Embedding by Unembedding

KAZUTAKA MATSUDA, Tohoku University, Japan
SAMANTHA FROHLICH, University of Bristol, UK
MENG WANG, University of Bristol, UK
NICOLAS WU, Imperial College London, UK

Embedding is a language development technique that implements the object language as a library in a host language. There are many advantages of the approach, including being lightweight and the ability to inherit features of the host language. A notable example is the technique of Higher-Order Abstract Syntax (HOAS), which makes crucial use of higher-order functions to represent abstract syntax trees with binders. Despite its popularity, HOAS has its limitations. We observe that HOAS struggles with semantic domains that cannot be naturally expressed as functions, particularly when open expressions are involved. Prominent examples of this include incremental computation and reversible/bidirectional languages.

In this paper, we pin-point the challenge faced by HOAS as a mismatch between the semantic domain of host and object language functions, and propose a solution. The solution is based on the technique of *unembedding*, which converts from the finally-tagless representation to de Bruijn-indexed terms with strong correctness guarantees. We show that this approach is able to extend the applicability of HOAS while preserving its elegance. We provide a generic strategy for Embedding by Unembedding, and then demonstrate its effectiveness with two substantial case studies in the domains of incremental computation and bidirectional transformations. The resulting embedded implementations are comparable in features to the state-of-the-art language implementations in the respective areas.

CCS Concepts: • **Software and its engineering → Functional languages**; *Domain specific languages*; Syntax; Polymorphism.

Additional Key Words and Phrases: EDSL, functional programming, higher-order abstract syntax

ACM Reference Format:
Kazutaka Matsuda, Samantha Frohlich, Meng Wang, and Nicolas Wu. 2023. Embedding by Unembedding. *Proc. ACM Program. Lang.* 7, ICFP, Article 189 (August 2023), 47 pages. https://doi.org/10.1145/3607830

1 INTRODUCTION

Programming languages solve problems in an expressive and flexible way. When there is a particular problem to be solved, a *domain-specific language* (DSL) is desirable as it specializes to the problem domain in question. DSLs can be *embedded*, that is created, as a library within another *host* language, to save implementation effort and allow *guest* language users access to host language features. A host language feature that is desirable to reuse is its binding constructs.

Binders are important because they introduce variables, with common binders including **let** and λ-abstractions. A brute force implementation of binders is cumbersome and error prone. A popular alternative for embedding a language with binders is Higher-Order Abstract Syntax (HOAS) [Church

Authors' addresses: Kazutaka Matsuda, kztk@tohoku.ac.jp, Graduate School of Information Sciences, Tohoku University, Aramaki Aza-aoba 6-3-09, Aoba-ku, Sendai, Miyagi, Japan; Samantha Frohlich, samantha.frohlich@bristol.ac.uk, University of Bristol, BS8 1QU, Bristol, Avon, UK; Meng Wang, meng.wang@bristol.ac.uk, University of Bristol, BS8 1QU, Bristol, Avon, UK; Nicolas Wu, n.wu@imperial.ac.uk, Imperial College London, SW7 2AZ, London, UK.

2475-1421/2023/8-ART189
https://doi.org/10.1145/3607830

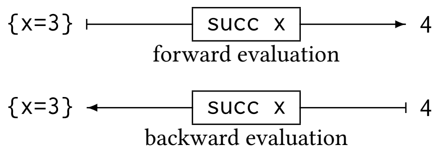


Fig. 1. Example of Advanced Semantics: Reversible Computation

1940; Huet and Lang 1978; Miller and Nadathur 1987; Pfenning and Elliott 1988]. As the name suggests, this technique uses higher-order functions to implement binders, essentially reusing the host's function binders as guest binders. There has been much development in this area, producing many variants of HOAS including the finally-tagless style [Carette et al. 2009], which supports extensibility in both syntax and semantics; or parametric HOAS (PHOAS) [Chlipala 2008], which adds support for non-compositional semantics.

However, HOAS and its variants encounter issues when trying to embed languages with advanced semantics that do not adhere to the traditional evaluation mode of taking open expressions and mapping free variable assignments to results. For example, reversible/bidirectional computation, exhibited in Fig. 1, suffers from this. Here, the open expression `succ x` can be evaluated forward (or "traditionally") by providing the values of the free variables (in this case the value of `x`) to get the result; or it can be evaluated backward by providing the result to yield the assignment of `x`. The problem lies in the attempt to embed guest functions with non-standard semantics as host functions with standard semantics.

So far, this problem has remained undocumented. But we suspect that it is the root cause of some of the struggles faced by language designers. Take the bidirectional programming literature as an example. Much work has gone into this area, with the popular approach of *lenses* and combinators [Foster et al. 2005, 2007] being awarded POPL's most influential paper in 2015, 10 years after its publication. But lenses (and all the follow on work) [Matsuda and Wang 2018a,d; Pacheco and Cunha 2010] struggled to include bindings in an embedded setting, resulting in significant difficulties in programming [Miltner et al. 2017]. There is a gap in the literature surrounding this problem, and it is not limited to bidirectional programming. We also investigate the problem in incremental computation, and speculate automatic differentiation, probabilistic programming, and quantum programming as potential targets for future exploration.

This paper pin-points and solves the issue of embedding binders and interpreting open expressions in domains where it was previously not possible. Our solution builds upon a technique known as *unembedding* [Atkey 2009; Atkey et al. 2009], which converts from the HOAS representation to de Bruijn-indexed terms. The crucial insight is that unembedding can be used to bridge the gap between the elegant syntax of HOAS and advanced domain semantics that were previously out of reach. The success of this solution is exhibited by our two substantial case studies, covering state-of-the-art languages in two of the domains afflicted with this problem: incremental computation and reversible computation. The incremental computation example successfully unembeds the cache-transfer-style (CTS) [Giarrusso et al. 2019] variant of the Incremental λ-Calculus (ILC) [Cai et al. 2014]. CTS builds upon the original language, which created incremental functional programs, to create even more incremental programs. In the domain of reversible programming, we create the first embedding of a bidirectional language with binders by embedding HOBiT [Matsuda and Wang 2018d], a higher-order lens language.

We name our new embedding framework *Embedding by Unembedding*. It acts as the glue between a familiar HOAS frontend and a system for managing environments, generalizing the original work that converts to de Bruijn-indexed terms, to allow for binders in more interesting semantic domains.

We design techniques to tame tedious management of environments and propose generic steps of embedding with the new framework. Specifically, we make the following contributions:[1]

- We identify a long-standing problem of being unable to use HOAS for complex application domains of languages with binders, pin-pointing the issue as a mismatch between host and guest language function semantics (§2).
- We propose unembedding (§3) as a solution to this problem.
- We build a new embedding framework based on unembedding and the finally-tagless style (§4).
- We conduct a case study by embedding Giarrusso et al.'s CTS (§5). The resulting language is easy to use, with implementation details properly hidden from programmers and scales well to higher-order constructs.
- We conduct another case study by embedding the bidirectional programming language HOBiT [Matsuda and Wang 2018d] (§6). The resulting language is the first successful embedding of a bidirectional language with binders.

We then discuss related work (§7) and future work (§8) before concluding (§9).

2 THE PROBLEM WITH HOAS

This section identifies and clarifies the problem with HOAS that our framework addresses. It begins by covering the necessary background to understand the problem (§2.1), then highlights the problem (§2.2).

2.1 Background: HOAS

The motivation for HOAS is that the manipulation of variables is cumbersome and sometimes hard to get right [Church 1940; Huet and Lang 1978; Miller and Nadathur 1987; Pfenning and Elliott 1988]. To do it properly, a number of issues need carefully considered: if there were two expressions bound to the same name, which expression is provided when that name is called (*name duplication*); whether this changes if the binders are nested (*nesting*); what happens when a name is called outside of its scope (*out of scope referencing*).

In an embedding setting, HOAS completely circumvents these problems by not implementing binders, but reusing the host language's binding infrastructure. It does this by encoding guest binders as higher-order functions in the host.

For example, HOAS would embed a let binder as a function with the following type, for some Exp (where Exp represents some arbitrary semantic domain in this shallow embedding example):

```
let' :: Exp a → (Exp a → Exp b) → Exp b
```

Here, the guest binder is created using a higher-order host function, where the first argument is the expression that gets bound, and the second is where the bound expression gets used. This allows for guest level terms to be written using host binders:

```
term y = let' y (\x → x)
```

This term expresses a let binder that does nothing, taking the variable y and doing nothing to it in the body. The point is to highlight how x is available to name the bound expression and use it in the body of the let, but no implementation effort was needed to control this naming, or its scoping, or the other binder issues. This is the ease and elegance of HOAS: an implementer gains the full expressivity of the host's binders, without any of the effort.

[1]A proof-of-concept prototype implementation of Embedding by Unembedding, which also contains the examples used in this paper, is available from https://doi.org/10.5281/zenodo.7990806.

Given its convenience, HOAS is the foundation of some prominent embedding techniques including *finally-tagless style* [Carette et al. 2009]. In Haskell, this consists of an embedding whose functions are given by a typeclass [Gibbons and Wu 2014] (or alternatively in ML families by a signature). Finally-tagless style is modular and extensible in both syntax and semantics: constructs can be grouped into different or dependent typeclasses; new constructs can be added as new typeclasses or methods; and new interpretations can be added as new typeclass instances. The let expression example can switch to this style, allowing it to generalize over the semantics Exp:

```
class Let exp where
  let' :: exp a → (exp a → exp b) → exp b
```

While we have focussed on the finally-tagless [Carette et al. 2009] variant of HOAS, this problem also affects parametric HOAS (PHOAS) [Chlipala 2008]. It may add support for non-compositional semantics, however, it is still not easy to implement advanced semantics in PHOAS for the same reason that it is hard for HOAS.

2.2 The Problem

With HOAS (and its variants), the semantics of the term under a binder in the guest language is implemented as a host language function. This representation permits only one operation: the application of the function. This is sufficient in many cases, but becomes a problem with complex semantic domains, where additional manipulation of variable environment is required outside of the standard instantiation of variables with function arguments. This is not just because these domains often feature non-standard binding constructs, but more crucially, the way the *open expressions* (found inside the binders) are treated differs from standard function bodies.

Take invertible computation as an example. An expressive language in this domain features special backward case constructs and pattern matching (details in §6). Moreover, unlike the standard computation model where an expression evaluates to a value under a value environment, the backward semantics is commonly modelled as an interpretation of an open expression (with the binder removed) and a given value to produce a value environment containing the bound variables (see, e.g., [Yokoyama et al. 2011]). For example, the open expression `succ x` from Fig 1, when executed backwards with the given value 4 produces a binding environment `{x = 3}`. It is not obvious how this behaviour can be encoded by a function as HOAS requires.

A similar problem arises in the exploration of incremental computation, where recomputation is avoided to achieve a performance gain. Cai et al. [2014]'s incremental λ-calculus achieves the goal by interpreting a function as a mapping from input changes to output changes. Open expressions are instrumental here too, as they have clear notions of input and output as opposed to closed ones. The situation is even more complicated with the state-of-the-art cache-transfer-style (CTS) [Giarrusso et al. 2019] variant, which introduces a notion of "cache" to support non-*self-maintained* computations [Cai et al. 2014]. We defer the details to the case study (§5). The key to know here is that this "cache" is a state carried through and updated by the computation, in addition to the already complex semantics of the open expressions. Again, HOAS struggles to encode this behaviour.

These domains share a common characteristic that the complex semantics requires some kind of manipulation of the environment. This is an interesting dilemma. On the one hand, HOAS gives us a simple programming interface as it avoids manipulation of variables. On the other hand, some semantics require access to environments, i.e., assignments to variables, which are hidden by HOAS (for a good reason).

In this paper, we advocate Embedding by Unembedding as a solution that works for both HOAS and PHOAS, and allows the embedding of a guest language whose semantics differ from its host, without sacrificing the desirable HOAS (or PHOAS) interface.

3 BACKGROUND: THE TECHNIQUE OF UNEMBEDDING

Embedding by Unembedding builds upon *Unembedding* [Atkey 2009; Atkey et al. 2009]. This section elaborates on this technique, covering what it is, its original motivation, and why it is suitable for our purposes (§3.1); how it leverages de Bruijn-indexed terms as a more amenable representation for their purposes and how this generalizes to solve our embedding problem (§3.2); and the details of how the technique works (§3.4).

3.1 What and Why

Unembedding is a technique that gives a round-tripping transformation from a finally-tagless [Carette et al. 2009] language representation to de Bruijn-indexed terms. As the name suggests, the technique is in a sense the opposite of HOAS embedding, converting an embedded implementation to an explicit Abstract Syntax Tree (AST) representation. The original motivation for this work is supporting certain optimizations that require global analysis of the AST [Atkey et al. 2009], to which de Bruijn-indexed terms are more amenable than the finally-tagless representation.

We observe that this technique has much greater applicability. Our iteration on their work distils out the desirable qualities of the representation into a generalization that we will introduce later.

3.2 The de Bruijn Representation

We first introduce a representation of environments, as the explicit access to environments is the advantage of de Bruijn representations over HOAS, and this explicit access is also crucial in our generalization presented in this paper. An environment is a collection of values of different types.

The simplest way to achieve this in Haskell is as a tuple, but as this gets messy, a heterogeneous list [Kiselyov et al. 2004] is more appropriate:

```
data VEnv (as :: [k]) where
  VENil  :: VEnv '[]
  VECons :: a → VEnv as → VEnv (a ': as)

exEnv = VECons 1 (VECons 'a' VENil) :: VEnv '[Int,Char]
```

(Here, "`':`" and "`'[]`" are the type-level versions of the cons operator "`:`" and the nil "`[]`", respectively.) `VEnv` works the same way as a normal list, but with the addition of a type level list (`as`), allowing it to hold values of different types as desired to implement an environment. For example, `exEnv` stores both an `Int` and a `Char`. This environment of values is a good start, but there are many other types of environment that may be required. For example, the guest types might be distinguished from Haskell types using a wrapper type; or one might wish to store guest expressions, as opposed to values, to represent open terms; or perhaps a value-level representation of a type environment is desired. To accommodate these, a common approach[2] is to generalize `VEnv` to wrap a container around the stored values:

```
data Env (f :: k → Type) (as :: [k]) where
  ENil  :: Env f '[]
  ECons :: f a → Env f as → Env f (a ': as)
```

Different instantiations of this container `f` create different environments:

[2]See, e.g., http://agda.github.io/agda-stdlib/Data.Star.Environment.html.

```
type VEnv     = Env Identity -- Value environment.
type EEnv exp = Env exp      -- Expression environment.
type TEnv     = Env Proxy    -- Type environment.
```

The `Identity` container[3] creates a value environment the same as `VEnv`; `exp` allows an environment to store guest language expressions; `Proxy`[4] creates a type environment.

This representation allows us to provide a unified interface for manipulation of environments. For example, the environment infrastructure includes a way to index into environments:

```
lookEnv :: Env f as → Ix as a → f a
lookEnv (ECons v _)   IxZ     = v
lookEnv (ECons _ env) (IxS n) = lookEnv env n

data Ix (as :: [k]) (a :: k) where
  IxZ :: Ix (a ': as) a                -- At element in question.
  IxS :: Ix as a → Ix (b ': as) a      -- Element lies further into the env.
```

Here, `lookEnv` takes an environment and a witness of type `Ix`, which ensures that a type `a` occurs in a type level list `as`. In the context of `lookEnv`, this corresponds to ensuring that the environment contains the corresponding `f a` value. To extract the value, it walks down the environment to the index at which the value lies using the witness type.

A de Bruijn representation of the Simply-typed λ-Calculus (STLC) demonstrates this infrastructure:

```
data DSTLC (env :: [Type]) a where
  DVar :: Ix env a → DSTLC env a
  DLam :: DSTLC (a ': env) b       → DSTLC env (a → b)
  DApp :: DSTLC       env (a → b) → DSTLC env a → DSTLC env b
```

Intuitively, `DSTLC env a` represents STLC terms that have type `a` and free variables in `env`. Hence, `DLam` says that we can construct a function of type `a → b` from an expression of type `b` with an extra free variable of type `a`. A variable is represented by its index (`Ix env a`) in the environment. For example, `x` is represented by `DVar IxZ` assuming `x` appears first in the environment, and hence `(\x → x)` is represented by `DLam (DVar IxZ)`. Using indices, this representation gives a canonical representation to α-equivalent terms. However, indices change when the environment grows e.g., `(\x → (\y → x) x)` is represented as this DSTLC term: `DLam (DApp (DLam (DVar (IxS IxZ) )) (DVar IxZ))`.

3.3 The HOAS Representation

In contrast, since in HOAS the variables are represented by using the host's bindings, there is no corresponding method to `DVar`, as presented in Fig. 2 with representation of the STLC in the finally-tagless style [Carette et al. 2009]. Here, the guest-level bindings are represented by the host-level bindings; e.g., `(\x → (\y → x) x)` is represented as `lam (\x → app (lam (\y → x)) x)`. A caveat is that we can pass arbitrary functions to `lam`, which, without restrictions, includes those that may not be STLC terms, and perhaps perform case analysis on their inputs. It is crucial to

[3] `Identity` is a container that does nothing, representing the identity functor/monad in Haskell.
It is defined: **newtype** `Identity a = Identity a`

[4] `Proxy` is a special Haskell type, that attaches a type to a placeholder constructor, holding no data, meaning that no values end up in the environment, but the type level list is preserved.
It is defined: **data** `Proxy a = Proxy`

```
class STLChoas exp where
  lam :: (exp a → exp b) → exp (a → b)
  app :: exp (a → b) → exp a → exp b
```

Fig. 2. STLC HOAS Representation

exclude such exotic terms [Chlipala 2008], e.g., by using the parametricity of `exp`, which also plays a key role in the unembedding [Atkey 2009].

HOAS is useful for both language EDSL users and implementers, as it uses host-language binders to represent guest-language binders. From the user's perspective, compared with the naïve string representation of variables, it allows us to use the host language's variables for a guest language variables, providing access to the host's name checking and IDE's support concerning variables (e.g., displaying their types and automatic completion). From the language implementer's perspective, it frees us from manipulation of names, including name checking, renaming, and substitution. Some advantages, such as removing the need for name-checking and renaming, also apply when comparing HOAS to de Bruijn terms. HOAS also has further advantages over de Bruijn terms, especially from the user's perspective, as variables are more intuitive than indices to program with and enjoy IDE's support. It is the goal of this paper is to extend the applicability to HOAS to more semantic domains, while preserving these advantages as much as possible, and this unembedding technique is the key to doing that.

The finally-tagless style of HOAS is preferable over plain HOAS because it addresses known issues with plain HOAS: exotic terms and defining semantics.[5] It is one of two state-of-the-art variants that address the issues, with the other being PHOAS [Chlipala 2008]. The use of typeclasses also has the additional advantages over PHOAS of extensibility (a typeclass can have subclasses) and efficiency (no tags are required to distinguish AST nodes—this is why the representation is called "tagless" [Carette et al. 2009]).

3.4 Converting between HOAS and de Bruijn

Unembedding achieves the best of both worlds: a language with the beloved user-friendly HOAS interface that is also amenable to analysis, by converting between HOAS and de Bruijn representations of a language. We expand upon this, using a similar conversion to attach a HOAS interface to a first-order representation that supports the semantics of interest.

The conversion from de Bruijn to an open HOAS term (`OTHoas`) falls out quite naturally. In the λ-calculus example, `DLam` and `DApp` map directly to `lam` and `app`, and `DVar` looks up what to inline in a value environment:

```
type OTHoas env a = forall exp. STLChoas exp ⇒ Env exp env → exp a
toHOAS :: DSTLC env a → OTHoas env a
toHOAS (DVar n)   = \g → lookEnv g n
toHOAS (DLam t)   = \g → lam (\x → toHOAS t (ECons x g))
toHOAS (DApp f p) = \g → app (toHOAS f g) (toHOAS p g)
```

Each line of the `toHOAS` function constructs a result of type `OTHoas`, in other words a function from an environment to a guest term. The first line finds its guest term in the provided environment of

[5]Defining transformations from a plain HOAS tree to its semantics is problematic. This is because the negative occurrence of the AST type in constructs such as λ requires a transformation in opposite direction. Even implementing the standard semantics is not straightforward (see, e.g., [Fegaras and Sheard 1996]). Using a typeclass solves this issue because *instantiating* a type parameter of the typeclass with its instance does not share the same issues as *transformation*.

guest expressions (g) at the location specified by `DVar n`. The second is translating the λ construct, so creates a `lam` term. The `lam` term requires a guest expression of the result type with an environment extended by the variable it introduces. This is created by recursively converting the body of the λ, `t`, with the required extended environment. Application is converted by providing the `app` construct with function and argument converted recursively.

Notice that, e.g., `OTHoas [a,b] c` is isomorphic to `forall exp. STLChoas exp ⇒ exp a → exp b → exp c`. Here, the universal quantification in `OTHoas` ensures that `toHOAS` cannot produce exotic terms.

The other direction executes the unpacking of the HOAS representation into an explicit syntax tree:

```
fromHOAS :: (forall exp. STLChoas exp ⇒ exp a) → DSTLC '[] a
fromHOAS h = runUSTLC h ENil
```

This is more involved because typing environment information needs to be recovered, necessitating an appropriate interpretation of `exp`: `USTLC` (the application of `runUSTLC` to `h` triggers `h` to assume this type). Since `exp` does not contain any information about the typing environment, the basic idea of unembedding is to make this type a function from this information, in the form of a typing environment, to the explicit `DSTLC` syntax tree:

```
newtype USTLC a = USTLC {runUSTLC :: forall env. TEnv env → DSTLC env a}
```

To complete `USTLC` as an interpretation of `exp`, `lam` and `app` must be defined over this type. The interpretation of `app` is:

```
app :: USTLC (a → b) → USTLC a → USTLC b
app (USTLC u1) (USTLC u2) = USTLC (\g → DApp (u1 g) (u2 g))
```

The environment `g` is fed to each subexpression `u1`/`u2` to allow the construction of a `DApp` term. The interpretation of `lam` is trickier, which has the following form (where the type annotations are just for readability):

```
lam :: (USTLC a → USTLC b) → USTLC (a → b)
lam k = USTLC $ \(g :: TEnv env) →
  let ga = ECons Proxy g :: TEnv (a ': env) -- extend env
      x  = USTLC (\(g' :: TEnv env') →      -- prepare arg for k
             weakenManyDSTLC ga g' (DVar IxZ :: DSTLC (a:env) a)) :: USTLC a
  in DLam (runUSTLC (k x) ga)
             -- apply k to x with extended arg to get desired arg for DLam
```

This instantiates `lam` concretely into the `USTLC` type that behaves as a λ: introducing a new variable that can be referred to in its body by extending the environment. It extends `env` to `a:env`; accordingly, it also prepares its value level representation `ga :: TEnv (a ': env)`. Recall that `TEnv` = `Env Proxy`. Thanks to the extension, we now can have a variable `DVar IxZ :: DSTLC (a :env) a`, which needs to be converted to `DSTLC env' a` to prepare `x :: USTLC a` to be fed to the `lam`'s argument `k`. This conversion is done by weakening,[6] which performs appropriate shifting of variables to adjust the environment from `ga` to the actual incoming one `g'`.

```
weakenManyDSTLC g1 g2 = weakenBy (compare g1 g2)
```

Here, `compare` and `weakenBy` have the following signatures.

[6]Note that at this assumes that the environment follows a last in, first out (LIFO) ordering.

```
compare  :: TEnv env1 → TEnv env2 → env1 ≤ env2
weakenBy :: env1 ≤ env2 → DSTLC env1 a → DSTLC env2 a
```

The function `compare` compares two type environments to have a witness that `env2` is an extension of `env1` (more precisely, `env2` = `env ++ env1` for some `env`). Then, `weakenBy` performs weakening by using the witness.

Of course, `compare` is a partial function, since `env1 ≤ env2` may not hold in general (e.g., `env1 = [a,b]` and `env2 = [b,a]`). Here, the exclusion of exotic terms matters; the conversion from the HOAS representation is meaningful only from polymorphic term. The universally quantified `exp` in `fromHOAS` ensures that the HOAS representation `h` consists only of `lam` and `app`, and hence only the operation on environments are extensions (using `lam`). As a result, since `x` appears in a deeper position in `k`, it will receive an extended environment (`g'`) than the outer one (`ga`) passed to `k x`. This is why `compare` always succeeds in `fromHOAS`. Atkey [2009] proves this fact formally by using the (Kripke) parametricity of `h`, together with the fact that `toHOAS (fromHOAS h) ENil` is equivalent to `h`.[7] That is, the two representations are isomorphic.

Thus, unembedding has unpacked the HOAS to create an explicit syntax tree. The original work used this to support optimizations and code generation, especially those that require non-local analysis of ASTs [Atkey et al. 2009; Matsuda and Wang 2018b; McDonell et al. 2013; Stewart 2010; Werk et al. 2012]).[8] Our key observation is that the unembedding technique has so much more potential. The current work is purely syntactic, but by considering semantics, unembedding can used to support the embedding of languages with advanced computation models.

4 EMBEDDING BY UNEMBEDDING

This section explores using unembedding for embedding languages with "complex semantic domains" and introduces our Embedding by Unembedding framework. The exploration is conducted with the STLC. With its standard semantics, it does not require our technique, but our primary focus here is the explanation of our technique and preparation for later sections, where we embed languages with more complex semantics that require Embedding by Unembedding.

The section begins by clarifying what languages we have observed Embedding by Unembedding is applicable to (§4.1), before beginning a standard embedding of the STLC (§4.2), and investigating where unembedding comes in useful (§4.3), and finishing in a summary of our technique (§4.4).

4.1 Applicability

We have observed that Embedding by Unembedding is applicable to languages that: are simply typed; have constructs that are either "first" or "second" order;[9] have compositional semantics; and support a rule for weakening environments. These qualities constitute sufficient criteria. By "first" or "second" order, we refer to two categories that we split language constructs into: those that introduce binders (which we refer to as *second-order*), and those that do not (*first-order*). The typing

[7]The opposite direction `fromHOAS (toHOAS t ENil) = t` can be proved by the induction on `t` without using parametricity.
[8]To be precise, Werk et al. [2012] and Stewart [2010] use the core conversion technique from McDonell et al. [2013], described in https://github.com/mchakravarty/hoas-conv, which is operationally similar to the unembedding but independently developed.
[9]We use the term "second-order" specifically in the sense of "second-order abstract syntax", as used in the line of work by Fiore et al. [1999], not in the sense of "second-order λ calculus" (System F).

rules for each category of construct will adhere to the following general forms and compositional semantics for a certain semantic domain $\mathcal{S}$:[10]

$$\frac{\{\Gamma \vdash e_i : A_i\}_{i=1,\dots,n}}{\Gamma \vdash \textbf{con}\ e_1\ \dots\ e_n : B} \qquad \frac{\{\Gamma, x_{i1} : A_{i1}, \dots, x_{im_i} : A_{im_i} \vdash e_i : B_i\}_{i=1,\dots,n}}{\Gamma \vdash \textbf{con}'\ (x_{11} \dots x_{1m_1}. e_1)\ \dots\ (x_{n1} \dots x_{nm_n}. e_n) : C}$$

Here, $x_{11}, \dots, x_{1m_1}, \dots, x_{n1}, \dots, x_{nm_n}$ are the variables bound by the construct $\textbf{con}'$. For example in the STLC, the function application and the λ abstraction belong to the former and latter sorts of constructs, respectively.

4.2 Embedding

Before utilizing unembedding, our embedding of the STLC must begin as any other: its syntax and semantics need to be considered and pinned down.

Semantic Domain. Firstly, the semantic domain must be identified. This involves considering what the meaning of the embedded language should be. Is it a reversible language where the semantics should run “backwards” by mapping results to free variable assignments? Does it need to support the translation of changes in the inputs to changes in the outputs to facilitate incremental computation? A concrete way of doing this is by reifying it as a datatype in the host language.

As discussed, semantics where unembedding will come in useful are those where open expressions matter. That is, those that must be ready for interpreting the standard typing rule for variables: $\Gamma, x : A, \Gamma' \vdash x : A$. This idea is encapsulated into the `Variables` typeclass below.

```
class Variables (sem :: [k] → k → Type) where
  var    :: sem (a ': as) a
  weaken :: sem as a → sem (b ': as) a
```

If the reified semantic type is an instance of this class, then it is amenable to unembedding. `Variables` represents the class of semantics (`sem`) that depend on some context (`[k]`, a list of types of kind `k`), have one particular variable in focus (`k`), and are able to be unembedded. The `var` method ensures that we can interpret $\Gamma, x : A \vdash x : A$ (where Γ' is empty in the above), and `weaken` ensures that more variables can be added to a context for interpretations. By mixing the two, we can interpret variables that appear in arbitrary positions in a context; e.g., `weaken (weaken var) :: Variables sem ⇒ sem (b ': c ': a ': env) a` corresponds to $\Gamma, x : A, y : B, z : C \vdash x : A$.

An astute reader may wonder why we only focus on weakening (`weaken`), when we could have other manipulations (namely, exchange and contraction) that ensure that Γ is essentially a set instead of a sequence. This is because weakening is the only manipulation of environments used in Embedding by Unembedding. That is, `Variables` encapsulates only the minimal set of operations needed, though we expect that the semantic domain must support general extensions and rearrangements on contexts, their actions on the domain, and that semantic functions respect such context manipulations.[11]

Example 4.1. The (standard) semantics of the STLC takes an environment of free variables to a result. This is realized as a concrete Haskell datatype by wrapping up a function from a variable environment to a result as a **`newtype`**.

```
newtype STLC env a = Sim { runSim :: VEnv env → a }
type VEnv = Env Identity
```

[10]The reader may notice that **con** is a special case of **con**′. We distinguish them for readability (**con** can be compared to *app*; **con**′ can be compared to *lam*), and to highlight that unembedding enables the the ability to treat **con**′.
[11]We conjecture that this treatment essentially corresponds to the functor category $[\mathbb{F}, \mathbf{Set}]$ in Fiore et al. [1999].

Here, runSim is exactly the semantics of the STLC: a function from a value environment to a result. The value environment is the Env type, where the type container is just Identity. The use of the Env type provided by our framework helps make this semantic type an instance of the Variables typeclass. For var, runSim can extract the first value from the VEnv. For weaken, a new Sim can be constructed so that it just ignores the first element of the environment:

```
instance Variables STLC where
  var :: STLC (a ': as) a
  var = Sim (\(ECons (Identity x) _) → x)
  weaken :: STLC as a → STLC (b ': as) a
  weaken (Sim f) = Sim (\(ECons _ env) → f env)
```

Language Constructs. Semantics in hand, the next step considers how each of the language constructs should behave in that domain. Each construct is represented in the semantic domain as a function over the semantics of terms-in-context.

Example 4.2. The STLC only has two constructs: one first-order (app), and one second-order (lam). The semantic function for a first-order construct such as app takes the form of a function over the STLC type. Thus the semantic function for app should be a function with two arguments: a function (f) and its argument (x), where its definition applies f to x. What differentiates this semantic function from a normal Haskell function representing app is that f and x are wrapped up in the STLC type:

```
appSem :: STLC env (a → b) → STLC env a → STLC env b
appSem fTerm aTerm = Sim (\g →
  let f = runSim fTerm g -- Unpack function using g.
      x = runSim aTerm g -- Unpack arg using g.
  in f x) -- Apply.
```

Second-order constructs are slightly more involved, due to the semantics needing to operate on the environment. In the example, lam takes a term of the STLC, represented by the STLC type, that knows how to make the result should it have the argument in its environment, and produces a function term:

```
lamSem :: STLC (a ': env) b → STLC env (a → b)
lamSem bTerm = Sim (\g x →
  let g' = ECons (Identity x) g -- Add x :: a to env.
  in runSim bTerm g')           -- Use to make b.
```

The function term works by storing its argument into the environment and using the original body to produce the result.

Syntax. While the semantic functions are an accurate representation of the semantics of the language, they are not very usable. Users want an interface with binders and variable names, not environments and indices, and language designers want to give them this with as little effort as possible. This is why HOAS is typically used to represent the syntax of an embedded language.

This step is the same as any other finally-tagless embedding: creating a typeclass to represent the language, where each construct is a method, and binders are achieved as host level functions.

Example 4.3. The STLC example can use STLChoas from Fig. 2.

Gap. Now the embedded language has both syntax and semantics. All that remains is to connect the two. Normally, this step is as simple as making the semantic type an instance of the finally-tagless syntax. However, this is not straightforward when the semantic type explicitly uses environments.

The STLC example has been contrived such that its standard semantics do this needlessly,[12] but domains such as those of our case studies will require it. Thus something is needed to act as a conduit between the HOAS syntax and interesting semantics. This something is unembedding.

4.3 Unembedding

Unembedding marries syntax and semantics, achieving a complex semantic domain without sacrificing the user-friendly interface. It facilitates making the semantic type an instance of the HOAS typeclass.

We create a framework around unembedding so that, on the surface, this connection can be done in two steps: wrap the semantic type in provided wrapper type `EnvI` (for "environment indexed"); apply provided lifting functions (`liftFOn` and `liftSO`) to the semantic functions to instantiate the typeclass methods. Here, the only choice that needs to be made is what lifting function to use. The framework provides a family of lifting functions: a variety for first-order constructs that cater to different numbers of arguments, and one for second-order constructs.

Example 4.4. As an example of using lifting functions, consider `STLC`. Here this means that `STLC` is wrapped in `EnvI` so that it can be an instance of `STLChoas`:

```
instance STLChoas (EnvI STLC) where
  app = UE.liftFO2 appSem -- not a binder ⇒ FO, 2 args ⇒ FO2
  lam = UE.liftSOn (ol1 :. End) lamSem -- binder ⇒ SO
```

Here, `app` is defined as the lifting of `appSem`. Because `app` is first-order, one of the `liftFO` functions is used, specifically `liftFO2` because `app` has two arguments. As a second-order construct, `lam` is different. Now its arguments can have arguments. To allow for this, `liftSOn` takes a description of what arguments are expected: how many there are and how many arguments (bindings) they themselves have. This description is a list, with each element representing an argument and how many arguments it itself takes. In the case of `lam`, there is one argument that itself has one argument, thus the singleton list of `ol1` is needed.

Under the hood, our framework is managing a de Bruijn-index-like first-order representation so that the involvement of the environment is not user-facing. This is a more general application of the unembedding technique, where semantic functions replace de Bruijn terms. Recall that the main challenge that Embedding by Unembedding tackles is the conflict of requirements between the advanced and non-standard guest function semantics and the desire to use HOAS (and thus standard host function semantics) to achieve this. The advanced semantics require an environment; whereas HOAS expects none because the whole point of it is to defer the handling of binders to the host language:

```
semAdv  :: [k] → k → Type
semHOAS ::       k → Type
```

This mismatch is a generalization of the one between de Bruijn indexed terms and HOAS expressions. Thus, a wrapper type can be used to connect the semantic functions to their HOAS representation. This wrapper type, is `USTLC` from §3, but with the first-order semantics (e.g. `DSTLC`) generalized out as a parameter (*sem*):

[12] However, the technique would be useful if one interprets STLC in an arbitrary cartesian closed category [Elliott 2017].

```
newtype EnvI sem a = EnvI { runEnvI :: forall env. TEnv env → sem env a }
```

Regarding implementation of first-order, non-binding constructs, recall that we pass around a value-level representation of the typing environment, env, in the implementation of app in §3. This actually serves as a common pattern. Hence, even without `liftFO` families, we can implement app of the STLC similarly with appSem as below.

```
app (EnvI e1) (EnvI e2) = EnvI (\g → appSem (e1 g) (e2 g))
```

The implementation depends only on the number `n` of arguments of a construct. We provide `liftFOn` to avoid this routine.

Implementation of second-order, binding constructs is analogous to the `lam` case in §3, with the implementation of `lam` without `liftSO` following the same story, the only differences being the use of `var` instead of `DVar IxZ`, and the use of `lamSem` instead of a constructor of de Bruijn terms, and a more general implementation of weakenMany that uses weaken from the `Variables` typeclass.

```
lam k = EnvI $ \(g :: TEnv env) →
  let ga = ECons Proxy g :: TEnv (a ': env)
      x  = EnvI (\g' → weakenMany ga g' var)
  in lamSem (runEnvI (k x) ga)
```

In general, the lifting of a binding construct is done by: for each argument k of type b that binds variables of types a1, . . ., an:

(1) Extend g to obtain `ga :: TEnv (a1 ': ... ': an ': env)`.
(2) Prepare variables `vi :: sem (a1 ': ... ': an ': env) ai` in the semantic domain `sem` by using `var` and weaken, and then convert them to `xi :: EnvI sem ai` by weakening:

    ```
    weakenMany :: Variables sem ⇒
                  TEnv env → TEnv env' → sem env a → sem env' a
    ```

    The function weakenMany implements our iteration on the original unembedding's `compare` and weaken, which repeatedly applies weaken from the `Variables` typeclass.

If we ignore manipulation of types, this pattern depends only on the number of bindings for each argument of a construct. So, we provide `liftSO` to perform the routine lifting by knowing the numbers of bindings, as `lam = liftSOn (ol1 :. End) lamSem`. Though useful, the implementation of `liftSO` is not in the scope of this paper, and thus we advise curious readers to see Appendix A.

While the implementation of the lifting functions is quite involved, in reality, this is just the tedious management of environments that is tamed by the Embedding by Unembedding framework. All the language designer actually has to do to connect their interesting semantics to a HOAS frontend is use them.

4.4 Embedding by Unembedding

Embedding by Unembedding successfully embeds a language with binders and interesting semantics, allowing for the interpretation of open expressions.

Open Expressions. The ability to interpret open expressions is a key feature provided through Embedding by Unembedding that existing embedding techniques have not yet tackled.

The interpretation of open expression is provided generically for any semantics that are members of the `Variables` typeclass:

```
runOpen :: Variables sem ⇒ (EnvI sem a → EnvI sem b) → sem '[a] b
runOpen f = let gA = ECons Proxy ENil
```

```
                    x  = EnvI $ \g' ⟶ weakenMany gA g' var
                in runEnvI (f x) gA
```

In the specific case where there is only one free variable, `runOpen` takes an open term and transforms it into the semantic type. Since the open term is expressed as a host level function, `runOpen` creates the `EnvI sem a` argument to run it on. It does this by creating a type environment with just the single free variable in it, and then creating a `var` expression where that environment is unified with the incoming one. To ensure that there is indeed only one free variable `runOpen` must be wrapped in `runOpenSTLC` similarly to the original unembedding.

```
runOpenSTLC :: (forall exp . STLChoas exp ⇒ exp a ⟶ exp b) ⟶ STLC '[a] b
runOpenSTLC f = runOpen f
```

The framework also provides a more general `runOpenN` for `n` free variables.

Example 4.5. Putting the embedded `STLC` to work builds on top of `runOpenSTLC`:

```
runSTLC :: STLC '[a] b ⟶ a ⟶ b
runSTLC t x = let g = ECons (Identity x) ENil in runSim t g
```

This function operates on the output of `runOpen`, interpreting the `STLC` term with one free variable as a function from `a` to `b`: the standard semantics of the λ-calculus. Here it is in action, evaluating the application of the identity function to a free variable that has a value of three:

```
term :: STLChoas exp ⇒ exp b ⟶ exp b
term x = app (lam (\y ⟶ y)) x

> runSTLC (runOpenSTLC term) 3
3
```

The STLC expression is written as `term`, whose type is what `runOpenSTLC` expects. All that remains is to use `runSTLC` to provide the value of the free variable.

Correctness. The correctness of Embedding by Unembedding—that `runOpenSTLC` implements the standard semantics for this particular case—is understood as a generalization of Atkey [2009]'s original unembedding followed by fusion [Gill et al. 1993]. Continuing to consider our STLC example, unembedding maps the `STLChoas` representations to the `DSTLC` in the one-to-one manner, and the semantics of `DSTLC` is given by replacing constructors with their corresponding semantic functions. Thus, fusing the two process together by replacing `DSTLC` constructs with their corresponding semantic functions in the methods of `STLChaos` for `USTLC`, we get an instance of `STLChoas` that coincides with the semantics given for de Bruijn terms. This is what Embedding by Unembedding is doing. In this process, we slightly extend the original discussion to show the correspondence in *open terms*, namely HOAS representations of type `forall exp. STLChoas exp ⇒ exp a ⟶ exp b` and de Bruijn terms of type `DSTLC '[a] b`, assuming the Kripke parametricity of the HOAS representations (observe `forall` in the type). The correctness of the fusion can also be proved by using the parametricity [Gill et al. 1993]. We can repeat the discussion on a case-by-case basis for languages satisfying the criteria in §4.1.

Summary. The process of Embedding by Unembedding can be summarized into a recipe, where each different part of the example constitutes a step:

Step 1. Identify and reify the semantic domain. (Shown for the STLC in Example 4.1)
Step 2. Prepare semantic functions for each construct. (Example 4.2)
Step 3. Provide a HOAS representation of the syntax. (Example 4.3)

Step 4. Implement semantics though appropriate liftings. (Example 4.4, recall that all the language designer had to do here was use the provided lifting functions and wrapper type.)

Step 5. Put the language to work! (Example 4.5)

Appendix B runs through the application of this recipe to another example: the ILC, the predecessor of the incremental language that is the subject of our incremental computation case study.

An Extensible and Modular Approach. A major benefit of embedding is the ease of extending the language, which is indeed the case in Embedding by Unembedding. In fact, our approach features the state-of-the-art modularity and extensibility of the finally-tagless style. To add new constructs, the recipe can just be repeated for the new constructs only, and the language can be created in a modular manner, with different constructs grouped into different and dependant typeclasses. The embedded language presented in §5 makes use of this advantage to design the core system simple as possible and enables us to extend the core to incremental primitives for user-defined datatypes. Also, we detail the adding of sequences[13] to the ILC, which can be found in Appendix B.

Case Studies. We will now demonstrate Embedding by Unembedding on two domains that require the technique to be embedded: incremental and reversible computation. These two domains could not previously be embedded due to the limitations of HOAS. Limitations that are now solved by Embedding by Unembedding.

Note that neither of our case studies feature λ binders. This is a consequence of the semantic domains of the languages in question, not the unembedding technique. In the case of the CTS variant of the ILC, the original work [Giarrusso et al. 2019] does not support the *typed* λ abstractions and applications as seen in STLC, as their semantics (which was originally untyped) does not respect types for them. We follow suit on choice of binder, avoiding λ, as this is a domain specific issue. In the case of our reversible case study, it is simply difficult to find a semantic domain that appropriately encapsulates the semantics and supports abstractions (see, e.g., [Abramsky 2005; Chen and Sabry 2021; Matsuda and Wang 2018d, 2020]). As such, we apply the same trick of switching to a let binder, this time deviating slightly from the original language.

5 CASE STUDY: INCREMENTAL λ-CALCULUS

The first non-trivial computation model we consider in this paper is incremental computation. This domain is focussed on speeding up recomputation, something that is invaluable in our world of big data [Acar and Chen 2013], and it is a technique that transcends language paradigms, with research into the incrementalization of functional [Chen et al. 2014a], imperative [Hammer et al. 2009], and object orientated languages [Shankar and Bodík 2007].

Unlike normal computations, incremental computation aims to only recompute those outputs that depend on the changes of the input in order to achieve efficiency. Generally, this involves mapping input changes to output changes instead of recomputation. For example, when summing a list, if just one element was incremented, it makes no sense to recompute the sum. It would be far more efficient to just increment the initial sum of the list. Many models of incremental computation exist. Our case study focusses on Cai et al. [2014]'s incremental λ-calculus, specifically an improved variant of it known as the cache-transfer-style (CTS) [Giarrusso et al. 2019], which propagates changes to the input into changes of the output without using pointers or destructive operations.

This section will cover the unembedding of the CTS [Giarrusso et al. 2019]. (Appendix B also walks through the unembedding of the original calculus.) It begins by introducing the CTS variant (§5.1) and why it is of interest (§5.2), before exploring the interesting parts of the unembedding (§5.3), including an evaluation of the performance of the embedded language.

[13] Sequences are a speedier version of lists in Haskell, and as ILC is about efficiency they are chosen.

5.1 Cache-Transfer-Style Incremental Functions

An incremental function in CTS can be characterized by the following type:

$$\exists C.(A \to (B, C), (\Delta A, C) \to (\Delta B, C))$$

The first component (which we call an *initializer*) is a transformation from A to B that also computes a "cache" C. The existential quantification tells us that the cache may differ between computations. The cache is used and updated for the second component (which we call a Δ-*translator*) to translate input changes ΔA into output changes ΔB. Take the list reversal *reverse* as an example. When an insertion to the ith position happens to the original input list xs, an insertion of the same element happens to the $(\mathit{length}\ xs - i)$th position of the corresponding output. So the cache can be the length of the original input. The cache may also be changed by the input change, which is the reason for the C in the output of the Δ-translator. Here, an insertion also changes the length of the input.

5.2 Why Unembedding is Desired and Applicable

As mentioned in §2.2, the ability to handle open expressions is crucial to the change-propagation-based incremental computation in general, not limited to Cai et al. [2014]'s original work and the CTS version [Giarrusso et al. 2019]. Unlike closed expressions, open expressions have clear inputs (value environments) and outputs (evaluation results), which makes the notions of input changes and output changes clear. That is, we can view an open expression as a transformation from value environments to resulting values and then consider the change propagation regarding the transformation.

It is true that we can avoid the direct manipulation of open expressions for particular computation models. For example, for the original ILC, where an incremental function has type $(A \to B, A \to \Delta A \to \Delta B)$, we can apply tupling [Chin 1993] to inject the incremental function in $(A, \Delta A) \to (B, \Delta B)$. Thus, we have a simple HOAS implementation that interprets `exp a = (a, Delta a)` using a type family `Delta` to represent changes on `a`. However, such a simplification is limited to particular models; actually, the tupling approach does not scale to the CTS variant [Giarrusso et al. 2019] due to existentially-quantified caches.

The CTS example is of interest because it expands the applicability of the original ILC to efficiently incrementalize more computations, and the addition of the cache that allows this precludes standard embedding. Giarrusso et al. [2019]'s current system is not ready for a (shallowly) embedded implementation because it targets an untyped calculus, assuming some global program transformations (namely, λ-lifting and a variant of A-normalization [Flanagan et al. 1993]—essentially, the call-by-value CPS transformation), which are not immediately ready for a shallowly embedded implementation in a typed language. Their accompanied implementation provides a deeply-embedded interface, which is designed for a demonstration of their idea, not user consumption: the abstract syntax is λ-lifted and A-normalized, with an interface that uses GADTs and exposes cache types, and the semantics is given by Haskell code generation in strings.

Embedding by Unembedding marks a road to a (shallowly) embedded implementation. First, its ability to access environments enables us to use the straightforward implementation that models a guest term $\Gamma \vdash e : A$ in $\exists C.(\llbracket \Gamma \rrbracket \to (\llbracket A \rrbracket, C)) \times ((\Delta \llbracket \Gamma \rrbracket, C) \to (\Delta \llbracket A \rrbracket, C))$. Second, we are able to represent the explicit sharing of computation naturally with a binding construct like **let**. This is crucial; in **`let`** `x = e` **`in`** `... x ... x ...`, interpretation of `e` happens twice in the guest language, while in `let' e (\x → ... x ... x ...)` the computation happens only once in the evaluation of this expression. Third, we can represent their "higher-order" combinators as binding constructs to enjoy higher-order programming without risking type safety. For example, consider the following, which implements an incremental computation of the cartesian product of given two sequences.

```
cartesianRecreation xs ys = concatMap (\x → map (\y → pair x y) ys) xs
```

(Note that we have eluded the type of this example as it contains specifics that we will introduce later.)

It is worth noting that there is a limitation on type safety coming from a gap in types: the original CTS [Giarrusso et al. 2019] targets an untyped language, while we here consider a typed EDSL. This gap becomes critical when we consider general λ-abstractions and applications, and we thus do not include λs in this section. We note, however, that we could have λs if we compromise type safety to some extent to use dynamic types for caches involved in function values (see Appendix D.2).

5.3 Unembedding Details of Interest

Unembedding CTS is fairly standard, following the general recipe in §4.4. However, there are two details of the unembedding that are worth exploring: the embedding of its binding construct, and its optimization to the performant implementation necessitated by its domain. These features will be considered in turn.

The Binding Construct. The core CTS language has first-order constructs: `unit`, `pair`, `fst` and `snd`; and one second-order binding construct: **`let`**. The semantic domain is created as a direct representation in Haskell of the type of CTS incremental functions:

```
data CTS env a = forall c. CTS (VEnv env → (a, c))              -- Initializer
                              ((DEnv env, c) → (Delta a, c))   -- Translator
```

These semantics encapsulate two evaluation modes: standard evaluation, or the initializer, that takes variable assignments (`VEnv`) to a result and an initial cache; and incremental evaluation, or the Δ-translator, that takes how variables have changed (`DEnv`) along with a cache and produces a change in the result (`Delta a`) along with a new cache. It involves two types of environment: a value environment (`VEnv`), which slightly differs from the one already introduced in that its container ensures that only types that have a notion of change, a `Delta`, are contained; and an environment of changes (`DEnv`), which stores these `Delta` values.

The semantic domain `CTS` is an instance of `Variables` as below.

```
instance Variables CTS where
  var = CTS (\(ECons (PackDiff x) _) → (x, ()))
            (\(ECons (PackDiffDelta dx) _, _) → (dx, ()))
  weaken (CTS f tr) = CTS f' tr'
    where
      f' (ECons _ g) = f g
      tr' (ECons _ dg, c) = tr (dg, c)
```

Here, `PackDiff` and `PackDiffDelta` wrap the values in the container type for their `VEnv` and `DEnv` environments; i.e., `VEnv = Env PackDiff` and `DEnv = Env PackDiffDelta`. The method `var` extracts the head from both value and change environment with the unit cache, `weaken` applies the initializer and the Δ-translator of the given CTS incremental function to the tails of the value and change environments, respectively.

Following the recipe, the **`let`** construct is unembedded as a semantic function over `CTS`.

```
letSem :: Diff a ⇒ CTS env a → CTS (a ': env) b → CTS env b
letSem (CTS i1 tr1) (CTS i2 tr2) = CTS i tr
```

```
  where
    i g =
      let (x, c1) = i1 g -- Get x::a with i1.
          (y, c2) =
            i2 (ECons (PackDiff x) g) -- Get y::b with i2 on env with x added.
      in (y, (c1, c2)) -- Result, y, is returned with new cache.
    tr (dg, (c1,c2)) = -- Same story but for incremental evaluation.
      let (dx, c1') = tr1 (dg, c1)
          (dy, c2') = tr2 (ECons (PackDiffDelta dx) dg, c2)
      in (dy, (c1', c2'))
```

Here, the constraint `Diff a` ensures that the type `a` has `Delta a` together with appropriate operations; this is actually what `PackDiff` and `PackDiffDelta` are wrapping. The function `letSem` is implemented by defining the initializer (`i`) and the Δ-translator (`tr`). They both work the same way: the first argument term is used to produce the `a`/`Delta a`, which is then added to the environment, so that the second argument term can produce the desired `b`/`Delta b`. The caches from the two argument terms are also combined. This is done in the simplest way: pairing them. Then, the semantic function is linked to the following HOAS syntax

```
let_ :: Diff a ⇒ exp a → (exp a → exp b) → exp b
```

with the lifting function.

```
let_ = liftSOn (ol0 :. ol1 :. End) letSem
```

We omit the discussions on the first order constructs `unit`, `pair`, `fst` and `snd` because their treatment is similar.

An astute reader would notice that we do not use caches in any essential ways in `letSem`. This is also true for the other constructs above: none of `unit`, `pair`, `fst` and `snd` uses the cache in any essential ways either. This is because these constructs are designed to glue together other incremental constructs for particular domains that may use the cache (e.g., sequence manipulating operations discussed below). Here, the extensibility of the finally-tagless representation [Carette et al. 2009] comes in handy: we can keep the core part small and then extend the core afterwards for problem domains of interest.

Being Performant. An unembedding of an incremental language would not fulfil its purpose if it was not performant. As such, with this unembedding, we took care to ensure that the incremental evaluation was faster than recomputation, and compared the performance of "eCTS" to the original.

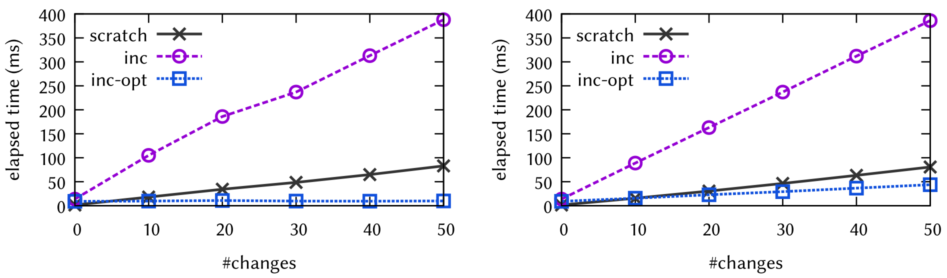
The example we will compare against is (again) the cartesian product. To implement this function, eCTS must first be expanded to include language constructs for manipulating sequences:

$$e ::= \cdots \mid \mathbf{emp} \mid \mathbf{concat}\ a \mid \mathbf{single}\ e \mid \mathbf{map}\ (x.e)\ e'$$

(`concatMap` from §5.2 can be implemented as the composition of `concat` and `map`). These constructs come with the following typing rules.

$$\frac{}{\Gamma \vdash \mathbf{emp} : \mathsf{Seq}\ A} \quad \frac{\Gamma \vdash e : \mathsf{Seq}\ (\mathsf{Seq}\ A)}{\Gamma \vdash \mathbf{concat}\ e : \mathsf{Seq}\ A} \quad \frac{\Gamma \vdash e : A}{\Gamma \vdash \mathbf{single}\ e : \mathsf{Seq}\ A} \quad \frac{\Gamma, x : A \vdash e : B \quad \Gamma \vdash e' : \mathsf{Seq}\ A}{\Gamma \vdash \mathbf{map}\ (x.e)\ e' : \mathsf{Seq}\ B}$$

Here, Seq a is a type for sequences of a-typed values. Notice that **map** is treated as a second-order construct, instead of a first-order construct involving function types. We do not encode λs directly in the DSL, but instead rely on the host language (Haskell)'s λs in programming. The expansion is



(a) changes to *xs* (b) changes to *ys*

Fig. 3. Experimental Results: incremental computation of *cartesian xs ys* when *length xs* = *length ys* = 200. Key: scratch = recomputation, inc = incremental computation, inc-opt = optimized incremental computation (optimizes away change propagation overhead).

achieved by repeating the recipe for these new constructs and adding their HOAS as a subclass of the typeclass of the core language. Full details of our implementation can be found in Appendix D.

The treatment of map, and the modularity and expandability of the language demonstrate a clear advantage of HOAS-based embedding [Carette et al. 2009]. We can extend the system more to implement a realistic example. In Appendix D.3, we implement an incremental version of HaXML filters [Wallace and Runciman 1999], where transformations are written based on filters of type `Tree` $\longrightarrow$ `Tree` and combinators working on filters are naturally higher-order functions.

Although the straightforward implementation discussed in this section has overheads, with the help of meta-programming techniques, a preliminary evaluation shows that our language has similar performance as the state of the art [Giarrusso et al. 2019]. The details of the implementation can be found in Appendix D.4. The key point is that it is still based on the same embedding approach, while using Template Haskell [Sheard and Jones 2002] to eliminate some of the overhead of unembedding, variable accesses and cache composition/decomposition, and it tries to remove the propagation of empty changes.

As a very preliminary experiment, we measure the performance of $f = \lambda z.\mathit{cartesian}\ (\mathit{fst}\ z)\ (\mathit{snd}\ z)$, which is taken from the original CTS paper [Giarrusso et al. 2019] (though we measure multiple changes instead of a single change as in the original paper). Specifically, for an input a_0 and a sequence of changes $\{da_i\}$, we compare the elapsed time of (1) from-scratch computation that computes $\{b_i = f\ \ a_i\}_{0 \le i \le n}$ for each intermediate inputs $a_{i+1} = a_i \oplus da_i$, and (2) incremental computation that first computes the result and a cache $(b_0, c_0) = f_{\mathrm{init}}\ a_0$, then propagates each changes $\{(db_i, c_i) = f_{\mathrm{tr}}\ (da_i, c_{i-1})\}_{1 \le i \le n}$ and finally apply the obtained changes $b_0 \oplus db_1 \oplus \cdots \oplus db_n$.

The experimental result is shown in Fig. 3. It is conducted on 13-inch MacBook Pro with 2.3 GHz Intel Core i7-1068NG7 CPU and 32 GB memory, using GHC 8.10.7 running on macOS 12.2.1. We use `Data.Sequence.Seq` in the package containers-0.6.5.1 to represent Seq. As we can see, the naive implementation (inc in the figure) is expectedly slow; the overhead of change propagation outweighs the recomputation cost. The interesting part is the comparison between inc-opt and scratch: the incremental version is slower for small number of changes due to the overhead, but as the number of changes grows, the potential of incremental computation becomes clear. It is also interesting to observe that there is a difference between changing *xs* (Fig. 3a) and changing *ys* (Fig. 3b). We think this is due to the effect of removing the unnecessary propagation of empty changes: when *xs* is changed, only the outer *map* and *concat* are affected, while when *ys* is changed, both inner and outer *map* and *concat* are affected, making the analysis less effective.

This result is very similar to what is reported in the original CTS paper [Giarrusso et al. 2019]. In summary, the CTS version exhibits an incremental behaviour: taking more time to start-up (the first execution) but a less or equal amount of time for update propagation, which is also the case for us. Moreover, in CTS, the update propagation is comparably faster for changes to *xs* and seems to take a constant time from their plot. On the other hand, the update propagation is slower for changes to ys and takes a linear time to the input size (the length of *xs* and *ys*). Again, this matches our result.

We anticipate that further optimization (as future work) will be able to remove sufficient unembedding overhead to close the gap observed above, which will push the result beyond the state-of-the-art in terms of performance.

6 CASE STUDY: BIDIRECTIONAL PROGRAMMING LANGUAGE

A *bidirectional transformation* is a piece of code that can be run in two directions and is expected to satisfy some round-tripping properties (i.e., transforming data to another form and back returns the original) [Foster et al. 2007]. This is a technique that allows a programmer to program two related operations at the same time, saving programming time, maintenance effort, and avoids the problem of separate implementations becoming out of sync.

The field of bidirectional programming has found numerous and wide-reaching applications for these including: data synchronization [Foster et al. 2007], gradual development of programs [Wang et al. 2010, 2013], deriving parsers from pretty-printers [Matsuda and Wang 2013, 2018b], Web/GUI programming [Perez and Nilsson 2015; Rajkumar et al. 2013], model synchronization in software engineering [Stevens 2008; Xiong et al. 2007; Yu et al. 2012], and view updating in database systems [Bancilhon and Spyratos 1981; Fegaras 2010; Hegner 2004; Tran et al. 2020].

This domain is a challenge to embed, because to properly develop bidirectional transformations, one requires specialized programming languages that enforce the round-tripping properties, preferably by construction.

In this section, we demonstrate that the approach of Embedding by Unembedding is able to address the long-standing challenge of embedding bidirectional languages with binders. In particular, we consider a certain class of lenses as a semantic domain, where a class of lens combinators are treated as semantic functions. The unembedding enables us to compose lenses as functions, in order to give extra programming flexibility compared with the standard embedding of lenses as a combinator language. The resulting embedded system is able to closely model the state-of-the-art stand-alone language HOBiT [Matsuda and Wang 2018d], creating eHOBiT.

After introducing lenses (§6.1), and highlighting the suitability of unembedding in this case (§6.2), this section showcases eHOBiT (§6.3), investigating its embedding, comparing it to the original, and discussing alternative embedding approaches.

6.1 Lenses

Formally, a lens between s and v is defined as a pairs of functions:

```
data Lens s v = L {get :: s → v, put :: s → v → s}
```

Borrowing the terminology of the original application in databases [Bancilhon and Spyratos 1981; Foster et al. 2007; Hegner 1990], s is called *source* and v is called *view*. The view is typically the result of some query on the source. The basic idea of a lens is: put merges changes to the view with the source. For example, for a lens fstL with get (x,y) = x, a corresponding put can be defined as put (_,y) x = (x,y). The asymmetric nature of get and put separates bidirectional transformations (lenses) from inverses, with the former being more general and able to handle non-injective functions.

6.2 Why Unembedding is Desired and Applicable

A lens must satisfy round-tripping to be considered as *well-behaved* [Foster et al. 2007]. Here we use the system adopted by HOBiT, which is a slightly weaker version of the original round-tripping [Bancilhon and Spyratos 1981; Foster et al. 2007; Hegner 1990]. The exact definition of the round-tripping laws is not important here, which we leave to Appendix F.1. The point is that our embedding is able to support the compositional reasoning of the property, and thus correctness by construction. This combination of embedding, compositional reasoning, and binding constructs has never been achieved in the bidirectional programming literature.

Open expressions matter in this domain, necessitating unembedding, because a natural approach to a bidirectional language with binders is to interpret an open expression as a lens between value environments and return values. For example, we interpret a guest term `fstL x` as a lens whose `get` takes `{x = (1,2)}` and returns 1, and whose `put` takes the same `x` and 3 to return `{x = (3,2)}`. This is similar to the case of reversible languages where we interpret open expressions as bijections between environments and return values (see, i.e., Yokoyama et al. [2011]). With binding expressions, we are not able to fix the number of free variables in giving its semantic domain. This is why Embedding by Unembedding allows for eHOBiT.

HOBiT manipulates algebraic datatypes, and provides a bidirectional case expression, which pattern-matches on a value and performs branching with the matched result in a bidirectional (round-tripping) way [Matsuda and Wang 2018d]. We could include patterns in a core syntax of a language to embedded, but this makes the system complicated. Instead, we focus on the core behaviour of the case expressions: decomposition of products and branching based on tags of sum-typed values. We mainly focus on the former, which has the guest syntax **`let`** `(x,y) = e1` **`in`** `e2`, and we call `unpair`. The latter construct can be handled similarly but contains subtlety in the problem domain (see three branching lens combinators in Foster et al. [2007]).

6.3 Unembedding Details of Interest

The feature of interest is, yet again, the binding construct: `unpair`. Its successful implementation, and more generally that of bidirectional case analysis, is what allows for the embedding of HOBiT. As such, its addition to a simple applicative lens language will be detailed, before comparing the new eHOBiT to the original.

Unpair. The simple bidirectional language that `unpair` will be added to consists of the following first-order constructs: primitive operators (`prim`), `unit` and `pair`. (The full syntax and typing rules for HOBiT can be found in Appendix C.2.)

The language is unembedding to the `LensTerm` semantic type:

```
newtype LensTerm env a = LT {runLT :: TEnv env → Lens (VEnv env) a}
type VEnv = Env Maybe
```

`LensTerm` realizes the idea of interpreting an open expression. It does this by building upon a lens between value environments and return values with two modifications: it takes `TEnv env` to perform intensional analysis on the structure `env` (to implement `var`), and `Maybe` is used as a container type of the environment to perform runtime linearity checking. We also use the functions: `merge :: VEnv env → VEnv env → VEnv env`, which merges component-wise value environments with checking of the linearity represented by `Maybe` and fails when it attempts to merge two `Just`s; and `unitEnv :: TEnv env → VEnv env`, which returns the value environments filled with `Nothing`, the unit element of `merge`. We shall omit its instance definition of `Variables`.

Fig. 4 shows the implementation of `unpair`'s semantic function. The main work is done by lenses manipulating and reformatting the environments: `rearrPair` takes the first two elements of an

```
unpairSem :: LensTerm env (a, b) → LensTerm (a ': b ': env) r → LensTerm env r
unpairSem t1 t2 = LT $ \g →
  let l1 = runLT t1 g
      l2 = runLT t2 (ECons Proxy (ECons Proxy g))
  in letLens l1 (rearrPair l2)

rearrPair :: Lens (VEnv (a ': b ': env)) r → Lens (VEnv ((a, b) ': env)) r
rearrPair l = L get' put'
  where
    get' (ECons (Just (a, b)) g) = get l (ECons (Just a) (ECons (Just b) g))
    put' (ECons (Just (a, b)) g) r =
      let ECons m (ECons n g') = put l (ECons (Just a) (ECons (Just b) g)) r
      in ECons (Just (fromMaybe a m, fromMaybe b n)) g'

letLens :: Lens (VEnv env) a → Lens (VEnv (a ': env)) b → Lens (VEnv env) b
letLens l1 l2 = L get' put'
  where
    get' g = get l2 (ECons (Just (get l1 g)) g)
    put' g b = let a = get l1 g
                   ECons m g1' = put l2 (ECons (Just a) g) b
                   g2' = put l1 g (fromMaybe a m)
               in merge g1' g2'
```

Fig. 4. Semantic Function for Unpairing

environment and pairs them; `letLens` is a lens that acts as a let for its two argument lenses. Since these are lenses, this manipulation is implemented as a `get` and a `put` function. These lenses both use `fromMaybe` to implement how HOBiT handles variables: an updated value of a variable defaults to the original value if the variable is unused. The `fromMaybe` function achieves this defaulting because when its first argument is `Nothing`, it returns its second argument. We are now ready to give `unpair` its HOAS representation:

```
unpair :: exp (a, b) → (exp a → exp b → exp r) → exp r
```

together with implementation.

```
unpair = liftSOn (ol0 :. ol2 :. End) unpairSem
```

This `unpair` destructs products, so to perform case analysis on all types, we also implement a function, `branch`, that does the same for sum types. Since algebraic datatypes can be expressed as recursive sum-of-product forms, this is a fairly comprehensive approach.

Comparison of eHOBiT *with* HOBiT. To demonstrate the programmability and expressiveness of the bidirectional EDSL, we perform a side-by-side comparison between HOBiT and our embedded version (Fig. 5). This example features further streamlining of the syntax:

```
case_ :: HOBiT exp ⇒ String → exp a → [Br exp a r] → exp r
(⟶) :: Pat '[] o a
        → (Func exp o (exp r), r → Bool, a → r → a) → Br exp a r
```

```
linesB :: BString → B[String]
linesB str =
  let (f, b) = breakNLB str
  in case b of
    '\n' : x : r →
        f : linesB (x : r)
        with (> 1) ∘ length
        by λb_. '\n' : ' ' : b

    _ →
        f : [ ]
        with (== 1) ∘ length
        by λb _. lastNL b

breakNLB :: BString → B(String, String)
breakNLBstr = ...
```

(a) Original HOBiT code (Matsuda and Wang [2018d, Fig. 2], with formatting)

```
linesB :: HOBiT e ⇒ e String → e [String]
linesB str =
  unpair (breakNLB str) $ \f b →
  case_ "linesB-case" b
  [ consP (constP '\n') (consP varP  varP) ⟶
    (\x r → f `consB` linesB (x `consB` r),
    (> 1) .length,
    \_b _ → '\n' : ' ' : _b),
    varP
    ⟶ (\_ → consB f nilB,
       (== 1) . length,
       \_b _ → lastNL _b) ]

breakNLB :: HOBiT e
         ⇒ e String → e (String, String)
breakNLB str = . . .
```

(b) Embedded eHOBiT code in Haskell

Fig. 5. Comparison of *linesB* in HOBiT and eHOBiT to showcase the faithfulness of the embedding.

These methods are derived from `unpair` and `branch`, where `Pat [] [a1, ..., an] a` represents patterns of type `a` that bind variables of types `a1 ... an`,[14] and `Func` is a type family defined as `Func exp [a1, ..., an] (exp r) = exp a1 → ... → exp an → exp r`. We leave the details of its definition to Appendix F.2.

The code comparison implements the running example from the HOBiT paper [Matsuda and Wang 2018d], which is a bidirectional version of the standard `lines :: String → [String]` function in Haskell that splits an input string by newlines as `lines "A\nB\nC" = ["A","B","C"]`. An intricate behaviour of `lines` is that it ignores the last newline as `lines "A\nB\nC\n" = ["A" ,"B","C"]`. We faithfully implement this behaviour and the bidirectional version `linesB` detects and preserves the existence of the last newline in the backward direction.

Unsurprisingly, HOBiT's version comes with less syntactic overhead, but thanks to derived functions including the pattern combinators, our EDSL version has a similar structure akin to the original HOBiT version. The syntactic overhead is not negligible, but is a reasonable price to pay for accessing host's functionalities, including typing checking, editor support, advanced compilation, interoperability, and so on.

There are methods existing in the literature for using host-level pattern matching in embedded DSLs, for example, that of McDonell et al. [2021]. The basic idea is to use host-level pattern matching to generate branches. For example, a destructor for `Either a b` takes two branches: one of type `f :: exp a → exp r` and the other of type `g :: exp b → exp r` in a finally-tagless EDSL. The branches can be unified as `Either (exp a) (exp b) → exp r`, and host level patterns used to write the unified branch as `\case { Left a → f a; Right b → g b }` . However, this approach assumes that the number of branches is fixed for a datatype, and the workarounds they suggest do not scale to recursive types like lists. Also, the approach does not work well with the widely-adopted top-down matching. In our EDSL, `case_` uses the top-down matching both in `get` and `put`, which plays an important role in the `linesB` example (Fig. 5b).

[14]The first argument of `Pat` is used for the difference list encoding of bindings inspired by Polakow [2015].

Unembedding is also not tied to the specific way that we have expressed bidirectional branching. Our (finally-tagless) embedding supports other approaches: the language can be extended rather easily to include such variants.

7 RELATED WORK

This section discusses work related to our framework; including places unembedding has previously been used, similar techniques, further details pertaining to the bidirectional application domain, and a review of languages that have been embedded in a similar manner.

Uses of Unembedding. Unembedding [Atkey 2009; Atkey et al. 2009] and similar techniques have been used in embedding for intensional program manipulation such as code generation. The high-performance array computing EDSL `accelerate` uses the technique to convert a surface language to an internal representation based on de Bruijn terms, with sharing recovery [McDonell et al. 2013]. The technique has been used also for having a better interface for an SMT solver [Stewart 2010], and a better syntax for probabilistic programs to be converted to OpenCL code [Werk et al. 2012]. Matsuda and Wang [2018b] showed how unembedding can be used to embed a reversible language FliPpr [Matsuda and Wang 2013, 2018c], in which users describe pretty-printers with annotations to obtain parsers corresponding to the pretty-printers, in the form of context-free grammars with semantic actions. In contrast, our contribution lies in demonstrating the usefulness of the unembedding for implementing advanced semantics beyond code manipulation. We also provide a general recipe for Embedding by Unembedding that others may build their embedding implementation upon.

Intensional Analysis on Open HOAS Terms. Babybel [Ferreira and Pientka 2017] is a meta-programming framework that supports manipulation of open terms in the HOAS representation, based on contextual type theory [Pientka 2008]. Roughly speaking, a contextual type $[\Gamma \vdash A]$ represents open guest terms that have type A under context Γ. They also provide a first-class substitution so that we can manipulate open terms without worrying about tedious shifting and renaming (both affect Γ). There is certain similarity between theirs and ours: both use HOAS and handle open terms. However, there is a critical difference: their HOAS lies in a guest language's intensional function space, while ours lies in the host language (Haskell)'s extensional one. For example, terms like `lam (\x → app (lam (\y → x)) x)` is a guest term instead of a host term in their system. This gives us intensional analysis on such guest terms: for example, they allow patterns like `lam (\x → m)` to match against the previous term to bind `m` with an open term `app (lam (\y → x) x)`. Also, unlike the tagless final style [Carette et al. 2009] and PHOAS [Chlipala 2008], the clear separation between the guest and host systems does not require polymorphism to avoid exotic terms. However, having guest terms with λs requires a separate embedding technique (though λs are only binding constructs in the system), if we use the idea in main-stream languages such as Haskell. For example, in Babybel, they use OCaml's AST-level pre-processor (PPX) to translate contextual-typed terms into de Bruijn indexed terms. In Haskell, we could use Template Haskell [Sheard and Jones 2002], but Embedding by Unembedding, with future extensions, could serve as another option.

Bidirectional EDSLs. Embedding bidirectional languages to benefit the host's programmability has been a longstanding problem faced by the research community. As discussed in §6, the standard approach of representing bidirectional building blocks (such as lens [Foster et al. 2007]) as black boxes is unsatisfactory for programmability, limiting programmers to unfamiliar combinators and unconventional program structures. Many efforts have been made to address this problem. Voigtländer [2009]'s semantic bidirectionalization interprets polymorphic functions as bidirectional transformations, which is convenient as programmers write host language's polymorphic functions, but with quite limited expressiveness—there is no way to specify refined bidirectional behaviours. The applicative lens framework [Matsuda and Wang 2015, 2018a] extends semantic bidirectionalization to support `prim`, `unit` and `pair` in §6 to use the host language to compose lenses. However, its expressiveness is still limited, as the approach does not scale to the lens combinators. The same authors address the expressiveness issues in HOBiT [Matsuda and Wang 2018d], which until now was a stand-alone language.

Applicative Lenses. By using the lens variant of `runOpenN`, `runOpenAppLensN` adapted from the `runOpenN` provided by our framework and introduced in §4.4, we can construct a lens from a polymorphic function; e.g., we can convert `reverse :: [a] → [a]` into a lens by `runOpenAppLensN (foldr consB nilB . reverse) :: Lens [a] [a]`. That is, we can mimic Voigtländer [2009]'s semantic bidirectionalization. However, our EDSL is not limited to polymorphic functions; e.g., we can apply concrete lenses by `prim :: HOBiT exp ⇒ Lens a b → exp a → exp b`, a construct that applies a lens to a result. In fact, this simple DSL is as expressive as the applicative lens framework [Matsuda and Wang 2015, 2018a], which uses Yoneda embedding instead of the unembedding, and thus does not scale to second-order constructs.

Similarly Embedded Languages. Kiselyov [2019] discussed an approach to implement nondeterministic computation by embedding a core first-order nondeterministic DSL (which can have second-order constructs), without monads. Advantages of this approach is that one can focus on a simple DSL, which is easier to reason about and supports various interpretations (e.g., abstract interpretation and staged interpretation) that are not possible with monads. The HOAS representation of the syntax plays an important role, as it enables us to use host's λs and higher-order functions to define programs in the DSL. Embedding by Unembedding framework encourages this design, as it enables us to embed a core first-order system with HOAS, where previously such an implementation was not obvious, as demonstrated in §6 and §5. PyRo (https://pyro.ai/), a popular probabilistic programming library in Python, shares a similar design [Bingham et al. 2021; Obermeyer et al. 2020] to Kiselyov [2019], though without using HOAS. This fact suggests a potential application of Embedding by Unembedding.

For a comparison of the unembedded CTS with the original and work related to CTS, see Appendix E.

8 FUTURE WORK

Embedding by Unembedding opens the door to embedding a whole new host of languages. We plan to make use of this technique, push it to its limit, and work out where its boundaries lie. From a practical perspective, our future work plans to put Embedding by Unembedding to work. Initially on targets we feel ripe for embedding (e.g., automatic differentiation), but then onto those that may present more of a challenge (perhaps in a linear setting). Then from a theoretical standpoint, we feel that there is much to gain from exploring the theoretical underpinnings of this technique.

Automatic Differentiation. Another interesting application of Embedding by Unembedding is automatic differentiation for higher-order languages. Having a correct automatic differentiation for a higher-order language is a challenging topic [Ehrhard and Regnier 2003; Huot et al. 2020; Krawiec et al. 2022; Mazza and Pagani 2021; Sherman et al. 2021; Vytiniotis et al. 2019; Wang et al. 2019], especially for the reverse mode, and Embedding by Unembedding sheds a new light on it: we can focus on the core first-order fragment while embedding allows us to use host-level higher-order functions—a system as a whole constitutes a differential higher-order language. Specifically, we can keep a straightforward and easy-to-reason semantics $[\![\mathbb{R}^a \to \mathbb{R}^b]\!] = \mathbb{R}^a \to (\mathbb{R}^b, \mathbb{R}^b \multimap \mathbb{R}^a)$ for the reverse-mode automatic differentiation, where $\multimap$ denotes a linear transformation and thus $\mathbb{R}^b \multimap \mathbb{R}^a$ represents the Jacobian matrix (at the input). By restricting the EDSL type to $\mathbb{R}$ or their products, $[\![\Gamma]\!] \simeq \mathbb{R}^a$ and $[\![A]\!] \simeq \mathbb{R}^b$ hold for some a and b to allow the semantic domain as above. Now, Embedding by Unembedding enables higher-order programming in a host language, without risking correctness. As mentioned earlier, we conjectured that the unembedding essentially models a system as a whole in a presheaf category. Thus, in this sense, the embedding-by-unembedding approach would be similar to λ_S [Sherman et al. 2021], which uses a presheaf category to model their higher-order differentiable language, but the unembedding has not been used for their accompanied implementation. Exploring embedded differentiable programming languages is an interesting future direction. One of the challenges in the application is performance—the overhead introduced by the unembedding is unwelcome especially in this problem domain. We believe that staged programming can remove the overhead (see Appendix D.4 for such a discussion for the incremental calculus).

Probabilistic Programming. We are also interested in building upon the work of Nguyen et al. to see how Embedding by Unembedding compares to their use of effect handlers to implement the advanced semantics of probabilistic languages, including using host language variables for random or optimisable variables.

Linear Unembedding. A future direction is to develop a linear version of the unembedding. As mentioned in §6, linear systems fit better for reversible and bidirectional computations. Since a practical linear programming language is now available as Linear Haskell [Bernardy et al. 2018], unembedding that utilizes the host's linear parametricity attracts more practical interest. Another interesting future direction is to apply Embedding by Unembedding to more practical situations. One candidate is automatic differentiation.

Criteria for Unembedding. In §4.1, we mentioned that our approach can be applicable to simply-typed languages where the typing rules and constructs have the designated forms and come with compositional semantics. This description, while intuitive, has points to clarify: for example, why a method $lam :: (e\ a \to e\ b) \to e\ (a \to b)$ has a polymorphic type (abstracting a and b), while we restrict a language to be simply typed; and what are the HOAS representations and semantic functions that corresponds to a construct in general? Specifically, our target languages are the second-order algebras [Fiore et al. 1999] as presented in Hamana [2018, 2019], where a symbol (operator) can have a (predicative) polymorphic type. What this means and why this is the case is the subject of future work.

Correctness. The correctness of our method, Embedding by Unembedding, intuitively says that the interpretation via `runOpen` coincides with the interpretation of the corresponding de Bruijn term. This is a generalization of the Atkey [2009] proof mentioned in §4.4, the formalization of which and expansion to also cover PHOAS is future work. Although the original proof does not cover PHOAS directly, showing that the PHOAS and the tagless-final style are isomorphic by using parametricity [Reynolds 1983; Wadler 1989] is straightforward.

Categorical Exploration. We showed in §6 that embedding lenses by unembedding leads to an EDSL as expressive as HOBiT [Matsuda and Wang 2018d]. This correspondence is not accidental. For terminating computations, we conjecture that the idea of unembedding [Atkey 2009] is to interpret HOAS terms in a presheaf $[C^{\mathrm{op}}, \mathbf{Set}]$, i.e., the category of functors from the opposite category of C to the category of the total (hence, terminating) functions, where C denotes the semantic domain of the (non-HOAS) terms. HOBiT adopted a staged semantics [Matsuda and Wang 2018d] to evaluate away λs in the first stage and leave bidirectional constructs for the second stage. It is known that the semantics of two-level languages can be modelled in a presheaf $[C^{\mathrm{op}}, \mathbf{Set}]$

(again, assuming termination of the first stage computation), where C denotes the semantic domain for the second stage [Moggi 1998]. This suggests that a language with a presheaf semantics (such as λ_S [Sherman et al. 2021]) might be embedded by unembedding just by embedding the “core part” of the semantic domain (i.e., C). Formalization and further investigation of this is an interesting future direction. What this means and why this is the case is the subject of future work.

Another conjectured relationship to category theory is a connection between the criteria in §4.1 and second-order algebras [Fiore et al. 1999; Hamana 2018]. The typing rule of the second-order construct **con′** in §4.1 has the same form as the one for an operator in a typed second-order algebra used in Hamana [2018]. Fiore et al. [1999] discuss the semantics of second-order algebra in a functor category $[\mathbb{F}, \mathbf{Set}]$. Here, $\mathbb{F}$ is a category of manipulations of indices, which can also be seen as manipulation of environments, and in fact include weakening, exchange, and contraction. In §4.2, we conjectured a connection between `Variables` and $\mathbb{F}$; they both characterize manipulations of environments, though `Variables` contains the minimal operations needed for Embedding by Unembedding. Thus, it is plausible to conjecture that we can have a unified framework of Embedding and Unembedding based on the second-order algebra and show that the finally-tagless [Carette et al. 2009] HOAS is an initial representation of second-order algebras. Then, we also expect that a language presheaf semantics such ones mentioned above can be modelled by a functor $\mathbb{F} \to C^{\mathrm{op}}$, which can turn a presheaf $[C^{\mathrm{op}}, \mathbf{Set}]$ into the functor category $[\mathbb{F}, \mathbf{Set}]$ of interest.

9 CONCLUSION

In this paper, we propose an embedding method called “Embedding by Unembedding”, which is based on the unembedding [Atkey 2009; Atkey et al. 2009] technique that originally gives a bijection between the HOAS representation and the de Bruijn term representation of a language. We identify that HOAS-based embedding does not directly support compositional semantics defined on terms-in-contexts, and design a general strategy for embedding of DSLs targeting advanced semantic domains. We also give two substantial case studies in the areas of incremental computation and bidirectional transformation, showing the effectiveness and applicability of our general approach.

ACKNOWLEDGMENTS

We thank Eijiro Sumii for useful feedback on a preliminary version of this research and for accepting the second-author’s visit in his group. We also thank Bristol Programming Languages research group for creating the wonderful environment for this work. This work was partially supported by JSPS KAKENHI Grant Numbers JP19K11892, JP20H04161 and JP22H03562, JSPS Bilateral Program, Grant Number JPJSBP120199913, EPSRC Grant *EXHIBIT: Expressive High-Level Languages for Bidirectional Transformations* (EP/T008911/1), Royal Society Grant *Bidirectional Compiler for Software Evolution* (IES\R3\170104).

A IMPLEMENTATION OF LIFTSO

A complication about the definition of `liftSO` is that it must deal with arbitrary number of arguments of a construct, where each argument can bind arbitrary number of variables.

For uniform manipulation of arguments, we first introduce the kind `Sig2 k`, which describes arguments of second order constructs:

```
data Sig2 k = [k] :⤳ k
```

Here, a type `[a1,...,an] :⤳ b :: Sign2 k` represents arguments that binds variables of types `[a1,...,an]` and have type `b`. This type is intended to be lifted to the type level by Haskell’s `DataKinds`. This enables the definition of a type for unembedded interpretations:

```
data URep (sem :: [k] → k → Type) (s :: Sig2 k) where
  UR :: TEnv as → (Env (EnvI sem) as → EnvI sem b) → URep sem (as ':↝ b)
```

and one for corresponding semantic objects.

```
data TermRep (sem :: [k] → k → Type) (env :: [k]) (s :: Sig2 k) where
  TR :: sem (Append as env) b → TermRep sem env (as ':↝ b)
```

Here, `URep` takes an value-level representation of `as` in addition, as otherwise we cannot know how many arguments we pass to a function of type `(Env (EnvI sem) as → EnvI sem b)` to obtain a semantic object `(sem (Append as env) b)`. Notice that type information is not available in runtime in Haskell in general, unless we explicitly convey them. Together these allow for the definition of `liftSO` (Fig. 6). Notice that the `Env (URep sem) ss → EnvI sem r` part represents an unembedded interpretation (with value-level representation of binding types), while `(forall env . Env (TermRep sem env) ss → sem env r)` the part represents the corresponding semantic function.

However, it is almost certain that users do not want to use such constructors explicitly for lifting, and hence we leverage Haskell's type-level programming to abbreviate this pain. As seen in the main body of the paper, lifting can be done as `liftSOn (ol0 :. ol1 :. End) letSem`, where users only need to specify the numbers of variables bound by each argument of a construct. The first step is to get rid of `Env` to represents arguments, and hence we prepare the following type families for "curried" representations.

```
-- Corresponds to (forall env. Env (TermRep sem env) ss → sem env r)
type family FuncTerm (sem :: [k] → k → Type) (env :: [k])
                     (ss :: [Sig2 k]) (r :: k) | r → sem env r where
  FuncTerm sem env '[] r = sem env r
  FuncTerm sem env ((as ':↝ a) ': ss) r = sem (Append as env) a
                                          → FuncTerm sem env ss r

-- Corresponds to Env (URep sem) ss → EnvI sem r
type family FuncU (sem :: [k] → k → Type) (ss :: [Sig2 k])
                  (r :: k) = res | res → sem r where
  FuncU sem '[] r = EnvI sem r
  FuncU sem ((as ':↝ a) ': ss) r = Func (EnvI sem) as (EnvI sem a)
                                   → FuncU sem ss r
```

With these type families, we can define `liftSOn` as below

```
liftSOn :: forall sem ss r. Variables sem ⇒ Dim ss
        → (forall env. FuncTerm sem env ss r) → FuncU sem ss r
liftSOn ns f =
  let h :: forall env. Env (TermRep sem env) ss → sem env r
      h = fromFuncTerm f
  in toFuncU ns (liftSO @sem h)
```

The function `liftSOn` takes `Dim ss`, which is used to provide a value-level representation of `asi` required for `URep`, where `ss = [as1:↝b1, ..., asn:↝bn]`. Specifying each `asi` is actually tedious, but an observation suggests that the length of each `asi` is enough for runtime. The GADT `Dim` is defined as below, leveraging the fact.

```
liftSO :: forall sem ss r. Variables sem ⇒
  (forall env. Env (TermRep sem env) ss → sem env r)
  → Env (URep sem) ss → EnvI sem r
liftSO f ks = EnvI $ \g → f (mapEnv (conv g) ks)
  where conv :: TEnv env → URep sem s → TermRep sem env s
        conv g (UR g1 k) = TR $ cnv g g1 k
        cnv :: TEnv env → TEnv as → (Env (EnvI sem) as → EnvI sem a)
            → sem (Append as env) a
        cnv g g1 k = let {ex_g = appendEnv g1 g; xs   = mkXs g g1 ex_g}
                     in runEnvI (k xs) ex_g
        mkXs :: TEnv env → TEnv as' → TEnv (Append as' env) → Env (EnvI sem) as'
        mkXs _ ENil _ = ENil
        mkXs p (ECons _ as) tg@(ECons _ tg')
          = let x = EnvI $ \g' → weakenMany tg g' var
            in ECons x (mkXs p as tg')
```

Fig. 6. Implementation of `liftSO`

```
data Dim (ss :: [Sig2 k]) where
  End  :: Dim '[]
  (:.) :: OfLength as → Dim ss → Dim ((as ':↝ a) ': ss)

data OfLength as where
  LZ :: OfLength '[]
  LS :: OfLength as → OfLength (a ': as)
```

The function `liftSO` uses the functions `toFuncU` and `fromFuncTerm`, which have the following types.

```
toFuncU :: Dim ss → (Env (URep sem) ss → EnvI sem r) → FuncU sem ss r

fromFuncTerm :: FuncTerm sem env ss r
             → Env (TermRep sem env) ss → sem env r
```

We omit the definitions of them as they are straightforward.

We also provide constants `olk` defined as `ol0 = LZ`, `ol1 = LS LZ`, `ol2 = LS ol1` ... for small `k` for convenience.

B UNEMBEDDING THE INCREMENTAL λ-CALCULUS

We clarify here that, unlike the cache-transfer-style variant [Giarrusso et al. 2019] (discussed in Section 5), embedding ILC itself is possible even without the Embedding by Unembedding (tupling can be applied as discussed in §5.2). Our primary focus here is to provide another example of the unembedding technique.

B.1 Unembedding ILC

Step 1: Identify the semantic domain. The semantics of `ILC` depends on having infrastructure for representing changes. §5 fully implements such an infrastructure, which we will borrow for this example. The change infrastructure consists of a `Diff` typeclass, whose member a have an associated `Delta a` type that represents changes (or deltas) that can be made to those types.

With the ability to describe changes, `ILC` terms should be interpretable in two ways:

(1) Using standard evaluation, where variable assignments map to results.
(2) Using differential evaluation, where inputs and input deltas are mapped to output deltas.

These semantics can be packaged up into the following concrete datatype:

```
data ILC env a = Inc
  { sEval :: VEnv env → a, dEval :: VEnv env → DEnv env → Delta a }
```

Here, `sEval` performs the standard evaluation, mapping variable assignments (`VEnv`) to a result; `dEval` does the differential evaluation, mapping the inputs (`VEnv`) and changes in them (`DEnv`) to an output change. For readability, we simply (ab)use the `Env` constructors from §3 to manipulate these environments, as their actual representations are not important in this section (we will revisit them in §5).

Since both parts of this semantic type feature an environment, it is easy to see that it is a member of the `Variables` typeclass. For `var`, `sEval` can extract the first value from the `VEnv`, and `dEval` can ignore the `VEnv` and extract the first delta from `DEnv`. For `weakenOne`, new `sEval` and `dEval` can be created that just ignore the first element of the environments:

```
instance Variables ILC where
  var :: ILC (a : as) a
  var = Inc (\  (ECons (PackDiff a) _) → a)
            (\_ (ECons (PackDiffDelta da) _) → da)
  weaken :: ILC as a → ILC (b : as) a
  weaken (Inc s d) = Inc s' d'
    where
      s' (ECons _ g) = s g
      d' (ECons _ vg) (ECons _ dg) = d vg dg
```

Step 2: Prepare semantic functions for each construct. (The core part of) `ILC` consists of: `unit`, `pair`, `fst_`, `snd_`, and `let_` constructs. Note that `let_` is the only second-order construct of this language.

The semantic function for `unit` is a single `ILC` type where the type in focus is `()`. This means that both `sEval` and `dEval` ignore their arguments and just return `()` and the unit delta respectively:

```
unitSem :: ILC env ()
unitSem = Inc (const ()) (\_ _ → UnitDelta)
```

Such semantic functions can also have arguments, for example, `pairSem :: ILC env a → ILC env b → ILC env (a, b)` takes one `ILC` term for the first element of the tuple, and one for the second. It runs both argument expressions, pairing together both the value and delta results. Those for `fst_` and `snd_` behave similarly: running the pair term and then throwing away either the first or second element of the result. That for `let_` is more interesting, as it is a second-order construct meaning that it manipulates the environments. It is defined by creating new `sEval` and `dEval` functions:

```
letSem :: Diff a ⇒ ILC env a → ILC (a ': env) b → ILC env b
letSem (Inc sa da) (Inc sb db)
  = Inc (\g →
          let x = sa g                    -- unpack a
              e' = ECons (PackDiff x) g -- add a to env
```

```
      in sb e'                          -- extract b with new env
    )
    (\vg dg →
      let x = sa vg                              -- unpack a
          g' = ECons (PackDiff x) vg             -- add a to env
          da' = da vg dg                         -- unpack da
          dg' = ECons (PackDiffDelta da') dg     -- add da to env
      in db g' dg'                               -- extract b with new env
    )
```

Here, the differential evaluation uses both sa and da to create a-typed values and Delta a-typed values to be added to the corresponding environments needed for db, which mirrors the standard evaluation.

Step 3: Provide the HOAS representation of the syntax. For ILC, this means that the language typeclass (ILChoas) has five methods. One for each construct:

```
class ILChoas exp where
  unit :: exp ()
  pair :: exp a → exp b → exp (a, b)
  fst_ :: exp (a, b) → exp a
  snd_ :: exp (a, b) → exp b
  let_ :: Diff a ⇒ exp a → (exp a → exp b) → exp b
```

Notice how the types of each method match those of the semantic functions, but with ILC abstracted out into e, and the environments omitted as the host language is handling them.

Step 4: Implement semantics through appropriate liftings. For ILC, this means that ILC is wrapped in EnvI so that it can be an instance of ILChoas:

```
instance ILChoas (EnvI ILC) where
  -- constructs without binders ⇒ we use liftFO
  unit = UE.liftFO0 unitSem -- 0 args ⇒ 0
  pair = UE.liftFO2 pairSem -- 2 args ⇒ 2
  fst_ = UE.liftFO1 fstSem  -- 1 arg  ⇒ 1
  snd_ = UE.liftFO1 sndSem
  -- our binding construct ⇒ we use listSO
  let_ = UE.liftSOn (ol0 :. ol1 :. End) letSem
```

Here, unit is defined as the lifting of unitSem. Because unit is first-order, one of the liftFO functions is used, specifically liftFO_0 because unit has no arguments. The rest of the first-order constructs are defined similarly, differing only in their choice of lifting function, which depends on how many arguments the construct in question has. As a second-order construct, let_ is different. Now its arguments can have arguments. To allow for this, liftSO takes a description of what arguments are expected: how many there are and how any arguments (bindings) they themselves have. This description is a list, with each element representing an argument and how many arguments it itself takes. In the case of let_, there are two arguments: the first taking no arguments (ol0); the second taking one (ol1).

Step 5: Put the language to work! Putting the embedded ILC to work builds on top of runOpenILC:

```
runILC :: Diff a ⇒ ILC '[a] b → a → (b, Delta a → Delta b)
runILC t x = let
    vg = ECons (PackDiff x) ENil
    r = sEval t vg
    f da = let dg = ECons (PackDiffDelta da) ENil
           in dEval t vg dg
    in (r, f)
```

This runILC operates on the output of runOpen and produces two functions: an ordinary transformation and its corresponding change translator. Note that both the input and output types are members of the Diff typeclass meaning that they can have changes applied to them by the change infrastructure.

This performs incremental computation! Consider this example term that for some tuple x will take the first element, and pair it with itself:

```
term x = let_ (fst_ x) (\y → pair y y)
```

It can be executed both normally by extracting the initial result, or incrementally by extracting the change function:

```
> fst (runILC term (7,6))
(7,7)
> (snd (runILC term (7,6))) (PairDelta (DInt 3) (DInt 2))
PairDelta +3 +3
```

In the latter example, the input tuple gets three added to its first element, and two to its second. This gets correctly translated to an output change where three would be added to both elements.

B.2 Extending ILC with Sequences

Embedding by Unembedding shares the modularity and extensibility of tagless-final style. To add new constructs, the recipe can just be repeated, and the language can be created in a modular manner, with different constructs grouped into different and dependant typeclasses. For example, to add sequences[15] to ILC the process unfolds as follows:

Step 1: Identify the semantic domain. When it comes to extending the language with constructs, this step is ready done. There is already a semantic type for ILC.

Step 2: Prepare semantic functions for each construct. Introducing sequences expands the language's constructors with: empty, singleton, concat, and map. Reifying these constructs into the embedding again consists of writing representative functions over the semantic type. For example, empty is defined similarly to unit:

```
emptySem :: ILC env (Seq a)
emptySem = Inc
  (const S.empty)    -- const the empty sequence
  (\_ _ → DSeq [])  -- const the empty change on sequences
```

Its semantics lift the empty sequence and its Delta into the ILC type. The rest of the semantic functions have the following types:

[15]Sequences are a speedier version of lists in Haskell, and as ILC is about efficiency they are chosen.

```
singletonSem :: ILC env a → ILC env (Seq a)
concatSem    :: ILC env (Seq (Seq a)) → ILC env (Seq a)
mapSem :: (Diff a, Diff b)
       ⇒ ILC (a : env) b → ILC env (Seq a) → ILC env (Seq b)
```

Exploring their definitions, brings to light different patterns in the way certain constructs behave depending on whether they are constructors, destructors or binders. The constructor `singleton` is similar to `pair` in that it takes an argument to produce the element of the list, then wraps it in the functor. The destructor `concat` can be compared to `fst_` or `snd_`, as it runs its argument and then changes the structure of the result (being sure to propagate this change to the `Delta` type). `map` is the most interesting as it can be defined in terms of `let_` to deal with the environment:

```
mapSem f xs = letSem xs (mapSem' f)
```

Then `mapSem' :: (...) ⇒ ILC (a : env) b → ILC (Seq a : env) (Seq b)` behaves similarly to `concat`, running the first term, then adjusting the result.

Step 3: Provide the HOAS representation of the syntax. This step is identical to adding more constructs to a normal tagless-final embedding: adding a new typeclass with those constructs as methods.

```
class ILChoas exp ⇒ ILCSeq exp where
  empty     :: exp (Seq a)
  singleton :: exp a → exp (Seq a)
  concat    :: exp (Seq (Seq a)) → exp (Seq a)
  map :: (Diff a, Diff b) ⇒ (exp a → exp b) → exp (Seq a) → exp (Seq b)
```

This typeclass has been made dependent on `ILChoas` so that it is clear that it is an extension of the language. The correspondence between the method types and the types of the semantics functions is again evident.

Step 4: Implement semantics through appropriate liftings. Instantiating `ILC` as member of this new typeclass follows the same pattern: wrapping it in the `EnvI` type, then defining the methods as the semantic functions using the provided lifting functions. Which lifting function to use is again decided by whether the construct is first or second order and how many arguments it has:

```
instance ILCSeq (EnvI ILC) where
  empty     = UE.liftFO0 emptySem
  singleton = UE.liftFO1 singletonSem
  concat    = UE.liftFO1 concatSem
  map       = UE.liftSOn (ol1 :. ol0 :. End) mapSem
```

Step 5: Put the language to work! New constructs allow for more exciting terms to interpret, and with no change in semantics, `runILC` and `runOpen` can still be used together to produce an initial result and a change function. For example, consider the term `cartesian`, which produces the cartesian product of two sequences. It can be run in an expression where the free variables is a pair of two sequences as described by `caInc`:

```
cartesian xs ys = concat (map (\x → map (\y → pair x y) ys) xs)
caInc = runILC (runOpen (\zs → cartesian (fst_ zs) (snd_ zs)))
```

```
> let (r, i) = caInc (S.fromList [1..3 :: Int], S.fromList [1..3 :: Int])
> r
```

```
fromList [(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)]
> i (PairDelta (DSeq [Ins 0 0]) mempty)
DSeq [Ins 0 (0,1), Ins 0 (0,2), Ins 0 (0,3)]
```

When run on the sequence that enumerates 1 to 3 paired with itself, the initial result `r` is indeed the cartesian product of those two sequences, and if the input tuple is updated to add 0 to the first sequence, this change is correctly converted to add the pairs `(0,1)`, `(0,2)`, and `(0,3)` to the result. Note that `S.` deliminates sequence functions from the Haskell `Data.Sequence`, and `DSeq` describes changes on a sequence as a list of insertions (`Ins`), deletions or replacements.

Thus `ILC` has been embedded and extended with sequences. It is worth noting that there is a known problem with the `ILC`. It is only efficient for certain "self-maintainable" functions. The full details of this inefficiency can be found in the sequel `ILC` paper that introduces the `CTS` [Giarrusso et al. 2019] variant, but long story short some functions written in `ILC` can introduce recomputation. In fact, the map semantic function for `ILC` cannot be implemented without recomputation. Happily, the `CTS` variant of `ILC` is able to be embedded, and that is exactly what our first case study in §5 does.

C SYNTAX AND TYPING RULES OF LANGUAGES TO EMBED

Here, we summarize the syntax and typing rules of the embedded languages discussed in the paper.

C.1 Syntax and Typing Rules of ILC/CTS DSL

The basic syntax is as below.

$$e ::= () \mid (e_1, e_2) \mid \mathbf{fst}\ e \mid \mathbf{snd}\ e$$

For each construct, we have the following type rules.

$$\frac{}{\Gamma \vdash () : ()} \qquad \frac{\Gamma \vdash e_1 : A_1 \qquad \Gamma \vdash e_2 : A_2}{\Gamma \vdash (e_1, e_2) : (A_1, A_2)} \qquad \frac{\Gamma \vdash e : (A_1, A_2)}{\Gamma \vdash \mathbf{fst}\ e : A_1} \qquad \frac{\Gamma \vdash e : (A_1, A_2)}{\Gamma \vdash \mathbf{snd}\ e : A_2}$$

$$\frac{\Gamma \vdash e_1 : A \quad \Gamma, x : A \vdash e_2 : B}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : B}$$

In §5, we extend the language as

$$e ::= \cdots \mid \mathbf{emp} \mid \mathbf{concat}\ a \mid \mathbf{single}\ e \mid \mathbf{map}\ (x.e)\ e'$$

together with the typing rules below.

$$\frac{}{\Gamma \vdash \mathbf{emp} : \mathsf{Seq}\ A} \qquad \frac{\Gamma \vdash e : \mathsf{Seq}\ (\mathsf{Seq}\ A)}{\Gamma \vdash \mathbf{concat}\ e : \mathsf{Seq}\ A} \qquad \frac{\Gamma \vdash e : A}{\Gamma \vdash \mathbf{single}\ e : \mathsf{Seq}\ A} \qquad \frac{\Gamma, x : A \vdash e : B \quad \Gamma \vdash e' : \mathsf{Seq}\ A}{\Gamma \vdash \mathbf{map}\ (x.e)\ e' : \mathsf{Seq}\ B}$$

Constructs to add depend on problems; for examples, in §D.3, to realized an incremental (but simplified) version of HaXML filters [Wallace and Runciman 1999], we add constructs for trees together with additional operations (such as *filter*) on sequences.

C.2 Syntax and Typing Rules of HOBiT to Embed

The syntax for the embedded HOBiT is as below.

$$e ::= \ell\ e \mid () \mid (e_1, e_2) \mid \mathbf{let}\ (x, y) = e_1\ \mathbf{in}\ e_2 \mid \mathbf{case}\ e_0\ \{x_i.e_i\ \mathbf{with}\ \phi_i\ \mathbf{by}\ \rho_i\}_i$$

The constructs come with the following type rules.

$$\frac{\Gamma \vdash e : A \quad \ell : \mathsf{Lens}\ A\ B}{\Gamma \vdash \ell\ e : B} \qquad \frac{}{\Gamma \vdash () : ()} \qquad \frac{\Gamma \vdash e_1 : A_1 \qquad \Gamma \vdash e_2 : A_2}{\Gamma \vdash (e_1, e_2) : (A_1, A_2)}$$

$$\frac{\Gamma \vdash e_1 : (A, B) \quad \Gamma, x : A, y : B \vdash e_2 : C}{\Gamma \vdash \mathbf{let}\ (x, y) = e_1\ \mathbf{in}\ e_2 : C}$$

$$\frac{\Gamma \vdash e_0 : A_1 + \cdots + A_n \quad \{\Gamma, x_i : A_i \vdash e_i : B \quad \phi_i : B \to \mathsf{Bool} \quad \rho_i : A_1 + \cdots + A_n \to B \to A_i\}_i}{\Gamma \vdash \mathbf{case}\ e_0\ \{x_i.e_i\ \mathbf{with}\ \phi_i\ \mathbf{by}\ \rho_i\}_i : B}$$

We do not have injections ($\mathsf{InL}\ e$/$\mathsf{InR}\ e$) and unit decompositions ($\mathbf{let}\ () = e_1\ \mathbf{in}\ e_2$) in the syntax (and type rules), as the lens application construct $\ell\ e$ is so strong to express them as derived forms.

D MORE DETAILS OF EMBEDDED INCREMENTAL LANGUAGE

D.1 Concrete Definition of `mapSem'`

Fig. 7 shows the definition of `mapSem'`. We say again that the fact that the changes (specifically, `DEnv`) form a monoid structure plays an important role in the definition of `mapSem'`. We can split changes on `DEnv (Seq a: env)` into ones on `Seq a` and ones on `env`, and process them separately by `transArg` and `transEnv`, respectively. Notice also that the pair `(h, htr)` is a correct monoid-respecting CTS incremental function, because they are obtained from `(f, tr)`, which is also a correct monoid-respecting CTS incremental function, by fixing `e` (and changes to them are also fixed to `deEmpty`).

D.2 Pseudo Functions

The definition of `mapSem'` gives us a hint of handling second-order construct whose semantic function treats an input open expression as "functions". Recall that `(h, htr)` is a (correct monoid-respecting) CTS incremental function, and `dh` can be thought as changes to the function (but affects only `h`). Unlike the original calculi [Cai et al. 2014; Giarrusso et al. 2019] where `htr` is integrated into `dh`, we couple it with the original function; such an integration makes the monoid structure dependent on `htr`, and the resulting function change does not respect the monoid structure.

The idea can be captured by the following datatype.

```
data PFun c a b = PFun (a → (b, c)) ((Delta a, c) → (Delta b, c))
instance Diff b ⇒ Diff (PFun c a b) where
  newtype Delta (PFun c a b) = DFun (c → (Delta b, c))
  PFun f tr /+ (DFun df ) = PFun f' tr
    where f' a = let (c, b) = f a; (c', db) = df c in (c', b /+ db)
```

(We omit the instance declaration of `Monoid (Delta b) ⇒ Monoid (PFun c a b)`.) With `PFun`, the definition of `mapSem'` becomes as Fig. 8, where the roles of `h`/`htr`/`dh` become more evident.

We can provide the semantic function for the application of `PFun`, but it is not much useful; recall that the goal of `PFun` is to extract common patterns in the treatment of input open expressions for "higher-order"-like functions such as `map`. It is worth noting that providing the HOAS representation (and its unembedded implementation) for `absPFunSem` is impossible, due to the timing of choosing `c`. In `absPFunSem`, choice of `c` can depends on `env`, and thus the result cannot be expressed as `exists c. CTS env (PFun c a b)`, where `c` is chosen without knowing `env`. Handling existential types in the DSL might help, but this means that the DSL is no longer simply-typed and thus requires non trivial extension of the unembedding that targets simply-typed system [Atkey 2009; Atkey et al.

```
mapSem' :: (Diff a, Diff b) ⇒ CTS (a ': env) b → CTS (Seq a ': env) (Seq b)
mapSem' (CTS f tr) = CTS f' tr'
  where
    f' (ECons (PackDiff xs) g) =
      let h x = f (ECons (PackDiff x) g)
          (ys, cs) = S.unzip (fmap h xs)
      in (ys, (cs, h))

    tr' (ECons (PackDiffDelta dxs) dg, (cs, h)) =
      let dgEmpty = mapEnv (\(PackDiffDelta _) → PackDiffDelta mempty) dg
          htr (dx, c) = tr (ECons (PackDiffDelta dx) dgEmpty, c)
          dh c        = tr (ECons (PackDiffDelta mempty) dg, c)
          (dys1, cs1) = transArg h htr (dxs, cs)
          (dys2, cs2) = transEnv dh cs1
          h' a = let (b, c) = h a; (dy, c') = dh c in (b /+ dy, c')
      in (dys1 <> dys2, (cs2, h'))

    transArg h htr (DSeq adxs, cs) = iterTr (transArgAtom h htr) (adxs, cs)

    transArgAtom h _htr (Ins i x, cs) =
      let (y, c) = h x
      in (DSeq $ pure $ Ins i y, S.insertAt i c cs)
    transArgAtom _h _htr (Del i, cs) =
      (DSeq $ pure $ Del i, S.delet’At i cs)
    transArgAtom _h htr (Rep i dx, cs) =
      let ci = S.index cs i
          (dy, ci') = htr (dx, ci)
      in (rep i dy, S.update i ci' cs)

    transEnv dh cs =
      let (dys, cs') = S.unzip (fmap dh cs)
      in (mconcat $ zipWith rep [0..] (toList dys), cs')
```

Fig. 7. An implementation of `mapSem'`, where `mapEnv :: (forall x. f x → g x) → Env f as → Env g as` is the mapping function for Env.

2009]. This issue can be addressed by using dynamic types to encapsulate c, which compromises type safety to some extent.

D.3 Programming Example: Incremental HaXML Filters

To demonstrate the programmability of our embedded incremental language, we implement query Q1 from Use Case “XMP” in XML Query Use Cases[16] by using an incremental version of HaXML filters [Wallace and Runciman 1999]. This query extracts books that are published by “Addison-Wesley” after 1991, keeping their titles and years, as an input and output example in Fig 9.

We use the following rose tree type to express XML fragments.

```
data Tree = Elem (Fixed String) (Seq Tree)
          | Attr (Fixed String) MyString | Text MyString
```

[16] https://www.w3.org/TR/xquery-use-cases/

```
mapSem' g = mapSem (absPFunSem g)

absPFunSem :: Diff a ⇒ CTS (a : env) b → exists c. CTS env (PFun c a b)
absPFunSem (CTS f tr) = CTS f' tr'
  where f' g = (PFun (\a → f (ECons (PackDiffDelta a) g ))
                             (\(da, c) → tr (ECons (PackDiffDelta da) dgEmpty, c))
               , ( ))
          where dgEmpty = mapEnv (\(PackDiffDelta _) → PackDiffDelta mempty) g
        tr' (dg,_) = (DFun (\c → tr (ECons (PackDiffDelta mempty) dg, c)), ( ))

mapSem'' :: Diff b ⇒ (forall c.CTS env (PFun c a b)) → CTS (Seq a : env) b
mapSem'' (CTS f tr) = CTS f' tr'
  where f' (ECons (PackDiffDelta as) g ) = let (ph@(PFun h ), c) = f g
                                                (as, cs) = Seq.unzip (Seq.map h bs)
                                            in (bs, (cs, c, ph))
        tr' (ECons (PackDiffDelta (DSeq adas)) dg, (cs, c, ph@(PFun h htr))) =
          let (DFun dh, c' ) = tr (dg, c)
              (dbs1,   cs' )   = iterTr (transArg h htr) (adss, cs)
              (dbs2,   cs'') = transEnv dh cs'
          in (dbs1 <> dbs2, (cs'', c', ph /+ DFun dh))
```

Fig. 8. Construction of PFun and an example of its use: we abused the Haskell syntax to use the bare exists.

```
type MyString = Seq Char
```

Here, `Fixed a` represents a values, which are not changed in Δ-propagation.

```
data ADTree = DChild (Delta Tree) | DTag String
            | DAttr String Delta MyString | DText Delta MyString
instance Diff Tree where
  newtype Delta Tree = DTree [ADTree]
  t /+ DTree ads = foldl applyADTree t ads
```

Here, `applyADTree :: Tree → ADTree → Tree` applies an atomic change to a given tree: `DChild dt` applies `dt` to the children `ts` of `Elem n ts`, `DTag n'` replaces the element name `n` of `Elem n ts` with `n'`, `DAttr k dv` replaces the attribute name to `k` and changes `v` by `dv` of `Attr k v`, and `DText ds` applies `ds` to `s` of `Text s`; they keep the original tree if it is not in a designated form. One might notice that `t /+ dt` cannot change the top-most constructor of *t*; to do so, we remove the node and insert a new one at the parent node of `t`. This simplifies implementations of some key methods such as `children :: exp Tree → exp (Seq Tree)`, where the change of the top-most construct would trigger the removal of all the elements from the output (and thus it needs to keep the length of the children as a cache if we would consider such changes).

We extend the language to include HaXML filters as Fig. 10, following the general recipe. Then, incremental versions of HaXML filters [Wallace and Runciman 1999] are represented by the following type.

```
type EFilter exp = exp Tree → exp (Seq Tree)
```

Also, the language comes with many derived filters and filter combinators; some important ones are excerpted in Fig. 11.

```
<bib>
  <book year="1994">
    <title>TCP/IP Illustrated</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
  <book year="1992">
    <title>Advanced Programming in the Unix environment</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
  <book year="2000">
    <title>Data on the Web</title>
    <author><last>Abiteboul</last><first>Serge</first></author>
    <author><last>Buneman</last><first>Peter</first></author>
    <author><last>Suciu</last><first>Dan</first></author>
    <publisher>Morgan Kaufmann Publishers</publisher>
    <price>39.95</price>
  </book>
  <book year="1999">
    <title>The Economics of Technology and Content for Digital TV</title>
    <editor>
      <last>Gerbarg</last><first>Darcy</first>
      <affiliation>CITI</affiliation>
    </editor>
    <publisher>Kluwer Academic Publishers</publisher>
    <price>129.95</price>
  </book>
</bib>
```

(a) Input

```
<bib>
  <book year="1994">
    <title>TCP/IP Illustrated</title>
  </book>
  <book year="1992">
    <title>Advanced Programming in the Unix environment</title>
  </book>
</bib>
```

(b) Corresponding Output

Fig. 9. Input and Output XMLs for Q1 (taken from https://www.w3.org/TR/xquery-use-cases/)

```
class CTSBase exp ⇒ CTSBool exp where
  not :: exp Bool → exp Bool
  if_ :: exp Bool → exp a → exp a → exp a

class CTSBool exp ⇒ CTSLiftPred exp where
  liftPred :: (a → Bool) → exp a → exp Bool

class (CTSSeq exp, CTSBool exp) ⇒ CTSSeqE exp where
  filter :: (exp a → exp Bool) → exp (Seq a) → exp (Seq a)
  append :: exp (Seq a) → exp (Seq a) → exp (Seq a)
  null :: exp (Seq a) → exp Bool

class (CTSSeqE exp, CTSBool e, . . . ) ⇒ CTSFilter exp where
  children :: exp Tree → exp (Seq Tree)
  makeElem :: String → exp (Seq Tree) → exp Tree
  hasTag :: String → exp Tree → exp Bool
  ...
  unsingle :: exp (Seq a) → exp a
```

Fig. 10. Extended Language for HaXML Filters

```
keep :: CTSFilter exp ⇒ EFilter exp
keep = singleton
none :: CTSFilter exp ⇒ EFilter exp
none = \_ → empty
tag :: CTSFilter exp ⇒ String → EFilter exp
tag s = filter (hasTag s) . keep
(>>) :: CTSFilter exp ⇒ EFilter exp → EFilter exp → EFilter exp
f >> g = concat . (map g) . f
f /> g = f >> children >> g
with :: CTSFilter exp ⇒ EFilter exp → EFilter exp → EFilter exp
f `with` g = filter (not . null . g) . f
```

Fig. 11. Derived Filters and Filter Combinators (Excerpt)

```
q1 :: (CTSFilter exp, CTSLiftPred exp) ⇒ exp Tree → exp Tree
q1 = unsingle . q1fitler

q1filter :: (CTSFilter exp, CTSLiftPred exp) ⇒ EFilter exp
q1filter = mkElem "bib" [keep /> ((tag "book" `with` byAW `with` after1991)
                        >> mkElem "book" [keep /> attr "year", keep /> tag "title"])]
  where
    byAW = keep /> tag "publisher" /> text "Addison-Wesley"
    after1991 = keep /> ifAttr "year"
      (\s a → if_ (liftPred (\str → (read (toString str) :: Int) > 1991) s)
      (keep a) (none a)) none
```

Fig. 12. Query Q1

Fig. 12 shows an implementation `q1` of the query Q1 in our embedded DSL. Mostly, the code are self-explanatory except some new derived methods: `attr` is a variant of `tag` for attributes, `mkElem a fs` constructs `Elem s ts` where `ts` are constructed by applying the filters `fs`, and `ifAttr s k f` applies `k v` to the input if an input is such an attribute named `s` of value `v`, and otherwise applies `f` to the input. Overall, `q1filter` is implemented quite concisely thanks to incremental filters in our EDSL. However, one can find that the use of `if_` breaks the abstraction using filters, indicated by the explicit argument `a`. Notice that `if_` requires `exp a` for then/else-clauses, and thus we cannot pass expressions of type `EFilter exp = exp Tree → exp Tree`. This is actually a cost to pay for not having higher-order functions in the DSL. For this particular case, the solution is simple: we prepare families of `if_` such as `if2 :: CTSBool exp ⇒ exp Bool → (exp a → exp r ) → (exp a → exp r ) → exp a → exp r`.[17]

In the above implementation, we intentionally separated the `liftPred` method into `CTSLiftPred`. The method is unfortunately not compatible with the optimized implementation discussed in §D.4 that uses staging to eliminate the cost caused unembedding. To avoid this incompatibility, we have to provide methods that replace `toString`, `read` (specialized to `Int`) and `(<)`, which requires substantial effort.

[17]We can provided a unified implementation using type families and GADTs, but cannot choose an appropriate one by using the type of the then/else-clause due to the treatment of the type variable exp in Haskell.

A subtlety in the code is `filter` used inside `with` (see Fig. 11). The associated semantic function with `filter` stores the whole input as a cache because it needs to produce appropriate insertions when a condition becomes true. Thus, the successive applications of `with` in `q1filter` is inefficient in the sense that the associated caches have overlaps. This suggests an optimization by code transformation, but it is not implemented by our current prototype.

D.4 More Efficient Implementation

The implementation based on CTS involves several places to improve in terms of performance: e.g.,

(1) Manipulation of value-level representations of typing environments that happens in runtime.
(2) The empty caches (of the unit type) stored in the whole cache.
(3) The function `transEnv` called even when dh is the identity change.

We address the first issue by using staging to perform the unembedding-related computation is the first stage and leave the essential computation in the second stage; specifically, we use Template Haskell [Sheard and Jones 2002]. The second issue is addressed by using join lists to manage the structure of caches so that the empty cache works as the identity element for the joining operation. In the combination of staging, we keep only the elements in the join lists in the second stage, but some semantic functions requires us to store the whole cache in a data structure. For this purpose, an optimized semantic domain also keeps the shape of the join list for inter-conversion between join lists in the first stage and ones in the second stage. For the third issue, we add a method `checkEmpty :: Delta a → Bool` to the typeclass `Diff`, which returns **`True`** only if the input change is an identical one. Also, we gather uses of variables so that the checking function becomes more accurate for `DFun`, which now has an extra Boolean field for `checkEmpty` that comes true when `checkEmpty` returns true for any element in de in the construction (i.e., in `absPFunSem`).

Combining these ideas, the semantic domain now changes to:

```
data CTSQ env a = forall cs us.
  CTSQ (TEnv env)
       (TConn cs)      -- cache shape
       (Env SBool us) -- uses
       (MCode (Env CodeDiff (Extr env us) → MCode (Code a, Conn Code cs),
               Env CodeDelta (Extr env us) → Conn Code cs
                  → MCode (CodeDelta a, Conn Code cs)))
```

Here, we write `Code a` for a type for code of type a,[18] `CodeM a` is the monad satisfying: `CodeM a ≃ (a → Code r) → Code a`. The monad is used to share the context in code generation, and avoid materializing whole structure; e.g., `CodeM (a, b)` does not immediately construct pairs in generated code. Also, `CodeD` and `CodeP` are variants of `Code` satisfying `CodeD a ≃ Code (Delta a)` and `CodeP a ≃ Code (PackDiff a)`. The type `Conn f cs` denotes a join list of which contents is `f ci` where `cs = [c1, ..., cn]`. We do not dive more details about this because the implementation is too complex for a paper.[19]

E RELATED WORK OF EMBEDDED INCREMENTAL LANGUAGE

eCTS vs CTS. Though the original CTS incremental λ-calculus [Giarrusso et al. 2019] is untyped, the accompanied implementation contains an simply-typed embedded DSL based on parametric HOAS [Chlipala 2008]. The EDSL is not suitable for users to directly program with; its (P)HOAS

[18] `Code a` is `Q (TExp a)` in GHC before 9, and is `Language.Haskell.TH.Code Q a` in GHC 9.

[19] See the source code (which uses a preliminary formalization of Embedding by Unembedding) available from https://archive.softwareheritage.org/swh:1:rev:fe7a2d44582d334d83d503d443843351af24d821 for the details.

representation follows the λ-lifted A'-normal form (a variant of the A-normal form [Flanagan et al. 1993]), and exposes the cache types. Also, the interpretation of the EDSL, code generation by strings, is not backed by the theoretical basis, unlike unembedding [Atkey 2009]. The definition of *mapSem* in Appendix D is adopted from their implementation. We note, however, that, to process changes one-by-one, their treatment of functions in the implementation becomes different from the one in the paper, and hence the correctness proof (and that in the accompanied Coq code) is not lifted to the EDSL.

Other Adaptations of CTS. Morihata [2020] gives an interface to Cai et al. [2014]'s incremental computation based on short-cut fusion [Gill et al. 1993] with reformulation. In his framework, a function to be incrementalized takes abstract polymorphic methods to be substituted by incremental functions. Thus, his use of the short-cut fusion is similar to the tagless-final style [Carette et al. 2009]. An issue of the original calculus, which the CTS version [Giarrusso et al. 2019] also addresses, is the recomputation of A required by a derivative $A \to \Delta A \to \Delta B$. Interestingly, tupling [Chin 1993] does not help here as ΔA is not available for the first run. Their idea can be viewed as Yoneda embedding to have $A \to \Delta A \to \Delta B \sim A \to \forall s.(s \to \Delta A) \to (s \to \Delta B)$. Then, one can tuple it with the original function to have $(A, s \to \Delta A) \to (B, s \to \Delta B)$, which can be viewed as an interpretation domain of $A \to B$ functions in the DSL. Now the issue of ΔA not being available is resolved by passing *id* to the tupled function yielding $A \to (B, \Delta A \to \Delta B)$. This approach scales to propagation of multiple updates: starting from $A \to \mathsf{Interact}\ (\Delta A)\ (\Delta B)$, one obtains $A \to (B, \mathsf{Interact}\ (\Delta A)\ (\Delta B))$. This approach is simple and type preserving, in contrast to Giarrusso et al. [2019] which targets untyped system and performs a global program transformation that requires λ-lifting and A'-normalizing as a preprocess. However, it is unclear how this can apply to higher-order methods such as *map*.

CTS Alternatives. We chose the CTS incremental calculus [Giarrusso et al. 2019] as a target because it is simple, easy to reason about, and realistic. There are many other frameworks for incremental computation. One of the most popular ones is self-adjusting computation [Acar et al. 2006], where programmers create, read and write *modifiables* so that a dependency graph among modifiables can be extracted to incrementally propagate changes of modifiables from input to output. We could extract the core part to make a DSL to embed by unembeddeding, whose API would be similar to the ML library presented by Acar et al. [2006]. However, we suspect that the resulting EDSL would have limited benefits as the manipulation of the modifiables is the main challenge and not made easy by the embedding. In later work [Chen et al. 2012, 2014b], systems are designed to generate modifiable-manipulate code from ordinary functional programs with annotations.

F MORE DETAILS OF EMBEDDED BIDIRECTIONAL LANGUAGE

F.1 Round-tripping Laws

A lens must satisfy round-tripping to be considered as *well-behaved* [Foster et al. 2007]. Here we use the system adopted by HOBiT, which is a slightly weaker version of the original round-tripping [Bancilhon and Spyratos 1981; Foster et al. 2007; Hegner 1990].

Definition F.1 ($\preceq$-well-behavedness [Matsuda and Wang 2018d]). Let $(S, \preceq)$ and $(V, \preceq)$ be partially-ordered sets. A lens $\ell :: \mathsf{Lens}\ S\ V$ is called $\preceq$-*well-behaved* if it satisfies

$$get\ \ell\ s = v \implies v \text{ is maximal} \wedge (\forall v'.\ v' \preceq v \implies put\ \ell\ s\ v' \preceq s) \qquad (\preceq\textbf{-Acceptability})$$
$$put\ \ell\ s\ v = s' \implies (\forall s''.\ s' \preceq s'' \implies v \preceq get\ \ell\ s'') \qquad (\preceq\textbf{-Consistency})$$

for any $s, s' \in S$ and $v \in V$, where s is maximal. □

Intuitively, $s_1 \preceq s_2$ means that s_2 is more determined than s_1. The laws coincide with the conventional well-behavedness when $(\preceq) = (=)$ on S and V. For the sake of this paper, we do not go into many details on round-tripping, but emphasize that the property is in fact preserved in our handling by using a type synonym Lens S V for $\preceq$-well-behaved lenses.

We use the weaker $\preceq$-well-behavedness so that the embedded language can have a standard simply-typed system that allows both weakening and contraction. Alternatively, we can keep the conventional well-behavedness if we consider a linear-typed language. Though the original unembedding proposal [Atkey 2009; Atkey et al. 2009] relies on the fact that a language to embed can be represented as a second-order algebra [Fiore and Hur 2010], we conjecture that a similar technique is also possible for linear cases. Recently, Haskell included support for linear types [Bernardy et al. 2018], and thus the unembedding of a linear language leveraging the parametricity of a linear-typed host language is an interesting future direction.

F.2 Pattern Combinators

Directly using *unpair* and *branch*$_n$ in programming is cumbersome, and it is desirable if we can write bidirectional branching as ordinary **case** expressions instead. To address this issue, we provide pattern-like combinators on top of the EDSL so far constructed.

We first confirm that patterns p can be viewed as linear-typed expressions. A convenient way to embed linear-typed terms is to use difference lists to represent typing environments (e.g., [Polakow 2015]) as $\Gamma_{\mathrm{in}} \vdash p : A \mid \Gamma_{\mathrm{out}}$, which reads that, by matching a pattern of type p, typing environment Γ_{in} changes to Γ_{out}. Then, a semantics of a pattern can be given as a pair of functions as below.

$$[\![\Gamma_{\mathrm{in}} \vdash - : A \mid \Gamma_{\mathrm{out}}]\!] = \mathsf{PBij}\ ([\![A]\!], [\![\Gamma_{\mathrm{in}}]\!]_{\mathrm{L}})\ [\![\Gamma_{\mathrm{out}}]\!]_{\mathrm{L}}$$

The semantics of typing environment $[\![\Gamma]\!]_{\mathrm{L}}$ is different from the one used for a term-in-context (VEnv *env*), as there is no need of weakening and contraction, and using $[\![x_1 : A_1, \dots, x_n : A_n]\!] = ((\dots((), [\![A_1]\!]), \dots), [\![A_n]\!])$ is fine.

So, the datatype for shallow-embedded patterns can be given as

```
data Pat i o a = Pat (TEnv i → TEnv o) (PBij (a, LVEnv i) (LVEnv o))
```

where `LVEnv o` implements $[\![\Gamma]\!]_{\mathrm{L}}$; specifically, `LVEnv = Env Identity`. The second parameter of `Pat` computes Γ_{out} form Γ_{in}, which will be used to decompose $[\![\Gamma_{\mathrm{out}}]\!]_{\mathrm{L}}$ in the embedded DSL (otherwise, it is unclear how `unpair` can be applied). The following are basic combinators for patterns.

```
varP  :: Pat i (i : a) a
unitP :: Pat i i ( )
pairP :: Pat m o a → Pat i m b → Pat i o (a, b)
liftP :: PBij a b → Pat i o b → Pat i o a
```

They are powerful enough to define many patterns; e.g., the nil and cons patterns can be defined via partial bijections `unnil :: PBij [a] ()` and `uncons :: PBij [a] (a, [a])`—which are the inverses of the partial bijections `nil :: PBij ( ) [a]` and `cons :: PBij (a, [a]) [a]` corresponding to `[]` and `(:)`, respectively—as:

```
nilP :: Pat i i [a]
nilP = liftP unnil unitP
consP :: Pat m o a → Pat i m [a] → Pat i o [a]
consP p1 p2 = liftP uncons (pairP p1 p2 )
```

An important property of Pat is that Pat [] o a defines a partial bijection between a and LVEnv o. Also, we can obtain TEnv o as mentioned before, which is used to define the following decomposition function.

```
decompEnv :: HOBiT exp ⇒ TEnv o → (Env exp o → exp r )
                               → (exp (LVEnv o) → exp r )
decompEnv ENil       k e = ununit (prim n2uL e) (k Nil)
decompEnv (ECons g ) k e = unpair (prim c2pL e)
                                (\x xs → decompEnv g (k . ECons x) xs)
```

Here, n2uL :: Lens (LVEnv [ ]) ( ) and c2pL :: Lens (LVEnv (a : as)) (a, LVEnv as) are the lenses defined by the obvious isomorphism between types.

Thanks to these patterns we can now have the following *derived* method.

```
case_ :: HOBiT exp ⇒ String → exp a → [Br exp a r ] → exp r
```

Here, Br is a type for branches defined by:

```
data Br exp a r where
  Br :: Pat [ ] o a → (Env exp o → exp r )
     → (r → Bool) → (a → r → a) → Br exp a r
```

Note that o is existentially quantified in the definition. The implementation of case_ involves branch_n and decompEnv, with construction of a lens to convert an input a to sums LVEnv o1 + ... + LVEnv on where o_i is the existentially qualified pattern for the ith branch. The ease of defining derived combinators demonstrate the clear benefit of embedding, where we can keep the core set of combinators (i.e., semantic functions) small and consequently suitable for complex reasoning. This advantage is decisive in bidirectional languages, as the need to reason about round-tripping is no longer at the cost of reduced language features.

We can even add extra features to the language by Haskell's type-level programming. Consider the following type family

```
type family Func exp as r where
  Func exp [ ]      r = r
  Func exp (a : as) r = exp a → Func exp a r
```

and a converter between the representations (whose straightforward definition is omitted):

```
fromFunc :: Func exp as (exp r ) → (Env exp as → exp r )
```

With these, we can define a combinator with the following type.

```
(⟶) :: Pat [ ] o a → (Func exp o (exp r ), r → Bool, a → r → a)
      → Br exp a r
p@(Pat _ _) ⟶ (b,f, recon) = Br p (fromFunc b) f recon
```

As a result, users of the DSL can be oblivious to the implementation and need not touch Env at all.

REFERENCES

Samson Abramsky. 2005. A structural approach to reversible computation. *Theor. Comput. Sci.* 347, 3 (2005), 441–464. https://doi.org/10.1016/j.tcs.2005.07.002

Umut A. Acar, Guy E. Blelloch, and Robert Harper. 2006. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.* 28, 6 (2006), 990–1034. https://doi.org/10.1145/1186632.1186634

Umut A. Acar and Yan Chen. 2013. Streaming Big Data with Self-Adjusting Computation. In *Proceedings of the 2013 Workshop on Data Driven Functional Programming* (Rome, Italy) *(DDFP '13)*. Association for Computing Machinery, New York, NY, USA, 15–18. https://doi.org/10.1145/2429376.2429382

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, Vol. 5608)*, Pierre-Louis Curien (Ed.). Springer, 35–49. https://doi.org/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*, Stephanie Weirich (Ed.). ACM, 37–48. https://doi.org/10.1145/1596638.1596644

François Bancilhon and Nicolas Spyratos. 1981. Update Semantics of Relational Views. *ACM Trans. Database Syst.* 6, 4 (1981), 557–575. https://doi.org/10.1145/319628.319634

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. *Proc. ACM Program. Lang.* 2, POPL (2018), 5:1–5:29. https://doi.org/10.1145/3158093

Eli Bingham, Fritz Obermeyer, Yerdos Ordabayev, and Du Phan. 2021. Tensor Partial Evaluation. HOPE 2021: The 8th ACM SIGPLAN Workshop on Higher-Order Programming with Effects. Extend Abstract Avaiable via: https://icfp21.sigplan.org/details/hope-2021-papers/4/Tensor-Partial-Evaluation.

Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. 2014. A theory of changes for higher-order languages: incrementalizing λ-calculi by static differentiation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 145–155. https://doi.org/10.1145/2594291.2594304

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.* 19, 5 (2009), 509–543. https://doi.org/10.1017/S0956796809007205

Chao-Hong Chen and Amr Sabry. 2021. A computational interpretation of compact closed categories: reversible programming with negative and fractional types. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. https://doi.org/10.1145/3434290

Yan Chen, Jana Dunfield, and Umut A. Acar. 2012. Type-directed automatic incrementalization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 299–310. https://doi.org/10.1145/2254064.2254100

Yan Chen, Jana Dunfield, Matthew A. Hammer, and Umut A. Acar. 2014a. Implicit self-adjusting computation for purely functional programs. *Journal of Functional Programming* 24, 1 (2014), 56–112. https://doi.org/10.1017/S0956796814000033

Yan Chen, Jana Dunfield, Matthew A. Hammer, and Umut A. Acar. 2014b. Implicit self-adjusting computation for purely functional programs. *J. Funct. Program.* 24, 1 (2014), 56–112. https://doi.org/10.1017/S0956796814000033

Wei-Ngan Chin. 1993. Towards an Automated Tupling Strategy. In *Proceedings of the 1993 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (Copenhagen, Denmark) *(PEPM '93)*. Association for Computing Machinery, New York, NY, USA, 119–132. https://doi.org/10.1145/154630.154643

Adam Chlipala. 2008. Parametric higher-order abstract syntax for mechanized semantics. In *ICFP*, James Hook and Peter Thiemann (Eds.). ACM, 143–156. https://doi.org/10.1145/1411204.1411226

Alonzo Church. 1940. A Formulation of the Simple Theory of Types. *J. Symb. Log.* 5, 2 (1940), 56–68. https://doi.org/10.2307/2266170

Thomas Ehrhard and Laurent Regnier. 2003. The differential lambda-calculus. *Theor. Comput. Sci.* 309, 1-3 (2003), 1–41. https://doi.org/10.1016/S0304-3975(03)00392-X

Conal Elliott. 2017. Compiling to Categories. *Proc. ACM Program. Lang.* 1, ICFP, Article 27 (aug 2017), 27 pages. https://doi.org/10.1145/3110271

Leonidas Fegaras. 2010. Propagating updates through XML views using lineage tracing. In *ICDE*, Feifei Li, Mirella M. Moro, Shahram Ghandeharizadeh, Jayant R. Haritsa, Gerhard Weikum, Michael J. Carey, Fabio Casati, Edward Y. Chang, Ioana Manolescu, Sharad Mehrotra, Umeshwar Dayal, and Vassilis J. Tsotras (Eds.). IEEE, 309–320. https://doi.org/10.1109/ICDE.2010.5447896

Leonidas Fegaras and Tim Sheard. 1996. Revisiting Catamorphisms over Datatypes with Embedded Functions (or, Programs from Outer Space). In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, Hans-Juergen Boehm and Guy L. Steele Jr. (Eds.). ACM Press, 284–294. https://doi.org/10.1145/237721.237792

Francisco Ferreira and Brigitte Pientka. 2017. Programs Using Syntax with First-Class Binders. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 504–529. https://doi.org/10.1007/978-3-662-54434-1_19

Marcelo P. Fiore and Chung-Kil Hur. 2010. Second-Order Equational Logic (Extended Abstract). In *Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6247)*, Anuj Dawar and Helmut Veith (Eds.). Springer, 320–335. https://doi.org/10.1007/978-3-642-15205-4_26

Marcelo P. Fiore, Gordon D. Plotkin, and Daniele Turi. 1999. Abstract Syntax and Variable Binding. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*. IEEE Computer Society, 193–202. https://doi.org/10.1109/LICS.1999.782615

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, Robert Cartwright (Ed.). ACM, 237–247. https://doi.org/10.1145/155090.155113

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2005. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, Jens Palsberg and Martín Abadi (Eds.). ACM, 233–246. https://doi.org/10.1145/1040305.1040325

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.* 29, 3 (2007). https://doi.org/10.1145/1232420.1232424

Paolo G. Giarrusso, Yann Régis-Gianas, and Philipp Schuster. 2019. Incremental λ-Calculus in Cache-Transfer Style - Static Memoization by Program Transformation. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11423)*, Luís Caires (Ed.). Springer, 553–580. https://doi.org/10.1007/978-3-030-17184-1_20

Jeremy Gibbons and Nicolas Wu. 2014. Folding Domain-Specific Languages: Deep and Shallow Embeddings (Functional Pearl). *SIGPLAN Not.* 49, 9 (aug 2014), 339–347. https://doi.org/10.1145/2692915.2628138

Andrew Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A Short Cut to Deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture, FPCA 1993, Copenhagen, Denmark, June 9-11, 1993*, John Williams (Ed.). ACM, 223–232. https://doi.org/10.1145/165180.165214

Makoto Hamana. 2018. Polymorphic Rewrite Rules: Confluence, Type Inference, and Instance Validation. In *Functional and Logic Programming - 14th International Symposium, FLOPS 2018, Nagoya, Japan, May 9-11, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10818)*, John P. Gallagher and Martin Sulzmann (Eds.). Springer, 99–115. https://doi.org/10.1007/978-3-319-90686-7_7

Makoto Hamana. 2019. How to prove decidability of equational theories with second-order computation analyser SOL. *J. Funct. Program.* 29 (2019), e20. https://doi.org/10.1017/S0956796819000157

Matthew A. Hammer, Umut A. Acar, and Yan Chen. 2009. CEAL: A C-Based Language for Self-Adjusting Computation. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 25–37. https://doi.org/10.1145/1542476.1542480

Stephen J. Hegner. 1990. Foundations of Canonical Update Support for Closed Database Views. In *ICDT (Lecture Notes in Computer Science, Vol. 470)*, Serge Abiteboul and Paris C. Kanellakis (Eds.). Springer, 422–436. https://doi.org/10.1007/3-540-53507-1_93

Stephen J. Hegner. 2004. An Order-Based Theory of Updates for Closed Database Views. *Ann. Math. Artif. Intell.* 40, 1-2 (2004), 63–125.

Gérard P. Huet and Bernard Lang. 1978. Proving and Applying Program Transformations Expressed with Second-Order Patterns. *Acta Inf.* 11 (1978), 31–55. https://doi.org/10.1007/BF00264598

Mathieu Huot, Sam Staton, and Matthijs Vákár. 2020. Correctness of Automatic Differentiation via Diffeologies and Categorical Gluing. In *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12077)*, Jean Goubault-Larrecq and Barbara König (Eds.). Springer, 319–338. https://doi.org/10.1007/978-3-030-45231-5_17

Oleg Kiselyov. 2019. Effects Without Monads: Non-determinism – Back to the Meta Language. *Electronic Proceedings in Theoretical Computer Science* 294 (May 2019), 15–40. https://doi.org/10.4204/eptcs.294.2

Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. 2004. Strongly Typed Heterogeneous Collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell* (Snowbird, Utah, USA) *(Haskell '04)*. Association for Computing Machinery, New York, NY, USA, 96–107. https://doi.org/10.1145/1017472.1017488

Faustyna Krawiec, Simon Peyton Jones, Neel Krishnaswami, Tom Ellis, Richard A. Eisenberg, and Andrew W. Fitzgibbon. 2022. Provably correct, asymptotically efficient, higher-order reverse-mode automatic differentiation. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–30. https://doi.org/10.1145/3498710

Ben Lippmeier (Ed.). 2015. *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*. ACM. https://doi.org/10.1145/2804302

Kazutaka Matsuda and Meng Wang. 2013. FliPpr: A Prettier Invertible Printing System. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science,* *Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 101–120. https://doi.org/10.1007/978-3-642-37036-6_6

Kazutaka Matsuda and Meng Wang. 2015. Applicative bidirectional programming with lenses. In *ICFP*, Kathleen Fisher and John H. Reppy (Eds.). ACM, 62–74. https://doi.org/10.1145/2784731.2784750

Kazutaka Matsuda and Meng Wang. 2018a. Applicative bidirectional programming: Mixing lenses and semantic bidirectionalization. *J. Funct. Program.* 28 (2018), e15. https://doi.org/10.1017/S0956796818000096

Kazutaka Matsuda and Meng Wang. 2018b. Embedding invertible languages with binders: a case of the FliPpr language. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018*, Nicolas Wu (Ed.). ACM, 158–171. https://doi.org/10.1145/3242744.3242758

Kazutaka Matsuda and Meng Wang. 2018c. FliPpr: A System for Deriving Parsers from Pretty-Printers. *New Gener. Comput.* 36, 3 (2018), 173–202. https://doi.org/10.1007/s00354-018-0033-7

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

Kazutaka Matsuda and Meng Wang. 2020. Sparcl: a language for partially-invertible computation. *Proc. ACM Program. Lang.* 4, ICFP (2020), 118:1–118:31. https://doi.org/10.1145/3409000

Damiano Mazza and Michele Pagani. 2021. Automatic differentiation in PCF. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–27. https://doi.org/10.1145/3434309

Trevor L. McDonell, Manuel M. T. Chakravarty, Gabriele Keller, and Ben Lippmeier. 2013. Optimising purely functional GPU programs. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 49–60. https://doi.org/10.1145/2500365.2500595

Trevor L. McDonell, Joshua D. Meredith, and Gabriele Keller. 2021. Embedded Pattern Matching. *CoRR* abs/2108.13114 (2021). arXiv:2108.13114 https://arxiv.org/abs/2108.13114

Dale Miller and Gopalan Nadathur. 1987. A Logic Programming Approach to Manipulating Formulas and Programs. In *Proceedings of the 1987 Symposium on Logic Programming, San Francisco, California, USA, August 31 - September 4, 1987.* IEEE-CS, 379–388.

Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2017. Synthesizing Bijective Lenses. *Proc. ACM Program. Lang.* 2, POPL, Article 1 (dec 2017), 30 pages. https://doi.org/10.1145/3158089

Eugenio Moggi. 1998. Functor Categories and Two-Level Languages. In *FoSSaCS (Lecture Notes in Computer Science, Vol. 1378)*, Maurice Nivat (Ed.). Springer, 211–225. https://doi.org/10.1007/BFb0053552

Akimasa Morihata. 2020. Short Cut to Incremental Typed Functional Programs. Informal Proceedings of WPTE 2020: 7th International Workshop on Rewriting Techniques for Program Transformations and Evaluation. Available from http://maude.ucm.es/wpte20/papers/WPTE_2020_morihata.pdf.

Minh Nguyen, Roly Perera, Meng Wang, and Steven Ramsay. 2023. Effect handlers for programmable inference. (2023). https://doi.org/10.1145/3609026.3609729

Fritz Obermeyer, Eli Bingham, Martin Jankowiak, Du Phan, and Jonathan P. Chen. 2020. Functional Tensors for Probabilistic Programming. arXiv:1910.10775 [stat.ML]

Hugo Pacheco and Alcino Cunha. 2010. Generic Point-free Lenses. In *Mathematics of Program Construction*, Claude Bolduc, Jules Desharnais, and Béchir Ktari (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 331–352.

Ivan Perez and Henrik Nilsson. 2015. Bridging the GUI gap with reactive values and relations, See [Lippmeier 2015], 47–58. https://doi.org/10.1145/2804302.2804316

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*, Richard L. Wexelblat (Ed.). ACM, 199–208. https://doi.org/10.1145/53990.54010

Brigitte Pientka. 2008. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 371–382. https://doi.org/10.1145/1328438.1328483

Jeff Polakow. 2015. Embedding a full linear Lambda calculus in Haskell, See [Lippmeier 2015], 177–188. https://doi.org/10.1145/2804302.2804309

Raghu Rajkumar, Nate Foster, Sam Lindley, and James Cheney. 2013. Lenses for Web Data. *ECEASST* 57 (2013). https://doi.org/10.14279/tuj.eceasst.57.879

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Information Processing*, R.E.A. Mason (Ed.). Elsevier Science Publishers B.V. (North-Holland), 513–523.

Ajeet Shankar and Rastislav Bodík. 2007. DITTO: Automatic Incrementalization of Data Structure Invariant Checks (in Java). *SIGPLAN Not.* 42, 6 (jun 2007), 310–319. https://doi.org/10.1145/1273442.1250770

Tim Sheard and Simon L. Peyton Jones. 2002. Template meta-programming for Haskell. *ACM SIGPLAN Notices* 37, 12 (2002), 60–75. https://doi.org/10.1145/636517.636528

Benjamin Sherman, Jesse Michel, and Michael Carbin. 2021. λ_S: computable semantics for differentiable programming with higher-order functions and datatypes. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–31. https://doi.org/10.1145/3434284

Perdita Stevens. 2008. A Landscape of Bidirectional Model Transformations. In *GTTSE (Lecture Notes in Computer Science, Vol. 5235)*, Ralf Lämmel, Joost Visser, and João Saraiva (Eds.). Springer, 408–424. https://doi.org/10.1007/978-3-540-88643-3_10

Don Stewart. 2010. `yices-painless`: An embedded language for programming the Yices SMT solver. In Hackage: https://hackage.haskell.org/package/yices-painless. Visited 2022-06-27.

Van-Dang Tran, Hiroyuki Kato, and Zhenjiang Hu. 2020. Programmable View Update Strategies on Relations. *Proc. VLDB Endow.* 13, 5 (2020), 726–739. https://doi.org/10.14778/3377369.3377380

Janis Voigtländer. 2009. Bidirectionalization for free! (Pearl). In *POPL*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 165–176. https://doi.org/10.1145/1480881.1480904

Dimitrios Vytiniotis, Dan Belov, Richard Wei, Gordon Plotkin, and Martin Abadi. 2019. The Differentiable Curry. Program Transformations for Machine Learning@NeurIPS. Availble from: https://openreview.net/pdf?id=ryxuz9SzDB.

Philip Wadler. 1989. Theorems for Free!. In *FPCA*. 347–359.

Malcolm Wallace and Colin Runciman. 1999. Haskell and XML: Generic Combinators or Type-Based Translation?. In *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27-29, 1999*, Didier Rémy and Peter Lee (Eds.). ACM, 148–159. https://doi.org/10.1145/317636.317794

Fei Wang, Daniel Zheng, James M. Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. 2019. Demystifying differentiable programming: shift/reset the penultimate backpropagator. *Proc. ACM Program. Lang.* 3, ICFP (2019), 96:1–96:31. https://doi.org/10.1145/3341700

Meng Wang, Jeremy Gibbons, Kazutaka Matsuda, and Zhenjiang Hu. 2010. Gradual Refinement: Blending Pattern Matching with Data Abstraction. In *MPC (Lecture Notes in Computer Science, Vol. 6120)*, Claude Bolduc, Jules Desharnais, and Béchir Ktari (Eds.). Springer, 397–425.

Meng Wang, Jeremy Gibbons, Kazutaka Matsuda, and Zhenjiang Hu. 2013. Refactoring pattern matching. *Sci. Comput. Program.* 78, 11 (2013), 2216–2242. https://doi.org/10.1016/j.scico.2012.07.014

Michael Flænø Werk, Joakim Ahnfelt-Rønne, and Ken Friis Larsen. 2012. An embedded DSL for stochastic processes: research article. In *Proceedings of the 1st ACM SIGPLAN workshop on Functional high-performance computing, Copenhagen, Denmark, FHPC@ICFP 2012, September 9-15, 2012*, Andrzej Filinski and Clemens Grelck (Eds.). ACM, 93–102. https://doi.org/10.1145/2364474.2364488

Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi, and Hong Mei. 2007. Towards automatic model synchronization from model transformations. In *ASE*, R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer (Eds.). ACM, 164–173. https://doi.org/10.1145/1321631.1321657

Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. 2011. Towards a Reversible Functional Language. In *RC (Lecture Notes in Computer Science, Vol. 7165)*, Alexis De Vos and Robert Wille (Eds.). Springer, 14–29. https://doi.org/10.1007/978-3-642-29517-1_2

Yijun Yu, Yu Lin, Zhenjiang Hu, Soichiro Hidaka, Hiroyuki Kato, and Lionel Montrieux. 2012. Maintaining invariant traceability through bidirectional transformations. In *ICSE*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE, 540–550. https://doi.org/10.1109/ICSE.2012.6227162

Received 2023-03-01; accepted 2023-06-27