, which maps the first-order representation to the ergonomic one. This creates a setting in which either representation can be used, depending on needs and convenience.

Since the first-order representation is the gold standard in all aspects except usability, we place great importance on support for unembedding, and endeavour to advance the current state-of-the-art to make it evidently type-safe without reliance on complex external proofs.

3 Contextual Embeddings

In this section, we present Contextual Embeddings. We motivate their construction, and demonstrate their use on the STLC. We then justify the embedding by showing it to be isomorphic to the first-order embedding. Finally, we explore the practicality of the technique by demonstrating how to run expressions of a Contextual Embedding, and discussing the scope, performance, and usability of the technique.

3.1 Towards an Ergonomic First-Order Embedding

Our goal is to make the first-order embedding usable, but not to change it so much that unembedding becomes cumbersome or difficult to justify. The first change, which is inspired by HOAS, is to replace the body of λ-abstraction by a host language function:

```
data STLC1 :: Ctx -> Ty -> Type where
  Var1 :: Index ts a -> STLC1 ts a
  Lam1 :: (Index (ts :/: a) a -> STLC1 (ts :/: a) b) -> STLC1 ts (a :-> b)
```

This lets us write `ff` ergonomically by piggybacking on the host's binders:

```
ff = Lam1 (\x -> Lam1 (\y -> Var1 y)) :: STLC1 Empty (a :-> b :-> b)
```

However, there are two significant issues with this representation. To begin, it introduces *exotic terms* into the embedding, i.e. an element of the embedding's type that does *not* actually stand for a term of the guest language [Despeyroux, Felty, et al. 1995]. Consider for example the element `exotic`:

```
exotic :: STLC1 ts (a ':-> b)
exotic = Lam1 (\x -> case x of { Top -> ...; Pop _ -> ... })
```

Terms of the STLC cannot branch depending on the value of the bound variable's index! The cause of this exotic term is that the host language (in this case Haskell) supports pattern-matching on the `Index` data type. We would like to exclude such elements, as they introduce unexpected behaviour. Moreover, their presence is counter-productive towards our goal of usability since preventing errors is one of Molich and Nielsen [1990]'s heuristics of usability.

We exclude exotic terms by exchanging `Index` for a truncated version `ProxyTop`.

```
data ProxyTop :: Ctx -> Ty -> Type where
  ProxyTop :: ProxyTop (ts :/: t) t
```

`ProxyTop` is just like `Index`, but only has the `Top` constructor, allowing access only to the most recently bound variable. This does not allow exotic terms, as there is only one constructor on which we can pattern-match. Adopting this idea leads to the following version of the embedding:

```
data STLC2 :: Ctx -> Ty -> Type where
  Var2 :: ProxyTop ts a -> STLC2 ts a
  Lam2 :: (ProxyTop (ts :/: a) a -> STLC2 (ts :/: a) b) -> STLC2 ts (a :-> b)
notExotic = Lam2 $ \x -> case x of { ProxyTop -> Var2 x }
```

We can further guide users towards writing appropriate guest language terms by using the encapsulation mechanisms of the metalanguage, e.g. by making `ProxyTop` *opaque*, so that its constructors are not exported to user-facing modules (see §3.5 for more discussion).

However, this modified embedding only allows access to the most recently bound variable. For example, consider using it to define the Church boolean for 'true' ($\lambda xy.x$):

```
  tt = Lam2 (\x -> Lam2 (\y -> Var2 ???))
```

To fill in the hole we require a term of type `ProxyTop (('Empty ':/: a) ':/: b) a`. We would like to use `x`, but it has the type `ProxyTop ('Empty ':/: a) a` instead. Though the context can grow between the binding site and the use site of a variable, `ProxyTop` does not grow with it.

We can solve this if we have *evidence* that the context at the use site is an extension of the context at the binding site. We represent such evidence by a type `ExtE`. The constructor `Refl` evidences that the contexts are the same. The constructor `Step` evidences that an extension of the context has been extended by one more variable. The function `reify` can combine this evidence with the `ProxyTop` value to recover an appropriate `Index` at use site.

```
data ExtE :: Ctx -> Ctx -> Type where
  Refl  :: ExtE ts ts
  Step  :: ExtE ts ts' -> ExtE ts (ts' :/: t)
reify :: ExtE ts ts' -> ProxyTop ts t -> Index ts' t
reify Refl     ProxyTop = Top
reify (Step e) pt       = Pop (reify e pt)
```

Adding this evidence to the variable constructor allows access to variables bound deeper:

```
data STLC3 :: Ctx -> Ty -> Type where
  Var3  :: ExtE ts ts' -> ProxyTop ts t -> STLC3 ts' t
  Lam3  :: (ProxyTop (ts :/: a) a -> STLC3 (ts :/: a) b) -> STLC3 ts (a :-> b)
k' = Lam3 (\x -> Lam3 (\y -> Var3 (Step Refl) x))
```

Alas, that which was cast out through the door has returned by way of the window![3] The term `Var3 (Step Refl) x` is just `Var (Pop Top)` with extra steps, and de Bruijn indices have returned. But fear not! Finding the index can be *automated* through the host's *instance resolution* mechanism. We define a type class `ReifyIndex ts ts'` that features a (modified) `reify` function:

```
class ReifyIndex ts ts' where
   reify :: ProxyTop ts t -> Index ts' t
```

Next, we define instances of this class corresponding to the constructors of `ExtE`.

```
instance {-# OVERLAPPING #-} ReifyIndex ts ts where
  reify ProxyTop = Top
instance (ReifyIndex ts ts', ts'' ~ (ts' :/: a)) => ReifyIndex ts ts'' where
  reify i = Pop (reify i)
```

The first instance corresponds to the reflexive case, where the context is not extended and our `Index` is guaranteed to be `Top`. The second instance provides the `Step` case. It is defined recursively: assuming we can `reify` an `Index` from `ts` to `ts'`, then we can `reify` from `ts` to `(ts' :/: a)` by `Pop`ping. In other words, every 'snoc' corresponds to a `Pop`, until the contexts are equal. You can think of the `ProxyTop` as *saving a snapshot of the context at the bind site*, so that at the use site we can reify the correct `Index` via instance resolution.

[3] French proverb: "Chassé par la porte, il revient par la fenêtre."

Instance resolution in both Haskell and Agda has some nuances. The selection of an instance has to be *unambiguous* at every point. Thus, which instance is selected is based only on the *instance head*, not taking into account any of the constraints. Our two instance heads are `ReifyIndex ts ts` (i.e. the contexts are equal) and `ReifyIndex ts ts''` (i.e. the contexts are not *necessarily* equal, but they could be). These two instances overlap in the case where the contexts are equal, but since `ReifyIndex ts ts` is *strictly more specific* than `ReifyIndex ts ts''`, we can mark it as `OVERLAPPING` to make instance resolution unambiguous: if the contexts are equal, use the reflexive instance, otherwise, use the 'snoc'/`Step` instance.

Notice that even though we are calling it the 'snoc' instance, there's no 'snoc' in the instance head. Instead, we intentionally defer this 'snoc' check until *after* the instance has already been selected, which we do using the `ts'' ~ (ts' :/: a)` type equality constraint. This distinction is subtle, but it gives us more fine-grained control about how instances are *selected* vs. how they are *checked*. This will be especially important for our linear example later and the most general version of our technique. However, this trick also means that Haskell cannot tell that the recursive `ReifyIndex` constraint is structurally smaller, and so we must enable the `UndecidableInstances` extension. Unfortunately, this is justified externally: we are circumventing a check in the host language by appealing to reasoning that the compiler cannot verify. However, this argument is much simpler than the parametricity proof required by prior work [Atkey 2009]: the recursion is structurally recursive, since the `ts'' ~ (ts' :/: a)` constraint ensures the context shrinks by one element at each step. Moreover, should the argument prove wrong, the failure would manifest at compile time as non-terminating instance resolution rather than as a run time error.

3.2 Contextual Embedding of the STLC

Thus we arrive at the Contextual Embedding of the STLC.

```
data STLCCtx :: Ctx -> Ty -> Type where
  CVar  :: ReifyIndex ts ts' => ProxyTop ts a -> STLCCtx ts' a
  CLam  :: (ProxyTop (ts :/: a) a -> STLCCtx (ts :/: a) b) -> STLCCtx ts (a :-> b)
  CApp  :: STLCCtx ts (a :-> b) -> STLCCtx ts a -> STLCCtx ts b
  CStar :: STLCCtx ts Unit
```

The Contextual Embedding decorates the intrinsically typed first-order embedding by binding a strongly typed singleton object (`ProxyTop`) that snapshots the context at each variable's binding site and referring to this object whenever a guest variable is used. Because this gives us access to each variable's bind and use site contexts, we can infer each guest variable's de Bruijn index. Furthermore, this inference is outsourced to the host's instance search to keep the decoration non-invasive. This achieves guest code that is more readable (host names visually connect bind and use site occurrences of variables) and writeable (the instance resolution handles the de Bruijn indices) by humans, while preserving (2) and (3) of de Bruijn [1972]'s criteria with an embedding that is isomorphic §3.3 to the first-order embedding.

In short, this embedding preserves the benefits of the first-order embedding, while gaining usability. It is *intrinsically typed*, parameterised with contexts and types. It *supports 'super-fold' transformations* and is *definable in type theory*: it is represented as a GADT (in Haskell) or as a strictly positive, indexed inductive type (in dependent type theory), allowing definitions by pattern-matching. Finally, it is user-friendly: unlike the first-order embedding, variables have names, helping us define terms like `tt` and `ff` correctly:

```
tt = CLam (\x -> CLam (\y -> CVar x)) :: STLCCtx ts (a :-> b :-> a)
ff = CLam (\x -> CLam (\y -> CVar y)) :: STLCCtx ts (a :-> b :-> b)
```

3.3 Isomorphism to First-Order Embedding

```
unembed :: STLCCtx ts t -> STLC ts t
unembed (CVar i) = Var (reify i)
unembed (CLam e) = Lam
  (unembed (e ProxyTop))
unembed (CApp e1 e2)
  = App (unembed e1) (unembed e2)
unembed CStar = Star
```

```
contextualise :: STLC ts t -> STLCCtx ts t
contextualise (Var i) = toCVar i
contextualise (Lam e) = CLam
  (\_ -> contextualise e)
contextualise (App e1 e2)
  = CApp (contextualise e1)
         (contextualise e2)
contextualise Star = CStar
```

Fig. 2. Isomorphism between Contextual Embedding and first-order embedding

One of the greatest advantages of this embedding technique is that it supports type-safe unembedding. While this poses difficulties in HOAS-like embeddings, it is completely prosaic here (left of Fig. 2). Most of the heavy lifting is hidden in the `CVar` case, where the `ReifyIndex` instance and `ProxyTop` stored inside it are used to reify the de Bruijn index. The only other feature of interest is that every time we come across a binding construct, such as a λ-abstraction, we have to pass it (`ProxyTop :: ProxyTop (ts :/: t') t'`), enabling us to access the top variable. When this value reaches a variable, it is automatically reified to the right de Bruijn index.

We complement `unembed` with its inverse, `contextualise` (right of Fig. 2), which translates the first-order embedding back to the Contextual Embedding. There are two notable cases here: `Lam` and `Var`. In the `Lam` case, we ignore the variable. That might seem odd at first. To understand it, remember the purpose of this ignored value, the `ProxyTop`, is to enable the recovery of the de Bruijn index. However, in the case of `contextualise`, when we get to the `Var` case we *still have* the de Bruijn index, so the `ProxyTop` is indeed not needed here.

The `Var` case is the most complicated one. To understand how it works, it pays to notice that the data contained in the `CVar` is *existentially quantified*. Therefore, starting from an `Index ts' t` we need to produce a `ProxyTop` in a potentially smaller context `ts`, as well as an instance of `ReifyIndex ts ts'`. To achieve this we define an existential type that captures *just* this data.

```
data Var ts' t where MkVar :: ReifyIndex ts ts' => ProxyTop ts t -> Var ts' t
```

You can think of `Var` as a small Contextual Embedding with just variables. This allows us to consider the conversion of variables in isolation. The workhorse that carries it out is the function `fromIndex`.

```
fromIndex :: Index ts t -> Var ts t
fromIndex Top = varTop
  where varTop :: forall ts t. Var (ts :/: t) t
        varTop = MkVar (ProxyTop :: ProxyTop (ts :/: t) t)
fromIndex (Pop i) = weakenCVar (fromIndex i)
  where weakenCVar :: Var ts t -> Var (ts :/: t') t
        weakenCVar (MkVar p) = MkVar p
```

For every `Pop`, we weaken the context of the existential wrapper and thereby automatically weaken the `ReifyIndex` instance. For `Top`, we return `ProxyTop` in the existential wrapper. Here, `varTop` uses the `ScopedTypeVariables` language extension to specify the existential type; it would otherwise be ambiguous. Finally, we compose this conversion with a simple function that injects `Var` into the full Contextual Embedding (`STLCCtx`) to implement `toCVar`.

```
fromVar :: Var ts t -> STLCCtx ts t
fromVar (MkVar i) = CVar i
```

```
toCVar :: Index ts t -> STLCCtx ts t
toCVar = fromVar . fromIndex
```

We have proven that `unembed` and `contextualise` form an isomorphism in Lean 4 (included in artifact). `STLC` and `STLCCtx` correspond very closely, so the isomorphism proofs are almost entirely taken care of by inducting on the term. The core of the proof lies in the isomorphism between `Index` and `ProxyTop` with `ReifyIndex`. Once we have that, the rest follows without much trouble.

```
theorem isoL {ts:Ctx} {t:Ty} {e:STLC    ts t} : unembed (contextualise e) = e
```

```
theorem isoR {ts:Ctx} {t:Ty} {e:STLCCtx ts t} : contextualise (unembed e) = e
```

`indexIsoL`: `(reify . fromIndex)`. This lemma says that `reify` is the inverse of `fromIndex`. This follows by a straightforward induction on the index. Intuitively, for every `Pop` in the index, `fromIndex` weakens the `ProxyTop` by one, which `reify` then turns back into a `Pop`.

`indexIsoR`: `(fromIndex . reify)`. The opposite direction is more difficult because we no longer have a concrete index on which to induct. We have only a `ProxyTop` and a `ReifyIndex` instance. In practice, the instance (and therefore the index) is determined by the contexts, so you would hope that we could do some kind of induction on them, but sadly this is not immediately possible. In Haskell, Lean, and Agda, type classes are 'open world,' which means if we are given a `ReifyIndex ts ts` instance, we cannot say for certain that it is specifically the base case instance, because another instance for that type could be defined elsewhere. Therefore, we need a 'closed world' axiom which says 'if you give me a `ReifyIndex` instance, it must be one of our two defined instances'. We believe this is a reasonable assertion to make in practice because we can hide the type class from users by not exporting it. Armed with our 'closed world' axiom, we can now complete the proof in this direction by case analysis on the `ReifyIndex` instances.

Bringing this all together completes our proof. The interesting cases are: `isoL`'s `Var` case, which relies on `indexIsoL`; `isoR`'s `CVar` case, which relies on `indexIsoR`; and `isoR`'s `CLam` case, which requires function extensionality.

3.4 Assigning Semantics

Of course, for any embedded language, it is important to be able to assign semantics and run embedded code. This is particularly ergonomic with a Contextual Embedding due to the availability of the first-order embedding to expedite the process. We demonstrate this in Appendix A by assigning the standard model semantics (interpreting the STLC as a mapping from variable assignments to results) to our STLC example.

3.5 Scope of Contextual Embeddings

Host Requirements. Contextual Embeddings can be embedded into any host that supports GADTs, type-level data (in this case using Haskell's `DataKinds`), and has a suitable type class system. The key mechanism that is required is its ability to select instances based on type equality, such as in the base case of `ReifyIndex`, where both contexts are the same.[4]

De Bruijn Index Programming. Access to the `ProxyTop` constructor allows the user to program as if with de Bruijn indices: using it with a specified type is equivalent to specifying the de Bruijn Index (or more accurately, the de Bruijn level). Consider the following example:

```
id :: forall ts a. STLCCtx ts (a :-> a)
id = CLam (\x -> CVar (ProxyTop :: ProxyTop (ts:/:a) a))
```

This demonstrates how a programmer may avoid the benefits of using the host binder: instead of providing `x` to `CVar`, the function directly uses `ProxyTop`. Of course, programming this way is error-prone, so we recommend using a module boundary to hide this constructor. Any de Bruijn index programming is best left for the isomorphic de Bruijn representation.

Undefined and Non-terminating Terms. We exclude undefined and non-terminating terms from our analysis. That is, introducing these terms may or may not undermine the benefits of our technique and they should be considered a bug in all cases.

[4] See the artifact for additional examples in Agda and Lean.

Supported Metaprogramming. Contextual embeddings support metaprogramming through their simple unembedding. Moreover, polymorphism in embedded terms enable an additional form of parametric metaprogramming. For example, types such as `STLCCtx ts (a :-> a)` (with a universally quantified context `ts`) are allowed, and can be used to write combinators that construct larger programs, polymorphically in the context. Moreover, a theorem prover's access to the type information and ability to act based on that is another form of supported metaprogramming. All these forms of metaprograming are guaranteed to produce a well-formed terms.

3.6 Performance of Contextual Embeddings

Because our technique uses type class resolution, it incurs some compile time cost. However, the run time cost of converting between the Contextual Embedding and the first-order representation (via `unembed` and `contextualise`) is small.

To verify this, we performed some microbenchmarking.[5] We compared the Contextual Embedding of the STLC versus its first-order and HOAS counterparts in terms of compile time, and we measured the run time cost of `unembed` and `contextualise`. Our main benchmark is an STLC term consisting of an increasing number of lambdas, followed by returning the first bound variable:[6]

```
benchmark = CLam $ \y -> CLam $ \x -> ... CLam $ \x -> CVar y
```

We measured compilation time, and unembedding-contextualisation run time.

Compile-time Overheads. Our experiments show that there is a compile time overhead of the Contextual Embedding versus both the first-order embedding and the HOAS embedding. Fig. 3 and Fig. 4 show how compile time grows as the number of lambdas between the binding site and use site of our variable of interest increases. The tests were run with both the `-O1` and `-O0` optimisation flags. Fig. 3 compares our performance with the first-order representation. Fig. 4 compares our performance with the HOAS representation. For 100 nested lambdas, compile times stay below 1 second with `-O0` (the default for the Haskell Language Server), below the threshold of interrupting the user's flow of thought [Nielsen 1994].[7] Instance resolution costs are relatively negligible, as evidenced by the close compile times of Fig. 3. The first-order representation has no instance resolution or host lambdas, yet the compile times are on the same order of magnitude and scale similarly. The difference between our embedding and a HOAS embedding is caused by maintenance of the guest language's typing context (which has brought us benefits elsewhere).

Run-time Overheads. Appendix B shows a small run time cost for `unembed` and `contextualise` which grows linearly with the number of lambdas. This makes sense as Haskell types are erased, so there is no overhead from them at run time. Looking at the compiled code shows us what happens to the type classes and explains why this representation is so efficient. Consider `const3`:

```
const3 = CLam $ \x -> CLam $ \y -> CLam $ \z -> CVar x
```

When `const3` is compiled, the `CVar x` effectively compiles to:

```
CVar' (\p->Pop(Pop(case p of ProxyTop->Top))) x
```

where

```
CVar' :: (ProxyTop ts t -> Index ts' t) -> ProxyTop ts t -> STLCCtx ts' a
```

because once compiled, type classes correspond to carrying around a concrete instance of the contained functions (in this case `reify :: ProxyTop ts t -> Index ts' t`). Thus, at run time, unembedding amounts to the application of this function to x. In the other direction, contextualising

[5] These benchmarks were done on a machine with a 4.2 GHz Intel i5-3570K CPU, and with GHC 9.10.1 and cabal 3.12.1.0.

[6] `benchmark` could instead be written as `CLam (\y-> CLam (\x-> ... (CVar x)))`, without using `$`.

[7] Anecdotally, it appears that only with nested lambdas do compile times scale superlinearly. Compiling separate top-level lambdas seems to be roughly the sum of compiling them individually.

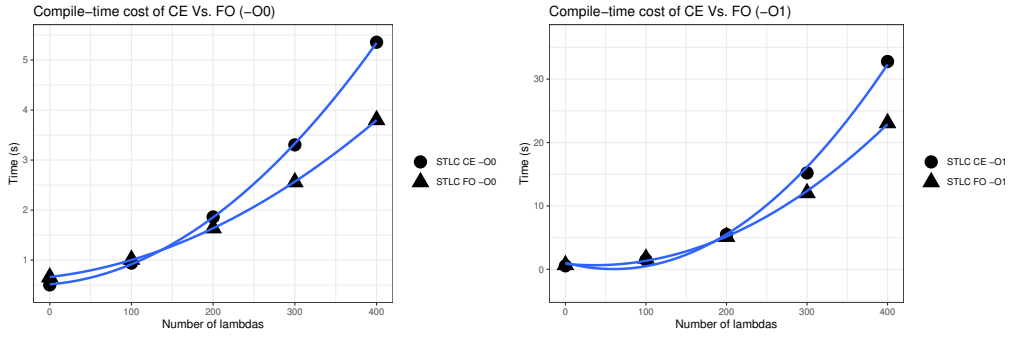


Fig. 3. Compile time: CE vs. FO

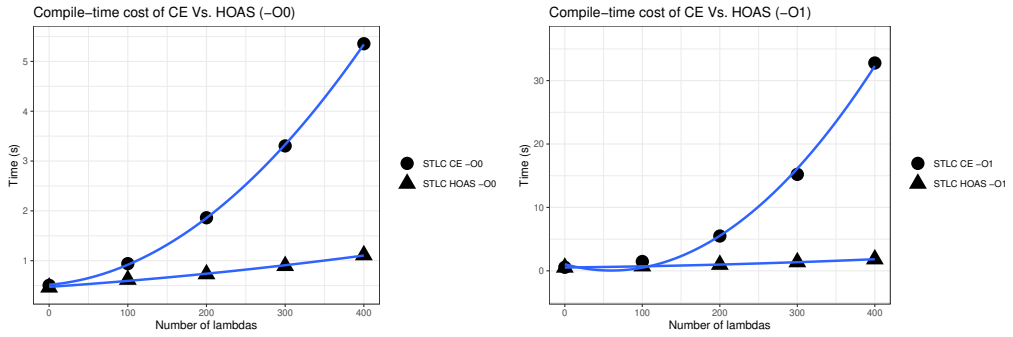


Fig. 4. Compile time: CE vs. HOAS

involves building up the `reify` function by recursing on the `Index`, which has a very simple structure and can therefore be executed quickly.

3.7 Usability of Contextual Embeddings

Type Inference. While most of our examples have type annotations, these are only there for readability: all of them can be inferred.[8] This includes the "levelled up" (§6) version of the technique. Further, it is the most general type which will be inferred. The only examples where the annotated type and the inferred type of a term would differ are `id` from §3.5, which uses explicitly scoped type variables, and `illTypedK2` later in this section, which is deliberately ill-typed.

Composability. Because our terms are polymorphic in the context, term fragments can be defined separately and combined without difficulty. For instance, it is straightforward to define 'smart constructors' that provide syntactic sugar:

```
lam2 k = CLam $ \x -> CLam $ \y -> k x y
const = lam2 $ \x y -> CVar x
```

No type annotations are required: the most general type is inferred. This flexibility is useful for building up libraries of reusable combinators on top of a Contextual Embedding.

[8]We should note this only applies for Haskell and Agda (which have very similar resolution mechanisms). Lean's instance resolution is far less principled, but still only requires top-level type annotations in most cases.

Reliability of Instance Resolution. The instance resolution at the heart of our technique is deliberately simple. At each step, the choice between the two instances is determined by whether the bind-site and use-site contexts match: if they do, the base case is selected; if the use-site context is larger, the recursive case is selected, and one element is peeled off. Since each recursive step makes the use-site context strictly smaller, resolution always terminates when it reaches the bind-site context. This is guaranteed to succeed because the bind-site context is always a prefix of the use-site context. In effect, the recursion discovers the difference between the two contexts, which is precisely the de Bruijn index. All remaining checks, e.g. that the context has the expected shape and that types match, are deferred constraints, verified only after the type checker has committed to an instance. Should any of these prove unsatisfiable, the resulting error is a type mismatch, not an instance resolution failure. This has pleasant consequences for the quality of error messages.

Quality of Error Messages. Error messages for contextually embedded terms are quite readable as Haskell error messages go. For example, consider this incorrectly typed term:

```
illTypedK2 :: STLCCtx ts (a :-> b :-> a)
illTypedK2 = CLam (\x -> CLam (\y -> CVar y))
```

The error message tells us the problem: `Couldn't match type 'b' with 'a'`.

Notice that this is *not* an instance resolution error. In fact, this type error is caught before instance search even occurs. Later, in §7, we shall see that even when instance search does happen for an ill-formed term, we still get a type error, not an instance resolution error.

Provability. For those doing metatheory on contextually embedded languages inside of a proof assistant, the advantage of our technique is the ability to specify proofs using named variables, but prove them using the de Bruijn representation. This relies on our isomorphism proofs, and therefore our 'closed world' axiom, and comes at the cost of switching back and forth between the representations. It is important to note that the inductive proofs themselves still take place on the first-order side; our technique improves the interface for stating theorems and writing readable examples, but does not change how the hard parts of mechanised metatheory are carried out.

4 Modal Example: Fitch-Style Modal Lambda Calculus

Moving on from the STLC, we present an embedding of a modal lambda calculus, which constitutes our first example of a non-structural language. The defining feature of modal languages is that they control when you can use variables. There are many examples of modal λ-calculi, including dual-context calculi [Davies and Pfenning 2001; Kavvos 2020] and Fitch-style calculi [Clouston 2018]. Both of these examples are amenable to a Contextual Embedding. The former is straightforward using our techniques, except that it uses two contexts. We shall therefore focus on the latter *Fitch-Style* modal λ-calculi, which have a more distinctively modal structure.

In structural languages like the STLC, the context grows as we move up a typing derivation, and elements in the context can be used freely at any point. Modal λ-calculi introduce additional controls on the use of variables. In Fitch-style modal λ-calculi, this is achieved by adding a modal type $\Box$, which we represent by the Haskell constructor `Box`.

```
data Ty = Unit | Ty :-> Ty | Box Ty
```

It is sometimes helpful to consider terms with box types as "global" terms: they are free to depend on other boxed terms, but not non-boxed terms, as these may depend on "locally" available data. To enforce this, the type of contexts is extended from that of the STLC with a lock.

```
data Ctx = Empty | Ctx :/: Ty | Lock Ctx
```

Indices are unchanged from the STLC and the first-order Fitch-Style Modal Lambda Calculus is embedded as follows:

```
data Fitch :: Ctx -> Ty -> Type where
  -- ...
  Shut :: Fitch (Lock ts) a -> Fitch ts (Box a)
  Open :: UnderLock ts ts' => Fitch ts' (Box a) -> Fitch ts a
class UnderLock (ts :: Ctx) (ts' :: Ctx) | ts -> ts' where
instance UnderLock (Lock ts) ts
instance UnderLock ts ts' => UnderLock (ts :/: a) ts'
```

`Var`, `Lam`, `App` and `Star` (elided) are unchanged from the STLC.[9] We focus instead on the new constructs `Shut` and `Open`, which introduce and eliminate boxed values respectively. The encapsulation of boxed values is managed by `Shut` who only allows them to arise from a locked context, ensuring that they cannot rely on non-`Box` values in the context. `Open` supervises safe unlocking. The only values that can be opened under a lock are boxed values. It does this by being very specific about its argument. Note that the argument's type is `(Fitch ts' (Box a))`, where `ts'` is specified to be under a lock by the type class constraint[10]. If the argument's type was `(Fitch ts' a)`, `Open` would not be doing its job properly because this would allow any value under a lock to be opened. The specificity of this argument type allows the host type system to ensure that the modality is properly respected. For example, `bad` would be disallowed by the type system. Here, `x` is added to the context with `Lam` and we shut access to it to introduce the modality. However, this term is bad because it tries to then use `x`, violating the encapsulation of boxed values.

```
bad :: Fitch ts (a :-> Box a)
bad = Lam {- x -} (Shut {- Lock -} (Var (Pop Top {- x -})))
```

Gratifyingly, the Contextual Embedding proceeds very similarly to that of the STLC demonstrating the generality of the technique. For completeness, here is the GADT of contextual terms:

```
data FitchCtx :: Ctx -> Ty -> Type where
  CVar  :: ReifyIndex ts ts' t => ProxyTop ts t -> FitchCtx ts' t
  CLam  :: (ProxyTop (ts :/: a) a -> FitchCtx (ts :/: a) b) -> FitchCtx ts (a :-> b)
  CApp  :: FitchCtx ts (a :-> b) -> FitchCtx ts a -> FitchCtx ts b
  CStar :: FitchCtx ts Unit
  CShut :: FitchCtx (Lock ts) a -> FitchCtx ts (Box a)
  COpen :: UnderLock ts ts' => FitchCtx ts' (Box a) -> FitchCtx ts a
```

`ProxyTop`, `unembed`, `contextualise`, and `ReifyIndex` are unchanged from their implementation for STLC. In the base case instance, `ProxyTop` ensures that the context is not locked. Likewise, in the also unchanged recursive instance, the check that ensured we only extend the context one variable at a time `(ts2 ~ (ts1 :/: a))` also handily ensures that we do not pop past a lock.

As before, the Contextual Embedding allows users to write terms with ease. For example, consider this term that is a *proof* of the K axiom of modal logic, i.e. $\Box(A \to B) \to \Box A \to \Box B$.

```
kaxiomCtx :: FitchCtx ts (Box (a :-> b) :-> Box a :-> Box b)
kaxiomCtx = CLam $ \f -> CLam $ \x -> CShut (CApp (COpen $ CVar f) (COpen $ CVar x))
```

5 Substructural Example: Linear λ-Calculus (LLC)

Substructural type systems control how many times you can use variables. In a linear type system, you must use each variable exactly once. This disallows expressions that clone variables (such as

[9] But note that `Ctx` *has* changed, giving `Pop:: Index ts t -> Index (ts :/: t') t` and `Top :: Index (ts :/: t) t` extra meaning: you cannot `Pop` past a lock or point to a lock with `Top`.

[10] `UnderLock` takes a context as the first variable (`ts`) and outputs the context 'under' the closest lock as the second variable (`ts'`). The functional dependency `ts -> ts'` ensures that `ts` uniquely determines `ts'`.

```
data Ty = Unit | Ty :* Ty | Ty :-* Ty
data Ctx = Empty | Ctx :/: CtxElem
data CtxElem = J Ty | N
data Index :: Ctx -> Ctx
           -> Ty -> Type where
  Top :: Index (ts :/: J t) (ts :/: N) t
  Pop :: Index ci co t
      -> Index (ci :/: a) (co :/: a) t
```

```
data LLC :: Ctx -> Ctx -> Ty -> Type where
  Var  :: Index ci co t -> LLC ci co t
  Lam  :: LLC (ci :/: J t1) (co :/: N) t2
       -> LLC ci co (t1 :-* t2)
  App  :: LLC ci ctx (t1 :-* t2)
       -> LLC ctx co t1 -> LLC ci co t2
  Star :: LLC ci ci Unit
  Pair :: LLC ci ctx t1 -> LLC ctx co t2
       -> LLC ci co (t1 :* t2)
  Letp :: LLC ci ctx (t1 :* t2)
       -> LLC (ctx :/: J t1 :/: J t2)
              (co  :/: N    :/: N   ) t
       -> LLC ci co t
```

Fig. 5. A first-order embedding of Linear λ-Calculus (LLC) as a GADT

$\lambda x.(x, x)$) and expressions that drop variables (such as $\lambda x.\lambda y.x$).

$$\frac{}{x : A \vdash x : A}\ \textsc{Var} \qquad \frac{\Delta_1 \vdash e_1 : A \multimap B \qquad \Delta_2 \vdash e_2 : A}{\Delta_1, \Delta_2 \vdash e_1 e_2 : B}\ \textsc{App}$$

Standard presentations of the LLC allow the context to be non-deterministically divided between sub-terms, as in the rule App [Polakow 2015], and enforce linearity by the variable rule Var, which is only applicable when the context consists of a single variable. This implicit non-determinism will not work with type class resolution. Instead, we opt for a deterministic presentation that features two contexts: an input and an output context [Grenrus 2019; Hodas 1995; Polakow 2015]. Elements of these contexts can either be types or placeholders representing 'consumed' values. Linearity is enforced by making sure that all values are used, i.e. the output context is empty.

For example, consider `varUsage`. This is a linear term ($\lambda x.x$) because the only variable in the input context (of type `J t`) is consumed, as indicated by the `N` replacing it in the output context. We represent this by defining contexts whose elements are either wrapped in a 'Just' need to be consumed), or are `N` to indicate that 'Nothing' is left (see Fig. 5). `Ctx` and `CtxElem`).

```
varUsage :: LLC (Empty :/: J t) (Empty :/: N) t
varUsage = Var Top
```

Fig. 5 shows the full Haskell embedding of the first-order LLC. It features a base type `Unit`, linear pairs (`:*`), and linear functions (`:-*`). Our `Index` type is updated to have two contexts and enforce the usage of the represented variable with `Top` matching on a switch from `J` to `N`. When it comes to embedded terms, `Lam` and our two new pair introduction and elimination constructs (`Pair` and `Letp`) are of interest. `Lam` enforces linearity by restricting the body of the lambda to use the newly introduced variable. When pairing two terms together, `Pair` carefully chains together the input and output contexts. These features of `Lam` and `Pair` are also found in `Letp`, as it also carefully chains contexts and enforces the usage of its bound variables.

Creating a Contextual Embedding for this language is more challenging than it seems at first. To appreciate why, consider the essence of the technique. We have been using `ProxyTop` to snapshot the context at the binding site, then at the use site we compare the contexts and recover the de Bruijn index with instance resolution. This works because the context always contains the information needed to recover the de Bruijn index. For example, in the case of the STLC, it was the difference in length between bind and use site contexts.

Of importance for considering LLC as a candidate for Contextual Embedding is how `ReifyIndex` is defined: in particular, its recursion pattern. Our base case is when the contexts are (type) equal (which works because the use site context is always an extension of the bind site context).

Consider now what the stopping point will be in the linear case. Matters here are complicated by two factors: there are now two contexts, and moreover these contexts may change between bind site and use site because variables could have been consumed (switching them from `J` to `N` in the context). For example, consider `app`:

```
app :: LLC ts ts ((a :-* b) :-* a :-* b)
app = Lam {- \f -} $ Lam {- \x -} $ App (Var {- f -} (Pop Top)) (Var {- x -} Top)

bx :: ProxyTop (ts :/: J (a :-* b) :/: J a) (ts :/: J (a :-* b) :/: N) a
ux :: LLC      (ts :/: N           :/: J a) (ts :/: N           :/: N) a
```

The use site of `x` is after `f` has been consumed, creating a difference between the bind site contexts of `x` (shown in `bx`) and the use site contexts of `x` (shown in `ux`). Notice how the context element corresponding to `f` (`J (a :-* b)`) is replaced by an `N`. This means that our stopping point can no longer be when the bind and use site contexts are equal, because they will not be equal.

It is possible to solve this by heavily modifying our instance resolution method, but this quickly becomes complex and very specialised to this particular language. Instead we will explore a simpler and more general version of our technique. Please see Appendix C for a sketch of the alternative solution, and our artifact for the implementation.

We will approach the more general solution by asking: what if instead of having to extract out the information we need each time, which must be done in a custom and specific way for each new type system, it was already separated out and directly accessible?

6 Contextual Embeddings, Levelled Up

The issue discussed in the previous section leads us to present a slightly extended kind of Contextual Embedding *with levels*. The key difference is that we make the information that we need to calculate the de Bruijn index more accessible. Previously, this type information was implicitly present in the typing context, and had to be extracted. If this information is accessible the type classes can be simplified to work in a uniform manner across different type systems.

We will specify exactly what this "levels" type information is, demonstrate a Contextual Embedding with Levels on the STLC, discuss the trade-offs, and revisit the LLC.

6.1 Contextual Embedding with Levels

To recover the de Bruijn indices, our instance mechanism essentially measured the number of additional bound variables between bind and use site contexts. We will now adapt our technique to work directly with context lengths. We shall call this information "levels" due to its similarity with *de Bruijn levels*, which count the depth of a variable from the top of a term.

Specifically, what we will refer to as the "level" is a natural number $l + n$, where l is the de Bruijn level, how many binders deep we are, and n is an arbitrary *offset*. This offset will be used to account for context polymorphism. For example, consider the two examples below. `levels` shows how the level increases as we go under binders with an empty outer context. Here, $n = 0$. `levels'` shows the same example again, but in a polymorphic outer context. Here, n is the length of `ts`.

```
levels :: STLCCtx Empty (a :-> b :-> b)
levels =
  CLam (\y -> {- l==1 -}
    CLam (\x -> {- l==2 -} CVar x))
```

```
levels' :: STLCCtx ts (a :-> b :-> b)
levels' =
  CLam (\y -> {- l==(length ts)+1 -}
    CLam (\x -> {- l==(length ts)+2 -} CVar x))
```

For our purposes, it does not matter what n is because what we care about is the relative difference in the level between bind site and use site.

```
data Nat = Z | S Nat
data IndexL
  :: Nat -> Ctx -> Ty -> Type where
  TopL :: IndexL (S l) (ts :/: t) t
  PopL :: IndexL l ts t
       -> IndexL (S l) (ts :/: x) t
```

```
data STLCL :: Nat -> Ctx -> Ty -> Type where
  VarL  :: IndexL l ts t -> STLCL l ts t
  LamL  :: STLCL (S l) (ts :/: a) b
        -> STLCL l ts (a :-> b)
  AppL  :: STLCL l ts (a :-> b)
        -> STLCL l ts a -> STLCL l ts b
  StarL :: STLCL l ts Unit
```

Fig. 6. A first-order embedding of the STLC with levels

```
class ReifyIndexL l l' ts t where
  reify :: ProxyTopL l t
        -> IndexL l' ts t
instance {-# OVERLAPPING #-}
  ( ts ~ (ts' :/: t))
  => ReifyIndexL l l ts t where
  reify ProxyTopL = TopL
instance ( l' ~ (S l'')
         , ts ~ (ts' :/: t')
         , ReifyIndexL l l'' ts' t)
  => ReifyIndexL l l' ts t where
  reify i = PopL (reify i)
```

```
data ProxyTopL :: Nat -> Ty -> Type where
  ProxyTopL :: ProxyTopL (S l) t
data STLCCtxL :: Nat -> Ctx -> Ty -> Type where
  CVarL  :: ReifyIndexL l l' ts t
         => ProxyTopL l t -> STLCCtxL l' ts t
  CLamL  :: ( ProxyTopL (S l) a
              -> STLCCtxL (S l) (ts :/: a) b)
         -> STLCCtxL l ts (a :-> b)
  CAppL  :: STLCCtxL l ts (a :-> b)
         -> STLCCtxL l ts a -> STLCCtxL l ts b
  CStarL :: STLCCtxL l ts Unit
```

Fig. 7. A Contextual Embedding of the STLC with levels

6.2 Contextual Embedding of the STLC with Levels

To make the levels accessible, they must be added to the first-order embedding as in Fig. 6. We represent the levels as a natural number, embedded in the standard way with the `Nat` data type. These levels then parameterise the index and term data types alongside the context and guest type. In `IndexL`, `TopL` ensures that we are at least one level deep, echoing how it checks that the typing context is non-empty, and asserting that the length of a non-empty context is at least one; `PopL` increments the level to keep it in correspondence with the length of the context. In `STLCL`, `LamL` increments the level for its body.

The Contextual Embedding (Fig. 7) also gets parameterised by the levels, but the main changes are in the `ProxyTop` and the huge simplification of the `ReifyIndexL` instances. Now, the `ProxyTop` just saves the level as opposed to the whole bind site context. This allows the logic of the type class instances to rely directly on the level instead extracting it from the context. The base case is when the levels are the same, and for every difference between the levels, we add a `PopL`.

The equality constraints on both instances ensure additional type safety. In the first instance, `ts ~ (ts' :/: t)` ensures that the correct type `t` (also saved by the `ProxyTop`) is indeed at the top of the context. In the second instance, `(l' ~ (S l''))` now joins `(ts ~ (ts' :/: t'))` in ensuring we only handle one `PopL` at a time. The elegance of this technique is the triviality of the instance resolution, which does not rely on these equality constraints (these are checked after), but on the type equality check seeing if the levels are the same (`ReifyIndexL l l ...`) or different (`ReifyIndexL l l' ...`).

6.3 Justification of Levels

While this simplification comes at the cost of redundant information in the first-order embedding, we argue this is a reasonable trade-off: all we have done is separated out what is already there for convenience. We can still unembed the Contextual Embedding to the canonical first-order

embedding (without the levels); and the levels buy us a simpler, more uniform and adaptable solution.

Unembedding. Unembedding from the Contextual Embedding with Levels to the first-order embedding with levels is defined identically to our previous unembedding function. If desired, the extra levels type information can then be forgotten to complete the translation to the canonical first-order embedding.

```
unembed:: STLCCtxL l ts t -> STLC ts t
unembed = forgetL . unembedL
unembedL :: STLCCtxL l ts t -> STLCL l ts t
unembedL (CVarL i) = VarL (reify i)
unembedL (CLamL e) =
  LamL (unembedL (e ProxyTopL))
unembedL (CAppL e1 e2) =
  AppL (unembedL e1) (unembedL e2)
unembedL CStarL = StarL
```

```
forgetL :: STLCL l ts t -> STLC ts t
forgetL (VarL x) = Var (forgetLI x)
forgetL (LamL b) = Lam (forgetL b)
forgetL (AppL f x)
  = App (forgetL f) (forgetL x)
forgetL StarL = Star
```

Uniform Solution. The uniformity of the solution is most evident when the Untyped λ-Calculus (ULC) is considered. Here the levels are not distilled, but are precisely the information we already have: there are no types in the context, just the level ensuring that nothing out of scope is referenced. The STLC builds upon this solution by adding the type information, but also retaining the level. Then, as you will see in our Contextual Embedding with Levels of the LLC, for a linear language with two contexts, we yet again add typing information separately instead of entangling it with the levels, which made them hard to extract.

```
data Index :: Nat -> Type where
  Top :: Index (S l)
  Pop :: Index l -> Index (S l)
```

```
data ULC :: Nat -> Type where
  Var  :: Index l -> ULC l
  Lam  :: ULC (S l) -> ULC l
  App  :: ULC l -> ULC l -> ULC l
  Star :: ULC l
```

Levelled Up Isomorphism. There still exists an isomorphism between the Contextual Embedding with Levels and the first-order embedding with levels. We have extended our Lean proof to show this for the STLC and LLC (which we shall present next). The proof extension is fairly trivial, with the proof structure remaining the same, just with some straightforward modifications for the different types and/or extra constructors.

7 Substructural Example: LLC Levelled Up

The Contextual Embedding with Levels of the LLC (Fig. 8) is constructed from the first-order embedding of the LLC following the same steps used for the STLC. First, the "level" type information is added. Terms and indices are both parameterised by levels, and levels are incremented by binders. Next, `ProxyTopL` is swapped in for the index type. It is just `ProxyTopL` from the STLC but with the top context elements exposed and enforced to go from `(J t)` to `N` for a `ProxyTopL` for type `t`. The key gain of levels is clear in `ReifyIndex`, because this works in exactly the same way as the ULC and STLC, there are just more contexts. The first base case instance still gets triggered by the bind and use site levels matching, there are just now two contexts that are checked to be non-empty. The tops of these contexts are also checked to be different because `i` should get used up and become `o` (`N`). The second type class, the recursive case, is still triggered by the levels not matching, adding a `PopL` per difference. While the number of type parameters may feel exuberant, they all play a role in the smooth and correct conversion from `ProxyTopL` to `IndexL` via instance resolution.

This embedding of the LLC enforces linearity and allows for ergonomic use of host binders:

```
pair :: LLCCtxL l ts ts (a :-* b :-* (b :* a))
```

```
data ProxyTopL :: Nat -> CtxElem -> CtxElem -> Ty -> Type where
  ProxyTopL :: ProxyTopL (S l) (J t) N t
class ReifyIndexL l l' i o ci' co' t | l l' i o ci' t -> co' where
  reify :: ProxyTopL l i o t -> IndexL l' ci' co' t
instance {-# OVERLAPPING #-} ( zs  ~ (zs' :/: i) , out ~ (zs' :/: o) )
  => ReifyIndexL l l i o zs out t where
  reify ProxyTopL = TopL
instance ( l' ~ (S l'') , zs  ~ (zs'  :/: z), out ~ (out' :/: z)
         , ReifyIndexL l l'' i o zs' out' t )
  => ReifyIndexL l l' i o zs out t where
  reify i = PopL (reify i)
data LLCCtxL :: Nat -> Ctx -> Ctx -> Ty -> Type where
  CVarL :: ReifyIndexL l l' i o ci' co' t => ProxyTopL l i o t -> LLCCtxL l' ci' co' t
  CLamL :: (ProxyTopL (S l) (J a) N a -> LLCCtxL (S l) (ci :/: J a) (co :/: N) b)
        -> LLCCtxL l ci co (a :-* b)
  CAppL  :: LLCCtxL l ci ctx (a :-* b) -> LLCCtxL l ctx co a -> LLCCtxL l ci co b
  CStarL :: LLCCtxL l ci ci Unit
  CPairL :: LLCCtxL l ci ctx a -> LLCCtxL l ctx co b -> LLCCtxL l ci co (a :* b)
  CLetpL :: LLCCtxL l ci ctx (a :* b)
         -> (ProxyTopL (S l) (J a) N a
              -> ProxyTopL (S (S l)) (J b) N b
              -> LLCCtxL (S (S l))
                  (ctx :/: J a :/: J b)
                  (co  :/: N   :/: N  ) t)
         -> LLCCtxL l ci co t
```

Fig. 8. A Contextual Embedding of the LLC with levels

```
pair = CLamL $ \x -> CLamL $ \y -> CPairL (CVarL y) (CVarL x)
```

And we still get nice error messages. Consider:

```
linearityError = CLamL $ \x -> CPairL (CVarL x) (CVarL x)
```

The linear variable `x` is used twice. The error message says: `Couldn't match type 'N' with 'J b' arising from a use of 'CVarL'`. Notably, instance search did occur here, but the error wasn't with the resolution itself (which is straight-forward thanks to the levels), it came from a type equality constraint/side condition generated by the search. It is these constraints which get to the real heart of the issue. In this case, that our variable `x` has already been used, hence the `N` instead of the expected `J b`.

8 Further Examples

Contextual Embeddings, especially once levelled up, are quite flexible in supporting different sorts of type systems. In this section, we shall consider a couple of variations including: supporting both linear and unlimited variables (this could be the basis of a mixed quantum and classical language); and numerically limiting how many times a variable can be used. Like the modal example of §4, a de Bruijn implementation of these examples must be done carefully, and requires domain specific knowledge: recall our use of the `UnderLock` type class to enforce correct use of the modality. Therefore, the focus of this section will be on sketching how the Contextual Embedding would need adjusted, assuming the existence of a sound de Bruijn implementation. We also plan to explore such implementations as future work.

Unlimited Variables. To support a mixture of linear and unlimited variables, the following adjustments to the Contextual Embedding would need to be made. For this setting, we would expect the guest types to have been extended. For example, `CtxElem` could have been extended with an unlimited case, perhaps: `U Ty`. To support this new context element, `Index` and `ProxyTop` will need extra constructors that allow unlimited variables to be used freely:

```
TopU :: Index (S m) (ts :/: U t) (ts :/: U t) t
ProxyTopU :: ProxyTop (S m) (U t) (U t) t
```

And the base case of `ReifyIndex` will need to feature a pattern-match to support this:

```
reify ProxyTopL = TopL
reify ProxyTopU = TopU
```

Now that `ProxyTop` has multiple constructors, they must not be exported from end-user facing modules, otherwise exotic terms could be constructed.

Numerically Limited Variables. This time, context elements may be annotated with a natural number indicating how many times they can be used: `L Nat Ty`. `Index` and `ProxyTop`'s top constructors can be adapted to decrease the natural number (instead of just switching from `J` to `N`) and ensure that no variable whose limit has reached zero is used.

```
TopL :: Index (S m) (ts :/: L (S n) t) (ts :/: L n t) t
ProxyTopL :: ProxyTop (S m) (L (S n) t) (L n t) t
```

9 Related Work

Reliance on External Arguments. A key difference between our work and previous work is reliance on external arguments. Prior work relies on a parametricity argument to justify the safety of unembedding [Atkey 2009; Keuchel and Jeuring 2012; Matsuda, Frohlich, et al. 2023; McDonell et al. 2013]. This argument is a paper-long proof, and has not been extended to a linear setting. Our implementation relies on the `UndecidableInstances` extension, which disables GHC's built-in check that instance resolution terminates. This is analogous in spirit: both circumvent a check in the host language by appealing to an argument that the compiler cannot verify. However, in our case termination is evident from inspection of the recursive instance, where the constraint `ts'' ~ (ts' :/: a)` ensures that the context shrinks by one at each step. Moreover, should the argument prove wrong, the failure would manifest at compile time as non-terminating instance resolution, rather than as a run time error. Our proof that `unembed` and `contextualise` form an isomorphism relies on the *closed world* assumption: that the only instances of `ReifyIndex` are the two we define. This can be ensured by not exporting the type class, as we recommend.

Higher-Order Abstract Syntax (HOAS) and Friends. Contextual Embeddings build on a series of research starting with Higher-Order Abstract Syntax (HOAS) [Huet and Lang 1978]. This introduced the trick of reusing the host language's binders, and still stands up in its original setting where pattern-matching on syntax is not a concern. Follow on research looks into using HOAS in other settings. Despeyroux, Felty, et al. [1995] allow it to be representable in theorem provers with "weakHOAS" (`ProxyTop` is inspired by this work), and Chlipala [2008] combines this with ideas from Washburn and Weirich [2008] for the current state-of-the-art. We directly advance this line of research by addressing the outstanding problem of type-safe unembedding. The key choice that achieves this is the accessibility of the concrete context as a type parameter of guest terms and our truncated index type `ProxyTop`. This differs from HOAS where the guest context is inaccessible because it has been handed off to the host language. This is made possible by utilising advances in host languages, namely type-level programming features, which allow us to manage contexts and

perform index reification automatically. We also bring this line of research into new domains by supporting modal and substructural type systems.

Embedding by Unembedding. Matsuda, Frohlich, et al. [2023] also present a technique for embedding languages with binders. They present a framework built upon Atkey et al. [2009]'s implementation of an isomorphism between de Bruijn and HOAS embedded terms. Their main contribution is seeing the value of unembedding for embedding languages with "interesting" semantic such as those in bidirectional or incremental domains. Contextual Embeddings are complementary to this work. Their interesting case studies can be reskinned to use a Contextual Embedding as a front-end, replacing the type-unsafe translation (justified by Atkey [2009]'s external proof) to their first-order semantics with a type-safe translation. Please see our artifact for an example of this reskinning. Regarding performance, our unembedding also all happens at compile time, as opposed to theirs which has to happen at run time. Our biggest advantage over this work is our support for modal and substructural languages, which they list as future work.

Names for Free (NFF). Bernardy and Pouillard [2013] present a first-order embedding with an ergonomic weakHOAS interface that uses parametricity to ensure that no mistakes are made due to accidentally using the wrong de Bruijn index. This work is similar in that it also uses a type class mechanism to automatically insert the correct index, though the mechanism is different in that they use parameterised variables as 'free names' and search through the context until the provided name matches the one in the context, rather than using the difference in levels to derive the index as we do. This is a very neat idea, although it inherently relies on incoherent instances, which can be tricky to get right if you want to do other fancy things with the type class system, as we do with substructural languages. Their paper only presents an untyped language, and while it is trivial to extend this to a typed setting, we found it non-trivial to unembed the language. This unembedding can actually be done in a type-safe manner, but this is not presented in their paper so our advance over this work amounts to a presentation of a type-safe unembedding and the support of modal and substructural languages.

Embedding a full Linear λ-Calculus (LLC) in Haskell. This work from Polakow [2015] is the only other work we have found that tackles the embedding of a non-structural language into a structural host. They embed the LLC into Haskell (without using its linear features). They present a HOAS embedding adorned with extra types to intrinsically enforce linearity of guest terms. This work is similar in that they embed the same presentation of the LLC with two contexts [Grenrus 2019; Hodas 1995] and the way they enforce linearity. It also features type-level `Nat`s which bear a superficial similarity to our 'levels', but are actually more similar to NFF, as they only use the `Nat`s as a source for fresh names. In our experiments, we found that these `Nat`s can be entirely replaced by NFF-style parameterised variables. Our contribution lies in the context of unembedding. When it comes to unembedding, this works suffers from the same problems as other HOAS embeddings, perhaps more acutely due to the linear setting. Unlike previous uses of unembedding, where Atkey [2009]'s proof can be appealed to, there exists no such proof in a linear setting.

Locally Nameless. The locally nameless representation of terms was introduced by Gordon [1994], and has been developed substantially for use in proof assistants by Aydemir, Charguéraud, et al. [2008]. The underlying mathematics of these embeddings has recently been developed by Pitts [2023]. See Charguéraud [2012] for a survey of the use of the technique in formalized metatheory. The idea of the locally nameless embedding is to mix techniques: while bound variables are represented with de Bruijn Indices, free variables are represented by *name*. This may be a string, but it may also come from any type with cofinite quantification (i.e. an infinite type where a fresh name distinct from a finite set can be drawn). As a result, they cannot avoid exotic terms: there

is no apparent way for static typing to constrain bound variables to be in-bounds [Charguéraud 2012, §2.4, 3.3]. Moreover, locally nameless representations do not seem amenable to static typing, and require external typing judgements. While locally nameless representations are an attractive technique for formalizing metatheory in a proof assistant, they do not appear to be a good interface for embedded programming.

Nominal Approaches. Nominal approaches [Pitts 2001] tackle names and binders with a method that is essentially a formalisation of the Barendregt's variable convention [1984]. The idea is that bound variable names are always fresh, avoiding issues arising from name duplication. The basis of this formalisation is Nominal Logic. While these approaches have achieved great success in systems such as nominal Isabelle [Urban and Tasson 2005], it is unclear how to implement them in settings like Haskell or dependently-typed proof assistants.

McBride's Classy Hack and Further Developments. A blog post from McBride [2010] sparked fruitful progress on the ergonomic construction of well-scoped de Bruijn terms, inspiring developments on the Agda mailing list [Kireev and Allais 2018] and in the Syntactic Haskell package [Axelsson 2025]. The core idea in all these developments is very similar to ours: keep the context information at type-level and use instance resolution (or take advantage of an Agda oddity in the case of Kireev and Allais) to allow users to use host binders by automatically injecting them into the correct de Bruijn representation of syntax. Notably, McBride's blog post applies the technique to a dependent type theory (the term language of Epigram), demonstrating its applicability beyond simply typed settings. Our work is a full realisation of this core idea, extending these developments with: implementations in multiple languages (Lean4, Agda, Haskell), unembedding, back-translation ("contextualisation", which was not covered by these works), proof of correctness, and of course our extension to apply this technique in modal and substructural domains.
McBride also has interesting work on a co-de-Bruijn representation of syntax [McBride 2018]. Unlike a de Bruijn representation, where irrelevant variables are discarded at the latest opportunity (at the leaves of the syntax tree); the co-de-Bruijn representation discards variables at the earliest opportunity. This grants efficiency gains by limiting term traversal, however, as McBride points out, it is "even less suited to human comprehension than de Bruijn syntax". This familiar trade-off makes exploring a Contextual Embedding based upon a co-de-Bruijn representation appealing future work. We also think that this approach might work particularly well in the linear setting.

Chlipala's Well-formedness. This work [Chlipala 2013, 2011] focuses on avoiding the need to assert an axiom to safely unembed PHOAS [Chlipala 2008]. For unembedding to be safe, the PHOAS term must be well-formed. A well-formed PHOAS term is one that only uses variables introduced by embedded binders. In theory, if a PHOAS term is truly parametric in its variable type, the parametricity should ensure the well-formedness of the term. However, the proof of this is not currently representable in the logic of today's theorem provers (even though we expect it to be true). So, in this work, instead of asserting this well-formedness for all terms, Chlipala [2013, 2011] asserts well-formedness individually for terms of interest. Requiring a proof of well-formedness for *dbify* (his version of `unembed`) indeed ensures the safety of the unembedding. Contextual Embeddings amount to baking this notion of well-formedness (expanded to also support open terms) into our higher-order representation of syntax. Thus direct parallels can be found between this work and ours. For example, in *dbify*, variables are instantiated as de Bruijn levels. This means that it works with the same logic as our `unembed`. The way we have constrained our higher-order syntax with `ReifyIndex` also ensures that contextually embedded terms are well-formed as specified by *wf* (his notion of well-formedness). We think that this agreement validates our approach, and again our extension to modal and substructural domains sets us apart.

Rebound. The recent Rebound work [De Santo and Weirich 2025] aids the creation of well-scoped and efficient term representations in Haskell. Of interest to us is their demonstration of McBride's classy hack and the Bernardy and Pouillard NFF technique to provide ergonomic front ends to their de Bruijn implementation. We believe this work to be complementary to ours, and that a Contextual front-end can be given to Rebound implemented terms. Of course, this would be limited to the scope of the library: it is Haskell-specific and while Rebound demonstrates handling of multiple binding sorts (e.g. System F), its core abstraction is single-sorted.

10 Conclusion and Future Work

Contextual Embeddings is a technique for embedding languages with binders that is as ergonomic as HOAS and advances the state-of-the-art in two key ways: support for type-safe unembedding, something that has long evaded prior work; and accommodation of languages with modern, non-structural type systems. Going forwards, we plan to build on this work as follows.

- Investigate how Contextual Embeddings interact with the POPLmark challenge [Aydemir, Bohannon, et al. 2005]. We expect the technique to improve the experience of stating theorems and writing test cases, though the core difficulty of the challenge lies in inductive proofs about binding, which would still be carried out on the first-order representation.
- Explore a co-de-Bruijn [McBride 2018] based Contextual Embedding.
- Due to the prosaic nature of the `unembed` function, it is natural to think about automating its definition. In Haskell, this could be done with TemplateHaskell.
- Quantum languages are necessarily linear because of the no-cloning theorem. This makes them an attractive target for Contextual Embeddings.
- One could consider our technique as a form of *metaprogramming*, i.e. using type class resolution to construct programs. This pattern could be applied to other domains.

Data Availability Statement

Our artifact is available at https://doi.org/10.5281/zenodo.19081887. It contains Contextual Embeddings in Haskell, Lean and Agda, the benchmarks, and the Lean proof.

Acknowledgments

We would like to thank Eddie Jones, Kazutaka Matsuda, and Robert Atkey for their crucial advice and feedback. This work was supported by the UKRI Engineering and Physical Sciences Research Council (EPSRC) grants EP/T008911/1, EP/Y000242/1 and EP/Y033418/1, the UKRI International Science Partnerships Fund (ISPF), a Royal Society Research Grant, and a grant from the Advanced Research + Invention Agency (ARIA).

A Assigning Semantics

The first step of assigning semantics is defining a mapping from guest language types to host language types so that the host can evaluate the guest's semantics. In Haskell, this can be done with the following type family.

```
type family InterpretType (ty :: Ty) where
  InterpretType Unit      = ()
  InterpretType (a :-> b) = (InterpretType a) -> (InterpretType b)
```

This is a type level function that pattern-matches on the guest types and translates them to the corresponding host types, using recursion for function types.

We also extend this mapping for contexts (type-level lists), applying `InterpType` pointwise.

```
type family InterpretCtx (as :: Ctx) where
  InterpretCtx Empty      = '[]
```

```
  InterpretCtx (as :/: a) = InterpretType a : InterpretCtx as
```

Now we can define the standard model semantics. Concretely, this (`StdModel`) will be a function from a heterogeneous list (`VEnv`) assigning a value for each of the types in the context to a result. Notice the use of the type families for interpreting the guest types `ts` and `t`. This is essential for the host to execute the `StdModel` function.

```
type StdModel ts t = VEnv (InterpretCtx ts) -> InterpretType t

data VEnv as where
  ENil  :: VEnv '[]
  ECons :: a -> VEnv as -> VEnv (a ': as)
```

Assigning these semantics to the Contextual Embedding of the STLC (`STLCCtx`), can be expedited by utilising `unembed`. This simplifies the assignment of semantics to a mapping from the more manageable De Bruijn representation of syntax (`STLC`) to the standard model semantics.

```
interp :: STLCCtx ts t -> StdModel ts t
interp = interpDB . unembed

interpDB :: STLC ts t -> StdModel ts t
interpDB (Var i)   env = lookEnv env i
interpDB (Lam f)   env = \x -> interpDB f (consEnv x env)
interpDB (App f x) env = (interpDB f env) (interpDB x env)
interpDB (Star)    env = ()
```

This mapping (`interpDB`) is straightforward, with `lookEnv` looking up values in an environment using an index and `consEnv` and adding values to an environment.

B Runtime Cost

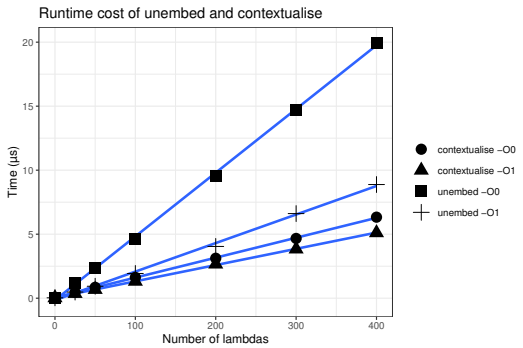


Fig. 9. Run-time cost of unembed and `contextualise`

C Alternative Solution to Levels for LLC

Here, we sketch how to contextually embed LLC without the levelled up technique. Please see our artifact for the full implementation. Warning, this implementation is complex and specialised, hence our decision to generalise our technique and use the levels.

The key insight is that for any `CVar` sub-expression, exactly one element of the context should switch from `J` to `N`. This leaves our type classes with two tasks: inserting this switch in the correct place, and generating the correct index at the use site. The first task is achieved by a type class that uses both bind site contexts and the use site input context to calculate the correct use site output context. The difference in length between either bind site context and the use site context will give the position of where the switch from `J` to `N` should be. The rest of the output context is just the same as the input context. With this in place, our recursion pattern for calculating the correct use site index can use this inserted outermost switch from `J` to `N` as the stopping point, because any outer elements will exactly match the input context as specified by the first type class. Thus, if the top elements of the context are the same, we add a `Pop`. Then when we hit the switch, this is our base case and we add the `Top`.

Thus the implementation of this is quite involved. It requires one incoherent instance, which is something that must be done with great care to ensure correctness.

References

Thorsten Altenkirch and Bernhard Reus. 1999. "Monadic Presentations of Lambda Terms Using Generalized Inductive Types." In: *Computer Science Logic* (Lecture Notes in Computer Science). Vol. 1683. Springer Berlin Heidelberg, 453–468. doi:10.1007/3-540-48168-0_32.

Robert Atkey. 2009. "Syntax for free: representing syntax with binding using parametricity." In: *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009. Proceedings* (Lecture Notes in Computer Science). Ed. by Pierre-Louis Curien. Vol. 5608. Springer, 35–49. doi:10.1007/978-3-642-02273-9_5.

Robert Atkey, Sam Lindley, and Jeremy Yallop. 2009. "Unembedding domain-specific languages." In: *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009*, 37–48. doi:10.1145/1596638.1596644.

Arnon Avron, Furio Honsell, and Ian Mason. 1987. *Using Typed Lambda Calculus to Implement Formal Systems on a Machine.* ECS-LFCS-87-31. Laboratory for Foundations of Computer Science, University of Edinburgh. https://www.lfcs.inf.ed.ac.uk/reports/87/ECS-LFCS-87-31/.

Emil Axelsson. 2025. *syntactic-3.8.4: Generic representation and manipulation of abstract syntax.* Retrieved July 4, 2025 from https://hackage.haskell.org/package/syntactic-3.8.4/docs/Language-Syntactic-Functional-WellScoped.html.

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. "Engineering formal metatheory." In: *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Association for Computing Machinery, 3–15. doi:10.1145/1328438.1328443.

Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. "Mechanized Metatheory for the Masses: The POPLMark Challenge." In: *Theorem Proving in Higher Order Logics* (Lecture Notes in Computer Science). Ed. by Joe Hurd and Tom Melham. Vol. 3603. Springer Berlin Heidelberg, Berlin, Heidelberg, 50–65. doi:10.1007/11541868_4.

Henk Barendregt. 1984. *Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam. ISBN: 978-0-444-87508-2.

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. "Linear Haskell: practical linearity in a higher-order polymorphic language." *Proceedings of the ACM on Programming Languages*, 2, 1–29, POPL. doi:10.1145/3158093.

Jean-Philippe Bernardy and Nicolas Pouillard. 2013. "Names for free: polymorphic views of names and binders." In: *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell*. Association for Computing Machinery, 13–24. doi:10.1145/2503778.2503780.

Edwin Brady. 2013. "Idris, a general-purpose dependently typed programming language: Design and implementation." *Journal of Functional Programming*, 23, 5, 552–593. doi:10.1017/S095679681300018X.

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

Arthur Charguéraud. 2012. "The Locally Nameless Representation." *Journal of Automated Reasoning*, 49, 3, 363–408. doi:10.1007/s10817-011-9225-2.

Adam Chlipala. 2013. *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant*. MIT Press. doi:10.7551/mitpress/9153.001.0001.

Adam Chlipala. 2011. *Intensional*. Retrieved July 7, 2025 from http://adam.chlipala.net/cpdt/html/Intensional.html.

Adam Chlipala. 2008. "Parametric higher-order abstract syntax for mechanized semantics." In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 143–156. doi:10.1145/1411204.1411226.

Ranald Clouston. 2018. "Fitch-Style Modal Lambda Calculi." In: *Foundations of Software Science and Computation Structures* (Lecture Notes in Computer Science). Ed. by Christel Baier and Ugo Dal Lago. Vol. 10803. Springer International Publishing, Cham, 258–275. doi:10.1007/978-3-319-89366-2_14.

Thierry Coquand and Gérard Huet. 1988. "The calculus of constructions." *Information and Computation*, 76, 2, 95–120. doi:10.1016/0890-5401(88)90005-3.

Rowan Davies and Frank Pfenning. 2001. "A modal analysis of staged computation." *Journal of the ACM*, 48, 3, 555–604. doi:10.1145/382780.382785.

N.G. de Bruijn. 1972. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem." *Indagationes Mathematicae (Proceedings)*, 75, 5, 381–392. doi:10.1016/1385-7258(72)90034-0.

Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob Von Raumer. 2015. "The Lean Theorem Prover (System Description)." In: *Automated Deduction - CADE-25* (Lecture Notes in Computer Science) 9195. Ed. by Amy P. Felty and Aart Middeldorp. Springer International Publishing, Cham, 378–388. doi:10.1007/978-3-319-21401-6_26.

Noé De Santo and Stephanie Weirich. 2025. "Rebound: Efficient, Expressive, and Well-Scoped Binding." In: *Proceedings of the 18th ACM SIGPLAN International Haskell Symposium* (Haskell '25). Association for Computing Machinery, Singapore, Singapore, 38–52. ISBN: 9798400721472. doi:10.1145/3759164.3759348.

Joëlle Despeyroux, Amy Felty, and André Hirschowitz. 1995. "Higher-order abstract syntax in Coq." In: *Typed Lambda Calculi and Applications* (Lecture Notes in Computer Science). Ed. by Mariangiola Dezani-Ciancaglini and Gordon Plotkin. Vol. 902. Springer Berlin Heidelberg, Berlin, Heidelberg, 124–138. doi:10.1007/BFb0014049.

Joëlle Despeyroux and André Hirschowitz. 1994. "Higher-order abstract syntax with induction in Coq." In: *Logic Programming and Automated Reasoning* (Lecture Notes in Computer Science). Ed. by Frank Pfenning. Vol. 822. Springer Berlin Heidelberg, Berlin, Heidelberg, 159–173. doi:10.1007/3-540-58216-9_36.

Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. 1997. "Primitive recursion for higher-order abstract syntax." In: *Typed Lambda Calculi and Applications* (Lecture Notes in Computer Science). Ed. by Philippe Groote and J. Roger Hindley. Vol. 1210. Springer Berlin Heidelberg, 147–163. doi:10.1007/3-540-62688-3_34.

Andrew D. Gordon. 1994. "A mechanisation of name-carrying syntax up to alpha-conversion." In: *Higher Order Logic Theorem Proving and Its Applications* (Lecture Notes in Computer Science). Ed. by Jeffrey J. Joyce and Carl-Johan H. Seger. Vol. 780. Springer Berlin Heidelberg, 413–425. doi:10.1007/3-540-57826-9_152.

Oleg Grenrus. 2025. *PHOAS to de Bruijn conversion*. Retrieved June 27, 2025 from https://oleg.fi/gists/posts/2025-02-13-phoas-to-db.html.

Oleg Grenrus. 2019. *Typed tagless-final interpreters for Linear Lambda Calculus de Bruijn indices*. Retrieved July 3, 2025 from https://okmij.org/ftp/tagless-final/course/LinearLC.hs.

Louis-Julien Guillemette and Stefan Monnier. 2007. "A type-preserving closure conversion in haskell." In: *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop* (Haskell '07). Association for Computing Machinery, Freiburg, Germany, 83–92. ISBN: 9781595936745. doi:10.1145/1291201.1291212.

Robert Harper, Gordon D. Plotkin, and Furio Honsell. 1993. "A framework for defining logics." *Journal of the ACM*, 40, 1, 143–184. doi:10.1145/138027.138060.

Chris Heunen and Jamie Vicary. 2019. *Categories for Quantum Theory: An Introduction*. Oxford University Press. doi:10.1093/oso/9780198739623.001.0001.

Joshua Seth Hodas. 1995. "Logic programming in intuitionistic linear logic: theory, design, and implementation." Ph.D. Dissertation. USA. UMI Order No. GAX94-27546.

Paul Hudak. 1996. "Building domain-specific embedded languages." *ACM Computing Surveys*, 28, 4es. doi:10.1145/242224.242477.

Gérard Huet and Bernard Lang. 1978. "Proving and applying program transformations expressed with second-order patterns." *Acta Informatica*, 11, 31–55. doi:10.1007/BF00264598.

G. A. Kavvos. 2020. "Dual-Context Calculi for Modal Logic." *Logical Methods in Computer Science*, 16, 3. doi:10.23638/LMCS-16(3:10)2020.

G. A. Kavvos. 2019. "Modalities, cohesion, and information flow." *Proceedings of the ACM on Programming Languages*, 3, POPL. doi:10.1145/3290333.

Steven Keuchel and Johan T. Jeuring. 2012. "Generic conversions of abstract syntax representations." In: *Proceedings of the 8th ACM SIGPLAN Workshop on Generic Programming* (WGP '12). Association for Computing Machinery, Copenhagen, Denmark, 57–68. ISBN: 9781450315760. doi:10.1145/2364394.2364403.

Roman Kireev and Guillaume Allais. 2018. *Typed Jigger in vanilla Agda*. Retrieved Mar. 20, 2026 from https://agda.chalmers.narkive.com/8CbnEcum/typed-jigger-in-vanilla.

Yannis Lilis and Anthony Savidis. 2019. "A survey of metaprogramming languages." *ACM Computing Surveys*, 52, 6, Article 113, 39 pages. doi:10.1145/3354584.

Nicholas D. Matsakis and Felix S. Klock. 2014. "The Rust Language." *ACM SIGAda Ada Letters*, 34, 3, 103–104. doi:10.1145/2692956.2663188.

Kazutaka Matsuda, Samantha Frohlich, Meng Wang, and Nicolas Wu. 2023. "Embedding by unembedding." *Proceedings of the ACM on Programming Languages*, 7, ICFP, 1–47. doi:10.1145/3607830.

Kazutaka Matsuda and Meng Wang. 2018. "HOBiT: Programming Lenses Without Using Lens Combinators." In: *Programming Languages and Systems* (Lecture Notes in Computer Science). Ed. by Amal Ahmed. Vol. 10801. Springer International Publishing, Cham, 31–59. doi:10.1007/978-3-319-89884-1_2.

Conor McBride. 2018. "Everybody's Got To Be Somewhere." *Electronic Proceedings in Theoretical Computer Science*, 275, 53–69. doi:10.4204/eptcs.275.6.

Conor McBride. 2010. *I am not a number, I am a classy hack*. Retrieved Feb. 23, 2025 from %7Bhttps://mazzo.li/epilogue/index.html%3Fp=773.html%7D.

Trevor L. McDonell, Manuel M.T. Chakravarty, Gabriele Keller, and Ben Lippmeier. 2013. "Optimising purely functional GPU programs." In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (ICFP '13). Association for Computing Machinery, Boston, Massachusetts, USA, 49–60. ISBN: 9781450323260. doi:10.1145/2500365.2500595.

Dale Miller and Gopalan Nadathur. 2012. *Programming with higher-order logic*. Cambridge University Press, New York, NY. 306 pp. doi:10.1017/CBO9781139021326.

Rolf Molich and Jakob Nielsen. 1990. "Improving a human-computer dialogue." *Communications of the ACM*, 33, 3, 338–348. doi:10.1145/77481.77486.

Jakob Nielsen. 1994. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN: 9780080520292.

Ulf Norrell. 2007. "Towards a practical programming language based on dependent type theory." PhD thesis. Chalmers University of Technology. https://research.chalmers.se/en/publication/46311.

Frank Pfenning and Conal Elliott. 1988. "Higher-order abstract syntax." In: *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, 199–208. doi:10.1145/53990.54010.

Andrew M. Pitts. 2023. "Locally nameless sets." *Proceedings of the ACM on Programming Languages*, 7, 488–514, POPL. doi:10.1145/3571210.

Andrew M. Pitts. 2001. "Nominal Logic: a first order theory of names and binding." In: *Theoretical Aspects of Computer Software*. Ed. by Naoki Kobayashi and Benjamin C. Pierce. Springer Berlin Heidelberg, Berlin, Heidelberg, 219–242. ISBN: 978-3-540-45500-4. doi:10.1007/3-540-45500-0_11.

Jeff Polakow. 2015. "Embedding a full linear lambda calculus in Haskell." In: *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*. Association for Computing Machinery, 177–188. doi:10.1145/2804302.2804309.

John C. Reynolds. 2003. "What do types mean? — From intrinsic to extrinsic semantics." In: *Programming Methodology* (Monographs in Computer Science). Ed. by Annabelle McIver and Carroll Morgan. Springer New York, 309–327. doi:10.1007/978-0-387-21798-7_15.

April Tune, Wendy Yang, and G. A. Kavvos. 2026. "Noninterference Through Bisimulation." In: *Trends in Functional Programming* (Lecture Notes in Computer Science). Ed. by Jeremy Gibbons. Vol. 15652. Springer Nature Switzerland, 259–280. doi:10.1007/978-3-031-99751-8_11.

Christian Urban and Christine Tasson. 2005. "Nominal techniques in Isabelle/HOL." In: *Proceedings of the 20th International Conference on Automated Deduction* (CADE' 20). Springer-Verlag, Tallinn, Estonia, 38–53. ISBN: 3540280057. doi:10.1007/11532231_4.

Geoffrey Washburn and Stephanie Weirich. 2008. "Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism." *Journal of Functional Programming*, 18, 1, 87–140. doi:10.1017/S0956796807006557.

Received 2025-11-13; accepted 2026-04-03