Compositional Embeddings of Domain-Specific Languages

YAOZHU SUN, The University of Hong Kong, China
UTKARSH DHANDHANIA, The University of Hong Kong, China
BRUNO C. D. S. OLIVEIRA, The University of Hong Kong, China

A common approach to defining domain-specific languages (DSLs) is via a direct embedding into a host language. There are several well-known techniques to do such embeddings, including *shallow* and *deep* embeddings. However, such embeddings come with various trade-offs in existing programming languages. Owing to such trade-offs, many embedded DSLs end up using a mix of approaches in practice, requiring a substantial amount of code, as well as some advanced coding techniques.

In this paper, we show that the recently proposed *Compositional Programming* paradigm and the CP language provide improved support for embedded DSLs. In CP we obtain a new form of embedding, which we call a *compositional embedding*, that has most of the advantages of both shallow and deep embeddings. On the one hand, compositional embeddings enable various forms of linguistic reuse that are characteristic of shallow embeddings, including the ability to reuse host-language optimizations in the DSL and add new DSL constructs easily. On the other hand, similarly to deep embeddings, compositional embeddings support definitions by pattern matching or dynamic dispatching (including dependent interpretations, transformations, and optimizations) over the abstract syntax of the DSL and have the ability to add new interpretations. We illustrate an instance of compositional embeddings with a DSL for document authoring called ExT. The DSL is highly flexible and extensible, allowing users to create various non-trivial extensions easily. For instance, ExT supports various extensions that enable the production of wiki-like documents, LATEX documents, vector graphics or charts. The viability of compositional embeddings for ExT is evaluated with three applications.

CCS Concepts: • **Software and its engineering** → **Abstraction, modeling and modularity**; **Domain specific languages**.

Additional Key Words and Phrases: Compositional Programming, Extensible Typesetting

1 INTRODUCTION

A common approach to defining domain-specific languages (DSLs) is via a direct embedding into a host language. This approach is used in several programming languages, such as Haskell, Scala, and Racket. In those languages, various DSLs – including pretty printers [Hughes 1995; Wadler 2003], parser combinators [Leijen and Meijer 2001], and property-based testing frameworks [Claessen and Hughes 2000] – are defined as embedded DSLs. There are a few techniques for such embeddings, including the well-known *shallow* and *deep* embeddings [Boulton et al. 1992].

Authors' addresses: Yaozhu Sun, Department of Computer Science, The University of Hong Kong, China, yzsun@cs.hku.hk; Utkarsh Dhandhania, Department of Computer Science, The University of Hong Kong, China, ud99@connect.hku.hk; Bruno C. d. S. Oliveira, Department of Computer Science, The University of Hong Kong, China, bruno@cs.hku.hk.

Unfortunately, shallow and deep embeddings come with various trade-offs in existing programming languages. Such trade-offs have been widely discussed in the literature [Gibbons and Wu 2014; Rompf et al. 2012; Scherr and Chiba 2014]. On the one hand, the strengths of shallow embeddings are in providing *linguistic reuse* [Krishnamurthi 2001], exploiting meta-language optimizations, and allowing the addition of new DSL constructs easily. On the other hand, deep embeddings shine in enabling the definition of complex semantic interpretations and optimizations over the abstract syntax tree (AST) of the DSL, and they enable adding new semantic interpretations easily. Regarding such trade-offs, Svenningsson and Axelsson [2015] made the following striking comment:

> *"The holy grail of embedded language implementation is to be able to combine the advantages of shallow and deep in a single implementation."*

While progress has been made in embedded language implementation, the holy grail is still not fully achieved in existing programming languages. Owing to the trade-offs between shallow and deep embeddings, many realistic embedded DSLs end up using a mix of both approaches in practice or more advanced forms of embeddings. For instance, there have been several approaches [Jovanović et al. 2014; Rompf et al. 2012; Svenningsson and Axelsson 2015] promoting the use of shallow embeddings as the frontend of the DSL to enable linguistic reuse, while deep embeddings are used as the backend for added flexibility in defining semantic interpretations. While such approaches manage to alleviate some of the trade-offs, they require translations between the two embeddings, a substantial amount of code, and some advanced coding techniques. Alternatively, there are more advanced embedding techniques, which are inspired by work on extensible Church encodings of algebraic data types [Hinze 2006; Oliveira 2009; Oliveira et al. 2006]. Such techniques include *tagless-final embeddings* [Carette et al. 2009; Kiselyov 2010], *polymorphic embeddings* [Hofer et al. 2008], and *object algebras* [Oliveira and Cook 2012], and they are able to eliminate some of the trade-offs too. In particular, those approaches eliminate the trade-offs with respect to extensibility, facilitating both the addition of new DSL constructs and semantic interpretations. However, being quite close to shallow embeddings, those approaches lack some important capabilities, such as the ability to define complex interpretations and the use of (nested) pattern matching to express semantic interpretations and transformations easily and modularly.

In this paper, we show that the recently proposed *Compositional Programming* paradigm [Zhang et al. 2021] and the CP language provide improved programming language support for embedded DSLs. Compositional Programming provides an alternative way to define data structures compared to algebraic data types in functional programming or class hierarchies in object-oriented programming. Compositional interfaces and traits play a role similar to algebraic data types and definitions by pattern matching in functional programs. However, unlike algebraic data types and definitions by pattern matching, compositional interfaces and traits are modularly extensible. *Intersection types* [Oliveira et al. 2016] enable the composition of interfaces, while *nested composition* [Bi et al. 2018] enables the composition of traits with pattern matching definitions. Thus, the CP language does not suffer from the infamous *Expression Problem* [Wadler 1998]. In addition, CP comes with several mechanisms to express *modular dependencies*, allowing powerful forms of dependency injection and complex semantic interpretations.

With those programming language features, we obtain a new form of embedding called a *compositional embedding*, with nearly all of the advantages of both shallow and deep embeddings. On the one hand, compositional embeddings enable various forms of linguistic reuse that are characteristic of shallow embeddings, including the ability to reuse host-language optimizations in the DSL and easily add new DSL constructs. On the other hand, similarly to deep embeddings, compositional

embeddings support definitions by pattern matching or dynamic dispatching, including optimizations and transformations over the abstract syntax of the DSL, as well as the ability to add new interpretations. In short, we believe that compositional embeddings come very close to the holy grail of embedded language implementation desired by Svenningsson and Axelsson [2015].

We reimplemented the CP language and made it available as an in-browser interpreter. Using this new implementation of CP, we illustrate an instance of compositional embeddings with a DSL for document authoring called E$_X$T (**Ex**tensible **T**ypesetting). The DSL is highly flexible and extensible, allowing users to create various non-trivial extensions. For instance, E$_X$T supports extensions that enable the production of wiki-like documents, LATEX documents, SVG charts, or computational graphics like fractals. Our largest application is Minipedia: a mini version of Wikipedia. Minipedia includes wiki pages for several states and cities and some tables that list countries by population or area. The E$_X$T DSL, as well as its various extensions and applications, is available online.

In summary, the contributions of this paper are:

- **Compositional embeddings.** We show that the previously proposed Compositional Programming [Zhang et al. 2021], together with the CP language, enables a new form of embedding, which we call a compositional embedding. This paper reveals that compositional embeddings have most of the advantages of shallow and deep embeddings.
- **Comparison of embedding techniques.** We present a detailed comparison between compositional embeddings and various techniques used in embedded language implementations, including shallow, deep, hybrid, and polymorphic embeddings.
- **The E$_X$T DSL.** We present a DSL for document authoring called E$_X$T as a realistic instance of compositional embeddings. E$_X$T is extremely flexible and customizable by users, with many features implemented in a modular way.
- **Implementation and applications of E$_X$T.** We have built several applications with E$_X$T, three of which are discussed in more detail in this paper. The largest application is Minipedia, and the other two applications illustrate computational graphics like fractals and a document extension for charts. The implementation of E$_X$T and its applications are available at:

<div align="center">

https://plground.org

</div>

2 EMBEDDINGS OF DSLS

In this section, we give some background on existing approaches to embedded DSLs and evaluate their strengths and drawbacks. This will be useful for the comparison in Section 3. We focus on *shallow* and *deep* embeddings [Boulton et al. 1992], which are the two main alternative forms of embeddings. We also discuss some other forms of embeddings near the end of this section. To illustrate all these embeddings, we present programs in Haskell, which is well known for its good support for embedded DSLs and has a syntax close to the CP language.

2.1 A Simple Region DSL

Inspired by Hudak [1998] and Hofer et al. [2008], we consider a simple region DSL for plane geometry. To illustrate the challenges in developing such a DSL, we consider five separate steps in the development, which illustrate various desired features, such as linguistic reuse and the ease of adding features for software evolution.

(1) **Initial system.** We start with a small region language with five constructs: `circle` for creating a circular region with a given radius, `outside` for a complement to a certain region, two set operators `union` and `intersect`, and finally `translate` for moving a region by a given vector. The initial interpretation is to simply calculate the syntactic size of a region.

```haskell
data Vector = Vector { x :: Double, y :: Double } deriving Show
```

```haskell
type Region = Int


circle    :: Double → Region
circle    _ = 1
outside   :: Region → Region
outside   a = a + 1
union     :: Region → Region → Region
union     a b = a + b + 1
intersect :: Region → Region → Region
intersect a b = a + b + 1
translate :: Vector → Region → Region
translate _ a = a + 1
```

```haskell
data Region where
  Circle    :: Double → Region
  Outside   :: Region → Region
  Union     :: Region → Region → Region
  Intersect :: Region → Region → Region
  Translate :: Vector → Region → Region

size :: Region → Int
size (Circle    _) = 1
size (Outside    a) = size a + 1
size (Union     a b) = size a + size b + 1
size (Intersect a b) = size a + size b + 1
size (Translate _ a) = size a + 1
```

                    (a) A shallow embedding   (b) A deep embedding

Fig. 1. The initial system for the region DSL.

(2) **Linguistic reuse.** We create a region with structure sharing to assess the difficulty of reusing host-language optimizations.

(3) **Extensibility with new constructs.** We extend the region language with three more constructs: univ for the universal region that contains all points, empty for the empty region that contains no points, and scale for resizing a region by two scale factors in a vector.

(4) **Extensibility with new operations.** We add a new interpretation that checks whether a given point resides in a region.

(5) **Complex interpretations, transformations, and optimizations.** Last but not least, we discuss three kinds of more complex interpretations that illustrate dependent interpretations (i.e. interpretations that need to call other interpretations in their implementations) and transformations over the AST of the region language.

2.2 Initial System

Fig. 1 shows the definitions of the initial DSL, including the language interface and a simple interpretation for shallow and deep embeddings.

Language interfaces. In the shallow embedding, Region denotes the semantic domain. All the five language constructs are implemented as functions (called denotation functions), which implement the respective semantics of the language constructs. Thus, the language interface is essentially the signature of the denotation functions and is entangled with the definition of a particular semantic domain. In the deep embedding, Region is defined as an algebraic data type[1]. Algebraic data types only capture the abstract syntax, thus enabling the language interface to be separated from any concrete semantics.

Semantic interpretations. There can be many semantic interpretations, which associate language constructs with their meanings. The initial semantics that we choose here is an abstract interpretation, which calculates the syntactic size of a region (i.e. the number of AST nodes). We opt to have an interpretation that is slightly artificial, but simple, so that we can focus on the key aspects of compositional embeddings. In the shallow embedding, the semantics is defined together with

---

[1]We use the notation of *generalized* algebraic data types [Peyton Jones et al. 2006] to make type annotations clear.

the language interface since we must specify some concrete type (`Int` in this case) for Region. In contrast, to create a new semantic interpretation in the deep embedding, we define a function `size` using pattern matching over the Region data type.

We can see that the two embeddings work quite differently: the shallow embedding encodes semantics in region constructors, and the interpretation function from Region to `Int` is merely the identity function; the deep embedding does nothing concerning constructors but hands over the work to the interpretation function `size`.

2.3 Linguistic Reuse and Meta-Language Optimizations

An important concern for embedded DSLs is *linguistic reuse* [Krishnamurthi 2001]. One of the key selling points of embedded DSLs is that they can reuse much of the infrastructure of the host language and inherit various optimizations that are available in the host language. However, there are important differences between shallow and deep embeddings with respect to linguistic reuse. As widely observed in previous work [Jovanović et al. 2014; Rompf et al. 2012; Svenningsson and Axelsson 2015], shallow embeddings make linguistic reuse easier. To illustrate this, we can create a region that contains a series of repeated subregions with horizontally aligned circles:

```
circles = go 20 (2 ** 18)
  where go :: Int → Double → Region
        go 0 offset = circle 1
        go n offset = let shared = go (n - 1) (offset / 2)
                       in union (translate Vector { x = -offset, y = 0 } shared)
                                (translate Vector { x =  offset, y = 0 } shared)
```

The region `circles` is defined via an auxiliary recursive function `go`. In the shallow embedding, `shared` is of type `Int`, and using `shared` twice in the **let**-body will avoid interpreting the region twice. The code above also works for the deep embedding, assuming some smart constructors such as `circle = Circle`. However, in the deep embedding, `shared` is an AST of type Region. In this case, the AST will be duplicated in the **let**-body, and later when we interpret the region, we need to traverse the same AST twice, duplicating the size calculation. Since `go` is recursive, it will lead to a lot of sharing for the shallow approach and a lot of repetition for the deep approach. As a result, the evaluation of `circles` is instantaneous in the shallow approach, whereas in the deep approach, it basically does not terminate in a reasonable amount of time. In short, in shallow embeddings, we are able to preserve sharing by naturally exploiting the sharing optimizations present in the host language, but sharing is lost in deep embeddings.

We should remark that this particular issue regarding sharing (which is just an instance of linguistic reuse) is well known [Gill 2009; Kiselyov 2011; Oliveira and Löh 2013]. There have been several proposed techniques that enable preserving sharing in deeply embedded DSLs. However, all those techniques add extra complexity to the DSL encoding, and they may even make the DSL harder to use. This is in contrast to the shallow approach, where no extra work is required by the programmer and host-language features are naturally reused.

2.4 Adding More Language Constructs

As part of evolving the DSL with additional features, we may decide to add more constructs for regions. To evaluate how easy it is to add more language constructs, we add three more language constructs for the universal or empty region as well as a scaling operator. Shallow embeddings make such extensions very easy. We only need to add three new denotation functions:

```
univ :: Region         empty :: Region         scale :: Vector → Region → Region
univ = 1               empty = 1               scale _ a = a + 1
```

Notwithstanding the modular extension in shallow embeddings, it is awkward to add language constructs in deep embeddings. We have to modify the algebraic data type Region and all related interpretation functions to add new cases. This is not modular because we must change the existing code. Even if we have access to previous definitions, it is inevitable to recompile all the code that depends on Region.

In essence, once we start adding new features, we are quickly faced with an instance of the *Expression Problem* [Wadler 1998]. Shallow embeddings make adding constructs easy, whereas adding new constructs in deep embeddings is more difficult and non-modular. As we shall see next, when adding new semantic interpretations, we have a dual problem.

2.5 Adding a New Interpretation

The syntactic size is too abstract to describe a region, so let us add a new interpretation that checks whether a region contains a given point. For example, a circular region of radius r contains (x, y) if and only if $x^2 + y^2 \leq r^2$. Although adding constructs is awkward in deep embeddings, adding a new interpretation function is trivial:

```
contains :: Region → Vector → Bool
Circle             r `contains` p = p.x ** 2 + p.y ** 2 ≤ r ** 2
Outside            a `contains` p = not (a `contains` p)
Union            a b `contains` p = a `contains` p || b `contains` p
Intersect        a b `contains` p = a `contains` p && b `contains` p
Translate Vector {..} a `contains` p = a `contains` Vector { x = p.x − x, y = p.y − y }
```

The code is mostly straightforward[2]. The only minor remark is that we use a record wildcard (Vector {..}) in the case of Translate to bring record fields (x and y) into scope.

However, there is no easy modular way to support multiple interpretations in shallow embeddings. We need to completely change the definition of Region and remap all the language constructs to a new semantic domain. The issue of semantic extension in shallow embeddings is dual to that of language extension in deep embeddings. To address the tension between the two dimensions, some alternative embeddings are proposed, such as tagless-final embeddings [Carette et al. 2009; Kiselyov 2010] and polymorphic embeddings [Hofer et al. 2008], though they still have some significant drawbacks, which will be revealed shortly.

2.6 Dependencies, Complex Interpretations, and Domain-Specific Optimizations

One area where deeply embedded DSLs shine is in enabling more complex kinds of interpretations. These interpretations may, for example, enable transformations or rewritings on the AST, which are helpful for writing domain-specific optimizations, among other things. For writing such complex forms of interpretations, we often require multiple *dependent* functions defined by pattern matching, and sometimes we may even need nested pattern matching. Such interpretations are very challenging for shallow DSLs and are often the reason why DSL writers prefer deep embeddings.

Dependent interpretations. Let us start with a dependent interpretation that shows a text representation of a region using a deep embedding:

```
text :: Region → String
text   (Circle      r) = "a circular region of radius " ++ show r
text   (Outside     a) = "outside a region of size " ++ show (size a)
text s@(Union     _ _) = "the union of two regions of size " ++ show (size s) ++ " in total"
text s@(Intersect _ _) = "the intersection of two regions of size " ++ show (size s) ++ " in total"
text s@(Translate _ _) = "a translated region of size " ++ show (size s)
```

---

[2]Two GHC language extensions are required here: *OverloadedRecordDot* and *RecordWildCards*.

Note that the definition of text *depends* on the previously defined size function. Such a definition is challenging in shallow embeddings. A workaround sometimes used in shallow embeddings is to use tuples to define multiple interpretations simultaneously. For example, we can define size and text together as:

```
type Region = (Int, String) -- (size, text)
circle     r = (1, "a circular region of radius " ++ show r)
outside    a = (fst a + 1, "outside a region of size " ++ show (fst a))
union    a b = (size, "the union of two regions of size " ++ show size ++ " in total")
  where size = fst a + fst b + 1
intersect a b = (size, "the intersection of two regions of size " ++ show size ++ " in total")
  where size = fst a + fst b + 1
translate _ a = (size, "a translated region of size " ++ show size)
  where size = fst a + 1
```

Mutually dependent interpretations. A similar, but more interesting, example is two interpretations for checking universality and emptiness:

```
isUniv :: Region → Bool                 isEmpty :: Region → Bool
isUniv Univ            = True           isEmpty Empty           = True
isUniv (Outside    a) = isEmpty a       isEmpty (Outside    a) = isUniv a
isUniv (Union    a b) = isUniv a || isUniv b    isEmpty (Union    a b) = isEmpty a && isEmpty b
isUniv (Intersect a b) = isUniv a && isUniv b   isEmpty (Intersect a b) = isEmpty a || isEmpty b
isUniv (Translate _ a) = isUniv a       isEmpty (Translate _ a) = isEmpty a
isUniv (Scale     _ a) = isUniv a       isEmpty (Scale     _ a) = isEmpty a
isUniv _               = False          isEmpty _               = False
```

Unlike in the previous example, where only text depends on size, the two definitions are *mutually recursive*, depending on each other. If we want to rewrite them using shallow embeddings, they also have to be encoded as a pair to call each other via **fst** and **snd**:

```
type Region = (Bool, Bool) -- (isUniv, isEmpty)

univ    = (True,  False)        union    a b = (fst a || fst b, snd a && snd b)
empty   = (False, True)         intersect a b = (fst a && fst b, snd a || snd b)
circle _ = (False, False)       translate _ a = (fst a,          snd a)
outside a = (snd a, fst a)      scale    _ a = (fst a,          snd a)
```

Generally speaking, using tuples to deal with dependencies is non-modular and awkward to use in that interpretations become tightly coupled. Adding more interpretations with dependencies, for example, would require a complete rewrite of the code: dependent interpretations have to be defined together with the interpretations that they depend on. More advanced encodings, such as tagless-final embeddings [Kiselyov 2010] or polymorphic embeddings [Hofer et al. 2008], provide ways to modularize *independent* interpretations into reusable units, but they cannot scale well to *dependent* interpretations and often have to resort to similar techniques using tuples like the code above. In Appendix A, we encode the two examples above in tagless-final embeddings and illustrate that they suffer from the same problems as shallow embeddings with respect to dependencies, both mutual and non-mutual.

Nested pattern matching. Another strength of deep embeddings is their ability to perform nested pattern matching, which is rather useful to inspect smaller constructs. Here we take an optimization that eliminates double negation for example. Nested pattern matching is used to check if an Outside is directly wrapped around an outer Outside. If so, both of them are eliminated; otherwise, eliminate is recursively called with inner constructs:

```
eliminate :: Region → Region
eliminate (Outside (Outside a)) = eliminate a
eliminate (Outside          a) = Outside (eliminate a)
eliminate (Union          a b) = Union (eliminate a) (eliminate b)
eliminate (Intersect      a b) = Intersect (eliminate a) (eliminate b)
eliminate (Translate      v a) = Translate v (eliminate a)
eliminate (Scale          v a) = Scale v (eliminate a)
eliminate                    a = a
```

The common use of nested pattern matching poses a second challenge to domain-specific optimizations. Kiselyov [2010] discusses the "*seemingly impossible pattern-matching*" problem with nested patterns, which appear not directly encodable in tagless-final embeddings. Instead, he proposes that such algorithms can often be implemented in a tagless-final interpreter as context-dependent interpretations by essentially changing the algorithm. An extra parameter is added so the semantic type looks like Ctx → repr. However, in the general case, it is not obvious how to convert an algorithm with nested pattern matching into one based on contexts.

3 COMPOSITIONAL EMBEDDINGS

After looking back at the existing embedding techniques, this section shows that Compositional Programming enables a new form of DSL embedding called a *compositional embedding*. Compositional Programming is a statically typed modular programming paradigm implemented by the CP language. For the details of the semantics of CP, we refer the reader to work by Zhang et al. [2021]. We have reimplemented CP as an in-browser interpreter, which can be directly run on a serverless web page. There are some small syntactic differences between our implementation and the original work, which we point out when necessary. Additionally, our implementation uses the call-by-need evaluation strategy, which optimizes the original call-by-name one.

Here, we only introduce the necessary language features of CP to illustrate compositional embeddings. With compositional embeddings, we can get many of the benefits of both shallow and deep embeddings without most of the drawbacks. We revisit the challenges illustrated in Section 2 using our approach in this section. A detailed comparison between compositional embeddings and various existing forms of embeddings, including shallow, deep, hybrid, and polymorphic embeddings, is presented at the end of this section.

3.1 Initial System, Revisited: Compositional Interfaces and Traits

As before, we start with an initial language interface inspired by Hudak [1998] and an abstract interpretation that calculates the syntactic size of a region:

```
type Vector = { x : Double; y : Double };          type Size = { size : Int };
type HudakSig<Region> = {                          sz = trait implements HudakSig<Size> ⇒ {
  Circle    : Double → Region;                       (Circle    _).size = 1;
  Outside   : Region → Region;                       (Outside   a).size = a.size + 1;
  Union     : Region → Region → Region;              (Union     a b).size = a.size + b.size + 1;
  Intersect : Region → Region → Region;              (Intersect a b).size = a.size + b.size + 1;
  Translate : Vector → Region → Region;              (Translate _ a).size = a.size + 1;
};                                                 };
```

Since CP is also statically typed, a *compositional interface* (HudakSig) serves as the specification of the DSL syntax, playing a similar role to algebraic data types in the deep embedding. The type parameter Region is called the *sort* of the interface HudakSig. Sorts can be instantiated with different concrete types that represent different semantic domains. Every constructor, which is capitalized, must return a sort. Compositional interfaces are implemented by *traits* [Bi and Oliveira 2018;

Ducasse et al. 2006], which serve as the unit of code reuse in CP. Traits play a similar role to classes in traditional object-oriented languages and are used to provide implementations for interpretations of the region DSL. To implement the abstract interpretation of the DSL, we first define the semantic domain type `Size`. Then we define a trait that implements `HudakSig<Size>`. In the body of the trait, we use *method patterns*, such as `(Circle _).size = 1`, to define the implementation. The method patterns used in the trait body allow us to define interpretations that look very much like the corresponding definition of `size` by pattern matching in Haskell in Fig. 1b.

3.2 Linguistic Reuse and Meta-Language Optimizations

The example of horizontally aligned circles in Section 2.3 can be translated to CP as:

```
sharing Region = trait [self : HudakSig<Region>] ⇒ {
  circles = letrec go (n:Int) (offset:Double) : Region =
              if n == 0 then Circle 1.0
              else let shared = go (n-1) (offset/2.0)
                   in Union (Translate { x = -offset; y = 0.0 } shared)
                            (Translate { x =  offset; y = 0.0 } shared)
            in go 20 (pow 2.0 18);
};
```

Since the region DSL can have multiple interpretations, `circles` should be oblivious of any concrete interpretation but only refer to the language interface. To achieve this, we use *self-type annotations*. The trait `sharing` is parametrized by a type parameter `Region` and annotated with a self-type `HudakSig<Region>`. Like Scala traits, CP traits can be annotated with self-types to express abstract dependencies. This serves as an elegant way to inject region constructors. All of the constructors formerly declared in the interface are available through self-references, for instance, `self.Circle` ("`self.`" can be omitted for the sake of convenience). These constructors are *virtual* [Ernst et al. 2006; Madsen and Møller-Pedersen 1989]: they are not attached to a specific implementation.

With the self-type dependency on `HudakSig<Region>`, the static type checker of CP rejects a direct instantiation like **new** `sharing @Size` (@ is the prefix for a type argument). Instead, we need to compose `sharing` with a trait implementing `HudakSig<Size>`, using a *merge operator*[3] [Dunfield 2014]:

```
test = new sharing @Size , sz;    -- test = new ((sharing @Size) , sz);
test.circles.size                 -- Above is an equivalent definition with redundant parentheses.
```

The merge of the two traits (`sharing @Size` and `sz`) is still a trait. With self-type dependencies satisfied, the merged trait can be instantiated successfully, and we can call its method. According to call-by-need evaluation, the result of `shared` is stored for subsequent use once evaluated. Thanks to the linguistic reuse of the host-language **let** expressions and their optimizations, evaluating `circles.size` is able to exploit sharing and terminates almost immediately. This is similar to the Haskell approach in Section 2.3 using shallow embeddings.

3.3 Adding New Language Constructs

In compositional embeddings, language constructs can be modularly added. We first create a new language interface inspired by Hofer et al. [2008] and then define a trait implementing it:

---

[3]We simplified the notation of the merge operator from double commas (,,) to a single comma (,).

```
type HoferSig<Region> = {                              sz' = trait implements HoferSig<Size> ⇒ {
  Univ  : Region;                                        (Univ     ).size = 1;
  Empty : Region;                                        (Empty    ).size = 1;
  Scale : Vector → Region → Region;                      (Scale _ a).size = a.size + 1;
};                                                     };
```

To compose multiple interfaces, we use *intersection types*. To illustrate this, we create a repository of shapes that make use of language constructs from both interfaces:

```
type RegionSig<Region> = HudakSig<Region> & HoferSig<Region>;
repo Region = trait [self : RegionSig<Region>] ⇒ {
  ellipse = Scale { x = 4.0; y = 8.0 } (Circle 1.0);
  universal = Union (Outside Empty) (Circle 1.0);
};
```

RegionSig is an intersection type combining the two interfaces into one. The definition of repo is similar to the definition of sharing in Section 3.2. It defines an ellipse, by using the Scale constructor from HoferSig and the Circle constructor from HudakSig. Similarly, it defines an essentially universal region by using constructors from both signatures.

3.4 Adding New Interpretations

Besides language constructs, it is also trivial to add new semantic interpretations in compositional embeddings. Just like creating a new interpretation function in deep embeddings, we only need to define a new trait implementing a compositional interface:

```
type Eval = { contains : Vector → Bool };
eval = trait implements RegionSig<Eval> ⇒ {
  (Circle         r).contains p = pow p.x 2 + pow p.y 2 ≤ pow r 2;
  (Outside        a).contains p = !(a.contains p);
  (Union        a b).contains p = a.contains p || b.contains p;
  (Intersect    a b).contains p = a.contains p && b.contains p;
  (Translate {..} a).contains p = a.contains { x = p.x − x; y = p.y − y };
  (Univ           ).contains _ = true;
  (Empty          ).contains _ = false;
  (Scale     {..} a).contains p = a.contains { x = p.x / x; y = p.y / y };
};
```

Similarly to Haskell, CP supports record wildcards {..}, which bring record fields into scope. We instantiate the sort with a function checking if a point resides in a region.

Composing multiple interpretations. Now that we have multiple interpretations, we use *nested trait composition*, provided by the merge operator, to compose all interpretations:

```
shapes = new repo @(Eval&Size) , eval , sz , sz';  -- nested trait composition
"The elliptical region of size " ++ toString shapes.ellipse.size ++ (if shapes.ellipse.contains {
    x = 0.0; y = 0.0 } then " contains " else " does not contain ") ++ "the origin."
--> "The elliptical region of size 2 contains the origin."
```

We provide the final semantic domain (Eval&Size), as the type argument, for the trait repo and merge it with three other modularly defined traits. The trait composition is called *nested* [Bi et al. 2018] because not only different sets of constructors, but also different semantics nested in the same constructor are merged. In our case, both interpretations are available under the two sets of language constructs after nested composition, as visualized in Fig. 2. Here, the language constructs and their semantics form a two-level hierarchy of traits. Such composition of whole hierarchies can be traced back to *family polymorphism* [Ernst 2001]. With this powerful mechanism of nested
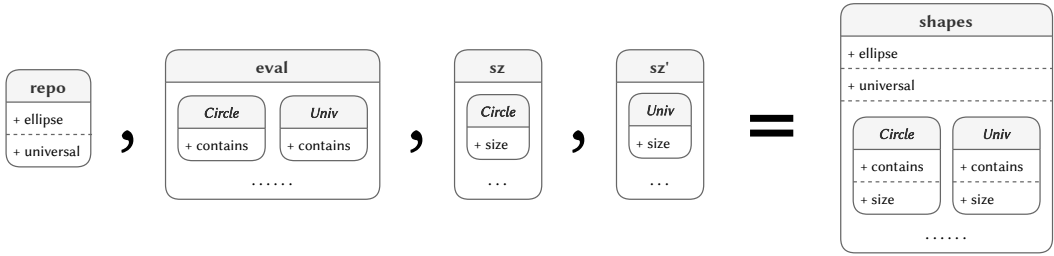
Fig. 2. A visualization of nested trait composition: inner components of traits are recursively composed.

composition, we can easily develop highly modular *product lines* [Apel et al. 2013] of DSLs, where DSL features can be composed *à la carte*.

3.5 Dependency Injection and Domain-Specific Optimizations

Now we illustrate CP's support for *modular dependent interpretations*. For space reasons, we skip the simpler example with text (which can be found in our online implementation) and focus on the more interesting example with isUniv and isEmpty. Similarly to the deep embedding in Haskell in Section 2.6, we define two compositional traits chkUniv and chkEmpty:

```
type IsUniv  = { isUniv  : Bool };        type IsEmpty = { isEmpty : Bool };

chkUniv = trait                           chkEmpty = trait
  implements RegionSig<IsEmpty ⇒ IsUniv> ⇒ {    implements RegionSig<IsUniv ⇒ IsEmpty> ⇒ {
 (Univ         ).isUniv = true;           (Empty         ).isEmpty = true;
 (Outside     a).isUniv = a.isEmpty;      (Outside     a).isEmpty = a.isUniv;
 (Union     a b).isUniv = a.isUniv || b.isUniv;  (Union     a b).isEmpty = a.isEmpty && b.isEmpty;
 (Intersect a b).isUniv = a.isUniv && b.isUniv;  (Intersect a b).isEmpty = a.isEmpty || b.isEmpty;
 (Translate _ a).isUniv = a.isUniv;       (Translate _ a).isEmpty = a.isEmpty;
 (Scale     _ a).isUniv = a.isUniv;       (Scale     _ a).isEmpty = a.isEmpty;
             _.isUniv = false;                        _.isEmpty = false;
};                                        };
```

The first thing to note is that the sort of RegionSig is instantiated with two types separated by a fat arrow ($\Rightarrow$) [4]. This is how we *refine* interface types for dependency injection. For example, chkUniv does not implement RegionSig<IsEmpty>, but we assume the latter is available. The static type checker of CP guarantees that chkUniv is later merged with another trait that implements RegionSig<IsEmpty>. That is why we can safely use a.isEmpty when implementing (Outside a).isUniv. The second trait chkEmpty is just the dual of chkUniv. It is hard to define such interpretations modularly in traditional approaches because they are dependent, but we make them compositional via dependency injection. Lastly, _.isUniv and _.isEmpty are *default patterns*, which compensate for the remaining constructors declared in the interface. For example, _.isUniv = **false** implies Empty.isUniv = **false** and Circle.isUniv = **false**. Note that in CP, for modularity, patterns are *unordered*, unlike functional languages like Haskell or ML. Therefore, default patterns are local to the traits where they are used. These patterns can be understood as a concise way to fill the missing cases in the local trait.

Another thing worth noting is that dependency injection in compositional embeddings is more modular than direct dependencies in deep embeddings because the former does not refer to concrete implementations. For example, in the compositional embedding above, chkUniv only requires that

---

[4]$\Rightarrow$ was originally denoted by % in work by Zhang et al. [2021].

the dependent trait should implement the interface `RegionSig<IsEmpty>`. However, in the deep embedding (see Section 2.6), isUniv depends on the specific `isEmpty` implementation.

Delegated method patterns. In some complicated transformations, nested pattern matching is required to inspect smaller constructs. Nested pattern matching is not directly supported in CP yet. However, there is a simple transformation that we can do whenever we would like to have nested patterns: we can delegate the inner tasks to other methods whereby only top-level patterns are needed. We call such a technique *delegated method patterns*.

Concerning the nested pattern matching in Section 2.6, we can implement it with two mutually dependent methods. Since delegated method patterns are not reused, the two methods are defined together for the sake of simplicity (but can always be separated into two traits):

```
type Eliminate Region = { eliminate : Region; delOutside : Region };
elim Region = trait [fself : RegionSig<Region>]
              implements RegionSig<Region ⇒ Eliminate Region> ⇒ {
  (Outside     a).eliminate = a.delOutside;
  (Union     a b).eliminate = Union a.eliminate b.eliminate;
  (Intersect a b).eliminate = Intersect a.eliminate b.eliminate;
  (Translate v a).eliminate = Translate v a.eliminate;
  (Scale     v a).eliminate = Scale v a.eliminate;
         [self].eliminate = self;

  -- delegated method patterns:
  (Outside a).delOutside = a.eliminate;
      [self].delOutside = Outside self.eliminate;
};
```

The translation from nested pattern matching to delegated method patterns is straightforward. Our code lifts nested patterns (Outside (Outside a)) to top-level delegated patterns. In this way, we do not change the underlying algorithm at all. This is in contrast with approaches like tagless-final embeddings that, as discussed in Section 2.6, seem unable to directly support nested patterns, requiring different algorithms to achieve the same goal. Though our approach is not as convenient as deep embeddings, it is noteworthy that we can just write the original algorithm *modularly*.

If we add new language constructs in the future, it is easy to augment the traits with more top-level patterns, but extending nested patterns is difficult because there is no name to denote the *extension point*. With delegated method patterns, the method name `delOutside` offers an extension point for additional cases in the nested pattern matching. For instance, if we wish for an extension supporting a new kind of region and a special case for eliminating a region outside this kind of region (eliminate (Outside (SomeRegion params))), we can have a trait with a method pattern of the following form, modularly defined in another trait:

```
(SomeRegion params).delOutside = ...
```

Although we believe that the design of CP can be improved to better support similar logic to nested patterns, there are important differences between traditional (closed) forms of pattern matching and open pattern matching [Zhang and Oliveira 2020]. For instance, patterns in CP are unordered for compositionality, but the order of patterns matters in conventional pattern matching. Thus the design of improved language support for nested patterns in CP, which could make the use of delegated method patterns more convenient, requires further research and is left for future work.

Linguistic reuse after transformations. Before finishing the discussion of modular transformations, let us revisit CP's support for linguistic reuse: can we still have meta-language optimizations for free after complicated transformations? The answer is yes. To demonstrate this point, we can modify

Table 1. A comparison between different embedding approaches.

	Shallow	Deep	Hybrid	Poly.	Comp.
Transcoding free	●	●	○	●	●
Linguistic reuse	●	○	●	●	●
Language construct extensibility	●	○	◐[1]	●	●
Interpretation extensibility	○	●	●	●	●
Transformations and optimizations	○	●	●	◐[2]	◐[2]
Linguistic reuse after transformations	n/a	○	○	●	●
Modular dependencies	○	◐[3]	◐[3]	○	●
Nested pattern matching	○	●	●	○	◐[4]

[1] The extensibility of language constructs is limited or precludes exhaustive pattern matching.
[2] Transformations require some ingenuity and are sometimes awkward to write.
[3] Dependencies do not require code duplication but still refer to concrete implementations.
[4] Nested pattern matching is implemented as delegated method patterns.

the definition of `circles` in Section 3.2 to have an inefficient `Outside (Outside (Circle 1.0))`. As usual, we compose all the necessary traits together to make sure no dependencies are missing:

```
test' = new sharing' @(Size & Eliminate Size) , sz , sz' , elim @Size;
test'.circles.eliminate.size
```

The evaluation terminates as quickly as before, meaning that meta-language optimizations are still performed even after a transformation.

We should remark that there are limits to the form of implicit sharing (and linguistic reuse) that shallow embeddings or compositional embeddings provide. For both shallow and compositional embeddings, implicit sharing will not work if the interpretation is a function, such as `contains`. In such cases, the sharing is still lost. Nevertheless, implicit sharing is an important feature that is often exploited in shallow DSLs. With compositional embeddings, we can take this idea further and make it work even after some transformations and optimizations have been applied. Moreover, it is also possible to adopt solutions with explicit sharing by modeling a `Let` construct in the DSL.

3.6 A Detailed Comparison between Different Embedding Approaches

In Table 1, we compare existing embedding approaches in the literature with compositional embeddings. Shallow and deep embeddings have been discussed in detail in Section 2, and the table summarizes the points that we have made. Hybrid approaches [Jovanović et al. 2014; Rompf et al. 2012; Svenningsson and Axelsson 2015] employ both embeddings together. However, transcoding from shallow to deep is generally needed, as both shallow and deep implementations must coexist side by side. There is a clear boundary between the two parts: linguistic reuse is available only in the shallow part, while complex transformations are available only in the deep part. Therefore, it is still hard to exploit host-language features after transformations. In work by Svenningsson and Axelsson, we cannot add more constructs to the deeply embedded AST but can only extend shallowly embedded syntactic sugar. In Scala, if *open* case classes [Emir et al. 2007] are used for deep embeddings, ASTs are also extensible but do not ensure the exhaustiveness of pattern matching. Compositional embeddings offer a unified approach directly capable of all these functionalities.

Non-modular dependencies in modular embeddings. Polymorphic embeddings [Hofer et al. 2008], tagless-final embeddings [Carette et al. 2009; Kiselyov 2010], and object algebras [Oliveira and Cook 2012] (denoted as Poly. in the table) also provide a unified encoding where most advantages

of shallow and deep embeddings are available. However, they cannot deal with dependencies modularly. Let us take tagless-final embeddings for example, which can also be implemented in Haskell. We refer curious readers to Appendix A for the complete code.

In the tagless-final embedding, region constructors are defined in a type class instead of a closed algebraic data type, and Size can be modularly defined as an instance of the type class since it has no dependency:

```haskell
class RegionHudak repr where
  circle    :: Double → repr
  outside   :: repr → repr
  union     :: repr → repr → repr
  intersect :: repr → repr → repr
  translate :: Vector → repr → repr
```

```haskell
newtype Size = S { size :: Int }
instance RegionHudak Size where
  circle        _ = S 1
  outside       a = S $ a.size + 1
  union       a b = S $ a.size + b.size + 1
  intersect   a b = S $ a.size + b.size + 1
  translate   _ a = S $ a.size + 1
```

But how about the textual representation? As a first try, we might write:

```haskell
newtype Text = T { text :: String }
instance RegionHudak Text where
  circle  r = T { text = "a circular region of radius " ++ show r }
  outside a = T { text = "outside a region of size " ++ show a.size }
  ......
```

Unfortunately, we will get a type error concerning a.size because a has type Text and thus does not contain a field named size. Once we need operations that have some dependencies on other operations, we get into trouble!

A simple workaround is to pack the two operations together and duplicate the code of size calculation. We have already described this approach for shallow embeddings in Section 2.6, and the same workaround works for tagless-final embeddings:

```haskell
data SizeAndText = ST { size :: Int, text :: String }
instance RegionHudak SizeAndText where
  circle  r = ST { size = 1, text = "a circular region of radius " ++ show r }
  outside a = ST { size = a.size + 1, text = "outside a region of size " ++ show a.size }
  ......
```

However, this is not modular since we have to duplicate code for size calculation, even though the size calculation is completely independent of the textual representation. If we have another operation that depends on size, we have to repeat the same code even again. This workaround is an *anti-pattern*, which violates basic principles of software engineering. Duplicating code every time we need a dependent interpretation is a serious problem since dependencies are extremely common. Nearly all non-trivial code will have dependencies. In functional programming, it is common to have functions defined by pattern matching that call other functions defined by pattern matching, which is where an important form of dependencies appears. As shown in Appendix B, some workarounds that avoid code duplication in Haskell are possible, but this typically comes at the cost of more code and the use of advanced language features.

The third dimension of the Expression Problem. In essence, dependencies are a third dimension that is not discussed in the formulation of the Expression Problem by Wadler [1998]: whether dependencies require code duplication or not. If we take this into consideration, we can get a revised formulation of the Expression Problem, where instead of only two dimensions, we have a third dimension that evaluates the ability of an approach to deal with dependencies. With this extra dimension taken into account, what we get is Table 2. While tagless-final or other embeddings address the two original dimensions of extensibility modularly, they become non-modular with

Table 2. A briefer comparison with respect to the three dimensions of the Expression Problem.

	FP	OOP	Poly.	Comp.
New constructs	○	●	●	●
New functions	●	○	●	●
Dependencies	●	●	○	●

● no code duplication   ○ code duplication needed

respect to dependencies, and programming with dependencies becomes significantly more awkward compared to conventional FP or OOP.

Note that we are not the first to observe the problem with dependencies. The problem above is known and discussed widely in the literature. In the context of polymorphic embeddings, Hofer et al. [2008] face the problem of dependencies and use the non-modular approach with tuples. Later Hofer and Ostermann [2010] adopt an extensible visitor pattern to improve modularity, but this results in much more boilerplate code. In his lecture notes on tagless-final interpreters, Kiselyov [2010] has a strong focus on discussing how to write non-compositional interpretations, which basically include operations with dependencies and nested pattern matching. He shows that non-compositional programs can often be rewritten as programs that use contexts instead. The problem of modular dependencies has been widely discussed within the context of object algebras. Most recently, Zhang and Oliveira [2017, 2020] propose solutions to the problem using meta-programming techniques. Zhang and Oliveira [2019] further explore modular solutions in both Scala and Haskell, but their approaches still require a lot of boilerplate code. In Scala, it is due to a lack of language support for nested composition, whereas Haskell needs to encode dependencies via type classes to modularize dependent interpretations (see Appendix B for a similar solution). The drawbacks of the existing solutions above motivate the design of CP and compositional embeddings.

Transformations. It is widely believed that transformations and optimizations are much easier to write in deep embeddings [Jovanović et al. 2014; Scherr and Chiba 2014]. We agree that it requires some ingenuity and is sometimes awkward to write complex transformations in compositional embeddings, but Kiselyov [2010] has demonstrated that several challenging transformations can be written with tagless-final embeddings. Compositional programming is a generalization of such forms of embeddings, and therefore all the transformations are possible with tagless-final embeddings are also possible in CP. Even for structurally non-local transformations, we can instantiate a sort with a function type (e.g. `RegionSig<Ctx→Region>`) and pass down the context when traversing the AST. Such contextual transformations can also be modular with the help of *polymorphic contexts* [Zhang et al. 2021]. We have implemented an example of common subexpression elimination inspired by Kiselyov [2011] (which can be found in our online implementation). Both our code and Kiselyov's are not fully modular with respect to an auxiliary data structure (his implementation relies on a closed algebraic data type). In our case, this is because nested composition for recursive types is currently not supported, which relies on future theoretical progress of reconciling distributive subtyping and recursive types. Nevertheless, our approach is less entangled than tagless final embeddings since we can handle dependent transformations modularly.

4 EₓT: A DSL FOR DOCUMENT AUTHORING

We have presented the advantages of compositional embeddings using a relatively small DSL. One may wonder if our approach scales to larger DSLs. As our answer to this question, we developed a larger DSL for document authoring called EₓT.

E$_X$T applies compositional embeddings to a document DSL, inspired by LATEX and Scribble [Flatt et al. 2009]. We have also done minor syntactic extensions to the CP language to support writing documents more naturally. Since documents usually involve large portions of text, it would be awkward to write such portions of text using string literals adopted by most programming languages. CP does not yet have facilities for syntactic extension that other languages (e.g. Racket, Haskell, or Coq) offer, so we have extended the parser directly. Thus, CP parses some document-specific syntax and desugars it to compositionally embedded fragments during parsing. Such a generative approach is similar to Racket macros [Ballantyne et al. 2020] and Template Haskell [Sheard and Peyton Jones 2002], which provide more flexible facilities for performing such desugaring. Nevertheless, the surface syntax of E$_X$T is essentially a set of lightweight syntactic sugar for its underlying compositional embeddings (similar, for instance, to Scribble in Racket).

4.1 Design Goals and Non-Goals

There are already a large number of document DSLs, so why shall we create yet another one? We explain why by identifying important design goals and non-goals of E$_X$T.

A document language for the web. A first goal is to have a lightweight but powerful document language tailored mostly for the web. We view E$_X$T as an alternative to Wikitext [MediaWiki 2003], the markup language used by Wikipedia, which shares similar goals. Similar to Wikipedia pages, users can directly edit E$_X$T documents on a web page. But different from Parsoid [MediaWiki 2011], the official Wikitext parser that runs on the server side, our implementation can directly render documents on the browser side.

General-purpose computation. The majority of existing document DSLs (e.g. CommonMark [Mac-Farlane 2014], Textile [Allen 2002], and reStructuredText [Goodger 2002]) have a fixed set of language features and do not provide general-purpose computation. For instance, we may want to compute a table from some data, like a table listing the largest cities in descending order of population. For such kinds of tasks, it is useful to have mechanisms for general-purpose computation that allow us to *compute* documents rather than manually write them.

Although some other document DSLs provide language constructs that enable some computation, they still have significant limitations. For instance, Wikitext provides templates, which play a role similar to functions in conventional programming languages. However, templates cannot perform arbitrary computation as CP does. The documentation says, "A template can call itself but will stop after one iteration to prevent an infinite loop." [5] In other words, templates by themselves are incapable of recursion or loops, but in practice, loops are often needed in Wikipedia. For instance, {{loop}} has been used on approximately 99,000 pages [6]. A Lua extension was introduced to bypass such restrictions, and the widely-used template {{loop}} invokes string.rep in Lua. Instead of handing it over to an external language, CP directly supports functions, and thus general-purpose computation is easily done in E$_X$T, including recursion.

Static type checking. In contrast to many document languages, CP and E$_X$T are statically type checked. Static type checking enables us to detect some errors that may arise when processing documents. In applications like wikis, it is very frequent that we need some form of structured data. States or cities, for example, may have infoboxes associated with them that list several pieces of data, such as areas, populations, official languages, etc. Using simple text to store such data can easily lead to simple errors. In addition, if we want to enable computed information as previously described, then it is useful to have a type system that allows us to write functions that take a well-defined

---

[5]The sentence is extracted from https://en.wikipedia.org/wiki/Wikipedia:TEMPLATE_LOOP.
[6]The number of transclusions (as of July 2022) is shown at https://en.wikipedia.org/wiki/Template:Loop.
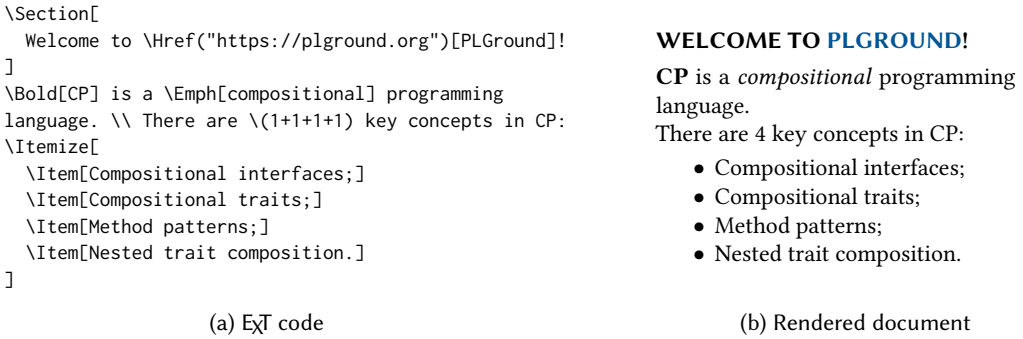
```
\Section[
  Welcome to \Href("https://plground.org")[PLGround]!
]
\Bold[CP] is a \Emph[compositional] programming
language. \\ There are \(1+1+1+1) key concepts in CP:
\Itemize[
  \Item[Compositional interfaces;]
  \Item[Compositional traits;]
  \Item[Method patterns;]
  \Item[Nested trait composition.]
]
```

WELCOME TO PLGROUND!

CP is a *compositional* programming language.

There are 4 key concepts in CP:

- Compositional interfaces;
- Compositional traits;
- Method patterns;
- Nested trait composition.

<div align="center">(a) E_XT code   (b) Rendered document</div>

<div align="center">Fig. 3. A sample document illustrating E_XT syntax.</div>

form of structured data. Most document languages that we know of, including Wikitext, LATEX, and Scribble, are not statically typed. We will discuss some examples where static typing is helpful in E_XT in Section 4.4.

Extensible meta-programming and customizability. A benefit of modeling a DSL as an algebraic data type or a compositional interface is that we can easily write meta-functions over the abstract syntax of the DSL. For instance, in the context of documents, such meta-functions could include rendering to various backends (such as LATEX or HTML) or computing a table of contents based on the document tree. In many external DSLs, such meta-functions are typically much harder to write, requiring us to directly modify the implementation. However, by embedding a DSL, the host language can act as a meta-language for the DSL and allow us to easily write meta-programs, for instance, as functions to process the AST by pattern matching.

In addition, the extensibility of compositional embeddings enables a form of extensible meta-programming: not only can we write meta-functions, but those meta-functions can be extended later to cater for new language constructs modularly defined by users. This enables DSL users to *modularly* extend the basic document language to add their own extensions, including infoboxes, vector graphics, etc. While Scribble does provide good facilities for syntactic extension and meta-programs, the document structure is fixed [Flatt and Barzilay 2009], so the DSL is not fully extensible. In short, E_XT is designed to be extensible and highly customizable by users.

Non-goals. Currently, our implementation of CP is a fairly naive call-by-need interpreter, so the performance of E_XT is not competitive with well-established tools like LATEX. Furthermore, unlike LATEX, which has extensive libraries developed over many years, there are no third-party libraries for E_XT. Thus, E_XT is *not* yet production-ready. Another non-goal is security. Since we never sanitize user-generated HTML, it is easy to perform JavaScript code injection on our website. It is off-topic to consider how to ensure web security here.

4.2 Syntax Overview

The syntax of E_XT is designed for easy document authoring, as shown in Fig. 3. It is reminiscent of LATEX, but still has significant differences. The similarity is that plain text can be directly written while commands start with a backslash. But we choose different brackets for command arguments:

- \Cmd[...] encloses document arguments. Square brackets indicate that a sequence of nested elements is recursively parsed. (Most commands in Fig. 3 are in such a form, for example, \Section and \Bold.)

- \Cmd(...) encloses positional arguments. Parentheses indicate that the whole construct is desugared to function application. (\Href in Fig. 3 is an example.)
- \Cmd{...} encloses labeled arguments. Braces indicate that these arguments are wrapped in a record. (As mentioned in Section 4.4, \Line{ x1 = 0.0; y1 = 0.0; x2 = 4.6; y2 = 4.8 } uses this form to distinguish different attributes.)

A command can be followed by an arbitrary number of different arguments in any order. This syntax is more flexible than the fixed form @op[...]{...} in Scribble. Although both arguments are optional in Scribble, they can neither occur more than once nor be shuffled. In addition, there are two more useful EᵪT constructs. One is string interpolation with the form \(...), which has exactly the same syntax as the Swift programming language. We can see an example of string interpolation in Fig. 3: \(1+1+1+1). It will be desugared to Str (**toString** (1+1+1+1)). This example will execute the code, which computes "4", and its string form will be interpolated in the document. The other construct is double backslashes (\\), a shorthand for newline, which is borrowed from LATEX. In EᵪT, it is equivalent to the \Endl command. There is also an example in Fig. 3, which inserts a newline character between the first and second sentences in the body.

Desugaring. As mentioned earlier, all EᵪT constructs are desugared to normal CP code. For example, the following two expressions are equivalent (the document DSL is enclosed within backticks):

```
`\Entry("id"){ hidden = true; level = 1 }[This entry is \Emph[hidden]]` -- is desugared to:
Entry "id" { hidden = true; level = 1 } (Comp (Str "This entry is ") (Emph (Str "hidden")))
```

Here, Comp and Str are two special commands that are assumed to be implemented by library authors. Comp is used for composing an AST of document elements, while Str is for plain text.

Specification. The grammar of EᵪT can be summarized in whole as follows (<arg>* means that <arg> may occur zero or more times):

```
<doc> ::= <str> | "\" <esc>                        <str> ::= text without "\" and "]"
<esc> ::= <cmd> <arg>* | "(" <exp> ")" | "\"       <cmd> ::= identifier
<arg> ::= "[" <doc> "]" | "(" <exp> ")" | "{" <rcd> "}"   <exp> ::= expression
                                                   <rcd> ::= record fields
```

4.3 Extensible Commands with Multi-Backend Semantics

EᵪT is so flexible that only a few commands are really compulsory. Additional commands and their semantics can be extended as needed. At the very beginning, we only need to define the aforementioned three special commands:

```
type DocSig<Element> = {
  Comp : Element → Element → Element;    Str : String → Element;    Endl : Element;
};
```

However, for rich text in real-world documentation, these commands are not sufficient for various document elements. Adding new commands and their semantics is not hard at all in compositional embeddings, like what we have done in Section 3.3 and Section 3.4. Language interfaces can be separately declared and then intersected with each other. Besides extensible commands, their semantics are also retargetable. We can modularly create different traits for different backends, such as HTML and LATEX:
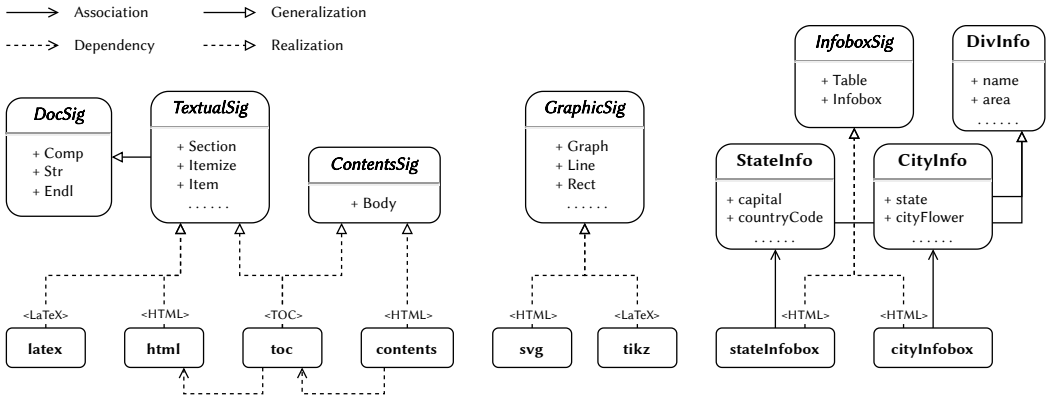
Fig. 4. A simplified diagram of E𝑋T components.

```
type HTML = { html : String };                    type LaTeX = { latex : String };
html = trait implements DocSig<HTML> ⇒ {          latex = trait implements DocSig<LaTeX> ⇒ {
  (Comp l r).html = l.html ++ r.html;               (Comp l r).latex = l.latex ++ r.latex;
  (Str    s).html = s;                              (Str    s).latex = s;
  (Endl    ).html = "<br>";                         (Endl    ).latex = "\\\\";
};                                                };
```

The components of E𝑋T. We have implemented several extensions of E𝑋T, as shown in Fig. 4. In the
upper part of the diagram, the capitalized names in a bold italic font (e.g. ***DocSig***) are compositional
interfaces, while those in a bold upright font (e.g. **DivInfo**) are normal types. The smaller boxes at
the bottom of the diagram are compositional traits. They implement different interfaces with the
sorts specified on the dashed arrows, (e.g. <HTML>). Specifically, TextualSig adds a set of common
commands for rich text, such as Section, Itemize, Item, etc. It extends DocSig and targets both
HTML and LaTeX. GraphicSig is used to draw vector graphics (including lines, rectangles, circles,
etc.), which are supported in HTML and LaTeX via SVG and Ti*k*Z respectively. InfoboxSig targets
only HTML and adds Wikipedia-like infoboxes (containing, for instance, information about areas
and populations of states or cities). All these compositional interfaces and traits can be combined
to form a heterogeneous composition. Notably, the implementation of ContentsSig, which is used
to generate a table of contents, introduces some non-trivial dependencies (see Section 5.1).

4.4 Static Typing

A major difference of E𝑋T from other document DSLs is that it is statically type checked. With
static typing, potential type errors can be detected ahead of time, saving users' time for debugging.
For example, when drawing a line using SVG, we need to use the <line> element with four
numeric coordinates (x1, y1, x2, and y2). However, since additional information of an SVG element
is represented as XML attributes, all of the coordinates have to be quoted as strings instead of
numbers. Only after rendering it with a web browser and opening developer tools can one check if
attributes are valid. In E𝑋T, we model the line construct like this:

```
type GraphicSig<Graphic> = {
  Line : { x1: Int; y1: Int; x2: Int; y2: Int } → Graphic;
  -- and more constructors
};
```

Before rendering lines via SVG, the types of arguments are checked against the language interface, and invalid values are rejected in advance. Note that the error messages are reported in terms of the E$_X$T surface syntax, which is more friendly to users than some meta-programming techniques, where errors are reported in terms of the generated code.

When modeling infoboxes and bibliography in Section 5.1, we also make use of data types to represent structured information. For example, we use `DivInfo` to model common information required by all political divisions, as well as two specific types for states and cities respectively, which extend `DivInfo` using intersection types:

```
type DivInfo = {                   type StateInfo = DivInfo & {      type CityInfo = DivInfo & {
  name       : String;               countryCode : String;            cityFlower : String;
  area       : Int;                  religions   : [String];          timeZone   : String;
  population : Int;                   -- and more fields               -- and more fields
  languages  : [String];           };                                };
};
```

In this case, there is quite a bit of structure in the data. It statically describes what fields are allowed and what types these fields have.

5 EVALUATION OF COMPOSITIONALLY EMBEDDED E$_X$T

In this section, we describe three applications of CP, compare them with alternatives in other document languages, and evaluate the use of compositionally embedded E$_X$T for the three applications.

5.1 Minipedia

Minipedia is a mini document repository of states and cities, which reconstructs a small portion of Wikipedia. Minipedia currently contains pages of the world's ten smallest countries and their capitals, as well as a structured data file storing all facts about them used for infoboxes. There are also pages consisting of sorted tables of microstates by area or population, which are computed using the information stored in the data file. Minipedia comprises over 4,000 lines of code in total, most of which is accounted for by document pages. Among the rest, the data file and E$_X$T libraries account for around 300 and 200 lines of code, respectively.

Drawbacks of Wikipedia. Compared to our approach, Wikipedia and its markup language Wikitext have several drawbacks, as partly mentioned in Section 4.1:

- All data in Wikitext is in string format. Wikitext does not differentiate data types like `Int`, `Bool`, etc. This makes type errors remain uncaught in Wikipedia infoboxes. For example, in the infobox of a country, the population field can be changed to a non-numeric meaningless value, and Wikipedia will not warn the editor at all.
- Statistical data is manually written on every Wikipedia page. This can easily result in data inconsistency for the same field across different pages, and even for different places on the same page. For example, the population size on a country page is often different from that in a statistical page listing all countries by population, owing to different data sources.
- Wikitext provides user-defined templates as a reusable unit of document fragments. They can be parametrized and play a similar role to functions. However, they do not natively support general-purpose computation like recursion. Along with the Lua extension and parser functions, Wikitext offers some computational power to Wikipedia documents, but it is not as easy to use as E$_X$T is.

Type safety and data consistency. Minipedia has a structured data file containing the information of states and cities, each piece of which is modeled as a record. The record types have been shown

in Section 4.4. Every record for states or cities is type checked against their corresponding types and collected in two arrays in the data file:

```
tuvalu = {                                     funafuti = {
  name       = "Tuvalu";                         name       = "Funafuti";
  area       = 26;                               area       = 2;
  population = 10645;                             population = 6320;
  -- and more fields                             -- and more fields
} : StateInfo;                                 } : CityInfo;
states = [tuvalu; {- and more states -}];      cities = [funafuti; {- and more cities -}];
```

Such a centralized data file forces different documents to read from the same data source and prevents data inconsistency. Not only infoboxes but also computed documents like sorted lists of countries can be created using the information from the data file. Moreover, if the area of Tuvalu, for example, is assigned a string value, there will be a type error before rendering.

Writing Minipedia pages in E$_X$T. Writing Minipedia documents is enabled by the E$_X$T libraries shown in Fig. 4. Different features of E$_X$T are defined in different libraries, and we can import whatever libraries we want when writing a document. For Minipedia, we only need HTML-related libraries. There are two modes for document authoring in Minipedia: "doc-only" and "program". In the "doc-only" mode, with required libraries specified, documents are written directly in E$_X$T without any wrapping code in CP, as shown in Fig. 3. In the "program" mode, a document is created programmatically in a mixture of CP and E$_X$T code. This mode makes general-purpose computation easier to write in a document.

For example, a sorted table of the smallest countries by area demonstrates general-purpose computation in the "program" mode. The program begins with an **open** directive, which imports definitions from the specified libraries. We sort the array of countries imported from Database using a predefined function sort and then iterate it to generate E$_X$T code. As explained in Section 3.2, we use self-type annotations to inject dependencies on the library commands. The intersection type DocSig<T>&TableSig<T> allows E$_X$T commands from both compositional interfaces to be used in the trait doc:

```
open LibDoc; open LibTable; open Database;

sortedStates = sort states;
doc T = trait [self : DocSig<T>&TableSig<T>] ⇒ {
  table = letrec rows (i:Int) : T = if i == 0 then `\Trow[
      \Theader[No.]  \Theader[Country]    \Theader[Area (km^2)]
    ]` else let state = sortedStates!!(i-1) in `\rows(i-1) \Trow[
      \Tdata[\(i+1)] \Tdata[\(state.name)] \Tdata[\(state.area)]
    ]` in `\Tbody[ \rows(#sortedStates) ]`;
};
document = new doc @HTML , html , table;
document.table.html
```

Adding a table of contents (TOC). We have introduced how to write Minipedia pages using E$_X$T libraries, but now let us go back and see how we implement a E$_X$T library. The library of TOC adds a new command \Body[...], which prepends a TOC to the document body. The difficulty here is that HTML rendering depends on the TOC, whereas the TOC in turn depends on the HTML rendering of section (and sub^{n}section) titles. As we have demonstrated in Section 3.5, we can modularly tackle such complex dependencies by refining the interface types:

```
type ContentsSig<Element> = { Body : Element → Element };
type TOC = { toc : String };

contents = trait implements ContentsSig<TOC ⇒ HTML> ⇒ {
  [self]@(Body e).html = self.toc ++ e.html;
};
toc = trait implements TextualSig<HTML ⇒ TOC> & ContentsSig<TOC> ⇒ {
  (Comp          l r).toc = l.toc ++ r.toc;
  (Section         e).toc = listItem 0 e.html;
  (SubSection      e).toc = listItem 1 e.html;
  (SubSubSection e).toc = listItem 2 e.html;
  (Body            e).toc = "<ul>" ++ e.toc ++ "</ul>";
                  _.toc = "";
};
```

With `TOC` specified in `ContentsSig<TOC ⇒ HTML>` and the self-reference specified in `[self]`, we can call `self.toc` to generate a TOC when rendering HTML. Similarly, we can call `e.html` when generating the TOC since the interface type is refined. A minor remark is that `listItem` is an auxiliary function to generate an appropriate `<li>` wrapping for a given nesting level.

5.2 Fractals and Sharing

In the second application, we briefly introduce how to implement computational graphics like fractals in EχT with the help of linguistic reuse. Fractal drawing is all about repeating the same pattern. This drives us to exploit meta-language sharing to avoid redundant computation. We have implemented several well-known fractals, such as the Koch snowflake, the T-square, and the Sierpiński carpet. For the sake of simplicity, we take the Sierpiński carpet as an example:

```
fractal T C = trait [self : DocSig<T> & GraphicSig<T><C> & ColorSig<C> & Draw T C]
              implements Draw T C ⇒ {
  draw {..} =
    let center = Rect { x = x + width/3; y = y + height/3;
                        width = width/3; height = height/3; color = White } in
    if level == 0 then center else
      let w = width/3 in let h = height/3 in let l = level-1 in
      let shared = draw { x = x; y = y; width = w; height = h; level = l } in `\Group(id)[
        \shared                      \Translate{x=w;y=0}(shared)   \Translate{x=2*w;y=0}(shared)
        \Translate{x=0;y=h}(shared)  \center                       \Translate{x=2*w;y=h}(shared)
        \Translate{x=0;y=2*h}(shared) \Translate{x=w;y=2*h}(shared) \Translate{x=2*w;y=2*h}(shared)
      ]`
};
```

We make variables shared via the **let** expressions in CP. There are many uses in the code above but, among them, `shared` accelerates fractal generation the most. The variable `shared` is later used eight times to constitute the repeating patterns in the Sierpiński carpet. `\Translate` is a command declared in `GraphicSig` for geometric translations. With automatic linguistic reuse in EχT, we only need to generate a pattern once instead of repeating it eight times. Besides the efficiency in generating fractals, our implementation of `\Translate` makes use of the `<use>` element in SVG to avoid the exponential growth of output. Therefore, the duplication of SVG elements is also eliminated. These optimizations are available for free thanks to the linguistic reuse in compositional embeddings.

5.3 Customizing Charts

In the last application, we illustrate how line charts and bar charts are modularly rendered using external data. Charts can be rendered into a document using the E$_X$T language and its support for vector graphics. What is interesting about charts is that there are many alternative ways to present charts and customize them. The flexibility of the CP language, in terms of modularity and compositionality, is then very useful here. We will show how to adapt traditional object-oriented design patterns for compositional embeddings when modeling graphic components.

Charting stocks. Taking stock prices as an example, we have a separate file containing data from big companies, as well as a few configuration items for chart rendering. The configuration includes the choice between lines and bars, whether to show borders or legends, what labels to display, and so on. Serving as an infrastructure, a base chart is created with the drawing functions for the caption and axes:

```
type Base = { caption : HTML; xAxis : HTML; yAxis : HTML };
baseChart (data : [Data]) = trait implements Base ⇒ {
  caption = `...`;  xAxis = `...`;  yAxis = `...`;
};
```

The simplified code above shows a rough sketch, where a base trait takes data as its parameter and implements the base interface. However, it does not implement the primary rendering function. As we have two options to visualize stock prices, we leave the choice to the STRATEGY pattern [Gamma et al. 1995].

STRATEGY pattern. Since the configuration is unknown until run time, the rendering strategy cannot be hard-coded in the base trait. Instead, we define two strategy traits implementing the rendering, respectively for lines and bars. They constitute two variants of the base chart. It is not the programmer but the configurator that decides which variant of charts should be rendered. In the original STRATEGY pattern, a chart object delegates to a desired strategy object, to which its mutable field points. That mutable field can be changed from clients who use the chart. In CP, we just need to merge the base trait with the strategy we want, without the need for mutable references:

```
type Render = { render : HTML };
lineStrategy = trait [self : Base] implements Render ⇒ {
  render = let lines = ... in `\xAxis \yAxis \lines \caption`
};
barStrategy = trait [self : Base] implements Render ⇒ {
  render = let bars = ... in `\xAxis \yAxis \bars \caption`
};
chart = baseChart data , if config.line then lineStrategy else barStrategy;
```

After merging, chart is still a *reusable trait*. This is impossible in traditional object-oriented languages, like Java, because they lack a dynamic way to compose classes at run time. In contrast, such a dynamic trait composition is ubiquitous in CP. It is also easy to add a new strategy, like switching to pie charts, and merge the base trait with the new one instead. If a client forgets to pick any strategy, the type checker will notify them because the trait types before and after merging are different. Moreover, another advantage of our approach over the original strategy pattern is that the self-type annotations make methods in the base trait accessible to strategy traits. So caption, xAxis and yAxis are directly shared with strategies, requiring no extra effort to pass them as arguments when delegating. This remedies the "communication overhead between Strategy and Context" caused by the original STRATEGY pattern [Gamma et al. 1995].

Decorator pattern. Besides the Strategy pattern, the Decorator pattern [Gamma et al. 1995] is also adapted for CP. Since decorations are also determined by external configuration at run time, we need a modular way to add additional drawing processes to the chart trait dynamically. When multiple decorators are enabled, their functionalities should be added. In the original Decorator pattern, the decorator class should inherit the chart class and be instantiated with a reference to a chart object. The decorator will store the chart object as a field and forward all methods to it. In concrete decorators, the rendering function will be overridden to add decorations. The decorator pattern sounds complicated in traditional object-oriented programming, but the same goal can be achieved easily in CP using the *dynamic inheritance* provided by CP:

```
type Chart = Base & Render;
borderDecorator (chart : Trait<Chart>) = trait [self : Chart] implements Chart inherits chart ⇒ {
  override render = let sr = super.render in let border = ... in `\sr \border`
};
legendDecorator (chart : Trait<Chart>) = trait [self : Chart] implements Chart inherits chart ⇒ {
  override caption = let legends = ... in `\legends`
};
chart' = if config.border then borderDecorator chart else chart;
chart'' = if config.legend then legendDecorator chart' else chart';
```

A decorator takes a chart trait as the argument and creates another trait inheriting it dynamically. In the implementation, we can override any methods as usual, and static type safety is still guaranteed. It is worth noting that, in legendDecorator, only caption is overridden while render is just inherited. In the original Decorator pattern, render will never be conscious of the overridden caption since the decorator hands over control to chart after forwarding render. This is partly why Gamma et al. [1995] warn readers that "a decorator and its component aren't identical". However, in CP, chart.render can access the overridden caption through the late-bound self-reference. This solution makes our code clean and easy to maintain.

True delegation via trait composition. The chart application shows how other aspects of the modularity of CP are helpful to create highly customizable DSLs. We have shown how CP adapts and simplifies object-oriented design patterns. In general, we avoid complicated class hierarchies in the original versions and replace them with relatively simple trait composition. It showcases that we can get rid of verbose design patterns if a proper language feature fills in the gap. In both patterns, component adaptation is needed at run time, so delegation is at the heart of the challenges.

6 DISCUSSION AND RELATED WORK

In this section, we discuss how the ideas of Compositional Programming and compositional embeddings can be more generally applied to other languages as well as related work.

6.1 Encoding Compositional Embeddings in Other Languages

There are three key features that enable compositional embeddings in CP: *compositional interfaces*, *self-references*, and *nested composition*. All the three features can be encoded, with some effort and boilerplate code, in other programming languages, such as Haskell, Scala, or even Java.

Compositional interfaces can be encoded indirectly using Haskell type classes, Scala traits, or Java interfaces. For example, the compositional interface on the left corresponds to the Scala trait for generalized object algebras [Oliveira et al. 2013] on the right:

```
type HoferSig<Region> = {                    trait HoferSig[In,Out] {
  Univ  : Region;                              def Univ  : Out
  Empty : Region;                              def Empty : Out
  Scale : Vector → Region → Region;            def Scale : (Vector,In) ⇒ Out
};                                           }
```

Here we distinguish between types used in input positions and types used in output positions. This distinction is the key to the mechanism of dependency injection in CP [Zhang et al. 2021].

Self-references are also essential to dependency injection in CP. Complex interpretations may depend on self dependencies other than child dependencies [Zhang et al. 2021], which has occurred in the example of the text interpretation:

```
[self]@(Translate v a).text = "a translated region of size " ++ toString self.size;
```

The method pattern above refers to self.size instead of a.size. This feature can be modeled via open recursion, or we may reuse the available self-references in some object-oriented languages like Scala.

Nested composition can be expressed in existing languages by writing explicit composition operators. For example, a merge operator for composing Eval and Count will have such a type:

```
merge : HoferSig<Eval> → HoferSig<Count> → HoferSig<Eval&Count>
```

In essence, a merge operator takes two interpretations and creates a single merged one. Meanwhile, this is where most boilerplate code would come from since we need to write a new implementation for every new signature of the merge operator. A workaround is to generate this kind of boilerplate code using meta-programming.

A detailed description of encoding the key features in Scala can be found in work by Oliveira et al. [2013]. Inspired by Bi et al. [2019], we also demonstrate a Haskell encoding in Appendix B. With the help of quite a few GHC language extensions, dependent interpretations are defined modularly in a tagless-final style, and their composition is directed by type classes. However, self-references are still not supported in that Haskell encoding, so we cannot directly address self dependencies like self.size. Moreover, the boilerplate code of smart constructors is needed for every language construct in the region DSL, and explicit projections are inevitable when writing interpretations. Compositional embeddings are more compact and cleaner in CP thanks to its language-level support for all of the three key features.

6.2 Related Work

Modular embeddings. There are quite a few existing approaches to modular embeddings, which essentially exploit solutions to the *Expression Problem* [Wadler 1998] in existing languages. Such approaches include, for example, *tagless-final embeddings* [Carette et al. 2009] and *data types à la carte* [Swierstra 2008] in functional programming, *polymorphic embeddings* [Hofer et al. 2008] and *object algebras* [Oliveira and Cook 2012] in object-oriented languages, and solutions to the *expression compatibility problem* [Haeri and Keir 2019; Haeri and Schupp 2016], just to name a few. Section 3.6 already compares those approaches with compositional embeddings in detail. In essence, the lack of sufficient programming language support for modularity makes it hard to model modular dependencies in those approaches. Compositional embeddings offer an elegant solution to many issues by exploiting the support for Compositional Programming in the CP language.

Hybrid embeddings. Svenningsson and Axelsson [2015] propose combining shallow and deep embeddings by translating a shallowly embedded interface to a deeply embedded core language. In this way, language interfaces can be shallowly extended, and multiple interpretations are possible in the deep core. In addition, some features in the host language can be exploited in the shallow

part, which is called *deep linguistic reuse* in Scala-Virtualized [Rompf et al. 2012]. Jovanović et al. [2014] further propose an automatic translation between shallow and deep embeddings using Scala macros instead of a manual translation. Their Yin-Yang framework targets lightweight modular staging [Rompf and Odersky 2010] as the deep embedding backend but completely conceals internal encodings from the users. A similar but slightly different approach called *implicit staging* is proposed by Scherr and Chiba [2014]. Implicit staging extracts an intermediate representation from a shallowly embedded DSL and reintegrates it after some transformations. Their research prototype works in Java through load-time reflection. Compared to such hybrid approaches, compositional embeddings do not require the use of two different embeddings and the translations between them while supporting most features from both shallow and deep embeddings.

Generative programming. As stated in Section 4, ExT performs a lightweight desugaring during parsing to generate compositionally embedded fragments in CP. This is an *ad-hoc* generative technique. Some generative approaches in other languages, such as Racket macros [Ballantyne et al. 2020] and Template Haskell [Sheard and Peyton Jones 2002], provide more flexible mechanisms compared to ours. Nevertheless, in this work, our goal is not to develop generative programming in CP. Instead, we try to keep the DSL as close to the underlying compositional embeddings as possible so that there is an obvious one-to-one mapping between ExT and CP. Rather than syntactic extensibility (i.e. the ability to extend the syntax of the source language), our focus is on semantic support for DSLs. Our approach is different from some generative frameworks like EVF [Zhang and Oliveira 2017] and Castor [Zhang and Oliveira 2020], which generate extensible visitors from annotations via meta-programming. Although they also enable modular and extensible programming language components, their code uses various non-standard annotations, whose semantics may be unclear to programmers. What is worse, type checking is delayed in the aforementioned generative frameworks so error messages are reported in terms of the generated code. This can be quite confusing without detailed knowledge of how the generated code works. In contrast, in our implementation, type checking is done in the source language, and error messages are reported in terms of the original ExT code.

Language workbenches. JetBrains MPS [Fowler 2005a], Xtext [Efftinge and Völter 2006], Monti-Core [Krahn et al. 2010], and Spoofax [Kats and Visser 2010], just to name a few, are popular *language workbenches* [Fowler 2005b] that enable easier DSL development. Erdweg et al. [2015] have done a comprehensive evaluation about them. Notably, in addition to textual DSLs, JetBrains MPS supports tables, mathematical symbols, and diagrammatic notations, which significantly enhance user experience. Modern language workbenches also provide editor support for user-defined DSLs. Similar to compositional embeddings, they often allow language components to be modularly composed, but their focus is mainly on syntactic extensibility instead of semantic extensibility. In contrast to direct embeddings into host languages, external tools are required by language workbenches to generate DSL implementations from specifications. This is the essential difference from compositional embeddings, where composition is directly built in the programming language.

Document DSLs. There are quite a few existing DSLs for document authoring. LaTeX is one of the most commonly used document languages, and ExT's syntax is inspired by it. In contrast to ExT, LaTeX is an external DSL and does not have a static type system. LaTeX supports arbitrary computation, but it is not easy to write meta-programs over the AST of LaTeX. Moreover, LaTeX is not intended to render web documents, which is an important goal of ExT. *Wikitext* [MediaWiki 2003] is widely used across wiki websites, including Wikipedia and Fandom. Wikitext has some design limitations like the absence of type safety, data consistency, and general-purpose computation, which have been elaborated in Section 5.1. *Skribe* [Gallesio and Serrano 2005] is a document DSL

built on top of the Scheme programming language and extends Scheme's syntax from S-expressions to *Sk-expressions* for easier text writing. Skribe makes use of Scheme macros and functions to construct documents. This idea is inherited and popularized by *Scribble* [Flatt et al. 2009] in the Racket community, and DrRacket offers very good tool support for Scribble. Regarding syntax, a new *@-notation* is designed for Scribble, which is desugared to normal S-expressions. E$_X$T's syntax is similar to Scribble's but is more flexible. Based on the infrastructure provided by Racket, Scribble supports arbitrary computation and linguistic reuse like E$_X$T. However, Scribble has a fixed document structure [Flatt and Barzilay 2009], and its rendering function performs conventional pattern matching on it. Thus, it is not easy to extend their core document constructs and functions without modifying the original source code. Furthermore, there is no static typing in Scribble, though it uses Racket's *contract* system to perform run-time type checking.

7 CONCLUSION

We have presented compositional embeddings and shown their advantages compared to existing embedding techniques in terms of modularity and compositionality. CP's support for dependency injection, pattern matching, and nested composition makes *seemingly* non-compositional interpretations feasible in compositional embeddings. Thus, dependent interpretations and complex transformations can be modularly defined. These virtues of compositional embeddings lay down the foundation for the E$_X$T DSL. The viability of compositionally embedded E$_X$T has been evaluated with three applications: *Minipedia* shows its extensibility, type safety, data consistency, and ability to modularly handle dependencies; *fractals* show that general-purpose computation and linguistic reuse are both available; finally, *charts* show a simpler approach to delegation compared to object-oriented design patterns. Last but not least, an in-browser implementation of CP and E$_X$T is publicly available, and all code in the present paper can be type checked and run online.

Compared to other techniques, compositional embeddings require dedicated programming language support. While we are working hard on improving CP and its support for Compositional Programming, we believe that an equally important outcome of our work is to inform current and future language designers about the benefits and applications of Compositional Programming and compositional embeddings. We hope that the results of our work will lead to other programming languages considering extensions that support Compositional Programming and compositional embeddings as well.

Future work. Since CP and E$_X$T are research prototypes, they are not yet production-ready. Our implementation of CP is an in-browser interpreter, which is not so fast as industrial-strength language implementations. We expect to improve its dynamic semantics and write a compiler for CP, targeting either JavaScript or WebAssembly.

Currently, E$_X$T is evaluated with some relatively small-scale applications. To show that E$_X$T can provide a realistic alternative to established wiki languages, we need to deploy E$_X$T on a much larger scale. At the moment, Minipedia is a toy compared to ultra-large wiki websites, such as Wikipedia. We hope to attract more people to try E$_X$T on our website and send feedback to us. We also need pragmatic advice from wiki experts to make E$_X$T a realistic alternative to Wikitext or other wiki markup languages.

ACKNOWLEDGMENTS

REFERENCES

Dean Allen. 2002. Textile Markup Language Documentation. https://textile-lang.com

Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer-Verlag. https://doi.org/10.1007/978-3-642-37521-7

Michael Ballantyne, Alexis King, and Matthias Felleisen. 2020. Macros for Domain-Specific Languages. In *OOPSLA.* https://doi.org/10.1145/3428297

Xuan Bi and Bruno C. d. S. Oliveira. 2018. Typed First-Class Traits. In *ECOOP.* https://doi.org/10.4230/LIPIcs.ECOOP.2018.9

Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2018. The Essence of Nested Composition. In *ECOOP.* https://doi.org/10.4230/LIPIcs.ECOOP.2018.22

Xuan Bi, Ningning Xie, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. Distributive Disjoint Polymorphism for Compositional Programming. In *ESOP.* https://doi.org/10.1007/978-3-030-17184-1_14

Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel. 1992. Experience with Embedding Hardware Description Languages in HOL. In *TPCD.*

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). https://doi.org/10.1017/S0956796809007205

Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP.* https://doi.org/10.1145/351240.351266

Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. 2006. Traits: A Mechanism for Fine-grained Reuse. *ACM Trans. Program. Lang. Syst.* 28, 2 (2006). https://doi.org/10.1145/1119479.1119483

Jana Dunfield. 2014. Elaborating Intersection and Union Types. *J. Funct. Program.* 24, 2–3 (2014). https://doi.org/10.1017/S0956796813000270

Sven Efftinge and Markus Völter. 2006. *oAW xText: A Framework for Textual DSLs.* Technical Report.

Burak Emir, Martin Odersky, and John Williams. 2007. Matching Objects with Patterns. In *ECOOP.* https://doi.org/10.1007/978-3-540-73589-2_14

Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2015. Evaluating and Comparing Language Workbenches: Existing Results and Benchmarks for the Future. *Comput. Lang. Syst. Struct.* 44 (2015). https://doi.org/10.1016/j.cl.2015.08.007

Erik Ernst. 2001. Family Polymorphism. In *ECOOP.* https://doi.org/10.1007/3-540-45337-7_17

Erik Ernst, Klaus Ostermann, and William R. Cook. 2006. A Virtual Class Calculus. In *POPL.* https://doi.org/10.1145/1111037.1111062

Matthew Flatt and Eli Barzilay. 2009. Low-Level Scribble API: Structures And Processing. https://docs.racket-lang.org/scribble/core.html

Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. 2009. Scribble: Closing the Book on Ad Hoc Documentation Tools. In *ICFP.* https://doi.org/10.1145/1596550.1596569

Martin Fowler. 2005a. A Language Workbench in Action - MPS. https://martinfowler.com/articles/mpsAgree.html

Martin Fowler. 2005b. Language Workbenches: The Killer-App for Domain Specific Languages? https://martinfowler.com/articles/languageWorkbench.html

Erick Gallesio and Manuel Serrano. 2005. Skribe: a Functional Authoring Language. *J. Funct. Program.* 15, 5 (2005). https://doi.org/10.1017/S0956796805005575

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional.

Jeremy Gibbons and Nicolas Wu. 2014. Folding Domain-Specific Languages: Deep and Shallow Embeddings (Functional Pearl). In *ICFP.* https://doi.org/10.1145/2628136.2628138

Andy Gill. 2009. Type-Safe Observable Sharing in Haskell. In *Haskell@ICFP.* https://doi.org/10.1145/1596638.1596653

David Goodger. 2002. reStructuredText: Markup Syntax and Parser Component of Docutils. https://docutils.sourceforge.io/rst.html

Seyed Hossein Haeri and Paul Keir. 2019. Solving the Expression Problem in C++, à la LMS. In *ICTAC.* https://doi.org/10.1007/978-3-030-32505-3_20

Seyed Hossein Haeri and Sibylle Schupp. 2016. Expression Compatibility Problem. In *SCSS.* https://doi.org/10.29007/xlbn

Ralf Hinze. 2006. Generics for the Masses. *J. Funct. Program.* 16, 4–5 (2006). https://doi.org/10.1017/S0956796806006022

Christian Hofer and Klaus Ostermann. 2010. Modular Domain-Specific Language Components in Scala. In *GPCE.* https://doi.org/10.1145/1868294.1868307

Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. 2008. Polymorphic Embedding of DSLs. In *GPCE.* https://doi.org/10.1145/1449913.1449935

Paul Hudak. 1998. Modular Domain Specific Languages and Tools. In *ICSR.* https://doi.org/10.1109/ICSR.1998.685738

John Hughes. 1995. The Design of a Pretty-printing Library. In *AFP*. https://doi.org/10.1007/3-540-59451-5_3

Vojin Jovanović, Amir Shaikhha, Sandro Stucki, Vladimir Nikolaev, Christoph Koch, and Martin Odersky. 2014. Yin-Yang: Concealing the Deep Embedding of DSLs. In *GPCE*. https://doi.org/10.1145/2658761.2658771

Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *OOPSLA*. https://doi.org/10.1145/1869459.1869497

Oleg Kiselyov. 2010. Typed Tagless Final Interpreters. In *SSGIP*. https://doi.org/10.1007/978-3-642-32202-0_3

Oleg Kiselyov. 2011. Implementing Explicit and Finding Implicit Sharing in Embedded DSLs. In *DSL*. https://doi.org/10.4204/EPTCS.66.11

Holger Krahn, Bernhard Rumpe, and Steven Völkel. 2010. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *Int. J. Softw. Tools Technol. Transf.* 12, 5 (2010). https://doi.org/10.1007/S10009-010-0142-1

Shriram Krishnamurthi. 2001. *Linguistic Reuse.* Ph. D. Dissertation. Rice University.

Daan Leijen and Erik Meijer. 2001. *Parsec: Direct Style Monadic Parser Combinators for the Real World.* Technical Report.

John MacFarlane. 2014. CommonMark Spec. https://spec.commonmark.org

Ole Lehrmann Madsen and Birger Møller-Pedersen. 1989. Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. In *OOPSLA*. https://doi.org/10.1145/74877.74919

MediaWiki. 2003. Wikitext. https://www.mediawiki.org/wiki/Wikitext

MediaWiki. 2011. Parsoid. https://www.mediawiki.org/wiki/Parsoid

Bruno C. d. S. Oliveira. 2009. Modular Visitor Components: A Practical Solution to the Expression Families Problem. In *ECOOP*. https://doi.org/10.1007/978-3-642-03013-0_13

Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the Masses: Practical Extensibility with Object Algebras. In *ECOOP*. https://doi.org/10.1007/978-3-642-31057-7_2

Bruno C. d. S. Oliveira, Ralf Hinze, and Andres Löh. 2006. Extensible and Modular Generics for the Masses. In *TFP*.

Bruno C. d. S. Oliveira and Andres Löh. 2013. Abstract Syntax Graphs for Domain Specific Languages. In *PEPM@POPL*. https://doi.org/10.1145/2426890.2426909

Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. 2016. Disjoint Intersection Types. In *ICFP*. https://doi.org/10.1145/2951913.2951945

Bruno C. d. S. Oliveira, Tijs van der Storm, Alex Loh, and William R. Cook. 2013. Feature-Oriented Programming with Object Algebras. In *ECOOP*. https://doi.org/10.1007/978-3-642-39038-8_2

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple Unification-Based Type Inference for GADTs. In *ICFP*. https://doi.org/10.1145/1160074.1159811

Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. 2012. Scala-Virtualized: Linguistic Reuse for Deep Embeddings. *High. Order Symb. Comput.* 25, 1 (2012). https://doi.org/10.1007/S10990-013-9096-9

Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *GPCE*. https://doi.org/10.1145/1868294.1868314

Maximilian Scherr and Shigeru Chiba. 2014. Implicit Staging of EDSL Expressions: A Bridge between Shallow and Deep Embedding. In *ECOOP*. https://doi.org/10.1007/978-3-662-44202-9_16

Tim Sheard and Simon Peyton Jones. 2002. Template Meta-programming for Haskell. In *Haskell@ICFP*. https://doi.org/10.1145/581690.581691

Josef Svenningsson and Emil Axelsson. 2015. Combining Deep and Shallow Embedding of Domain-Specific Languages. *Comput. Lang. Syst. Struct.* 44 (2015). https://doi.org/10.1016/j.cl.2015.07.003

Wouter Swierstra. 2008. Data Types à la Carte (Functional Pearl). *J. Funct. Program.* 18, 4 (2008). https://doi.org/10.1017/S0956796808006758

Philip Wadler. 1998. The Expression Problem. Posted on the Java Genericity mailing list. https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt

Philip Wadler. 2003. A Prettier Printer. In *The Fun of Programming*. https://doi.org/10.1007/978-1-349-91518-7_11

Weixin Zhang and Bruno C. d. S. Oliveira. 2017. EVF: An Extensible and Expressive Visitor Framework for Programming Language Reuse. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2017.29

Weixin Zhang and Bruno C. d. S. Oliveira. 2019. Shallow EDSLs and Object-Oriented Programming: Beyond Simple Compositionality. *Art Sci. Eng. Program.* 3, 3 (2019). https://doi.org/10.22152/programming-journal.org/2019/3/10

Weixin Zhang and Bruno C. d. S. Oliveira. 2020. Castor: Programming with Extensible Generative Visitors. *Sci. Comput. Program.* 193 (2020). https://doi.org/10.1016/j.scico.2020.102449

Weixin Zhang, Yaozhu Sun, and Bruno C. d. S. Oliveira. 2021. Compositional Programming. *ACM Trans. Program. Lang. Syst.* 43, 3 (2021). https://doi.org/10.1145/3460228