CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Thesis for the Degree of Licentiate of Engineering

Laying Tiles Ornamentally

An approach to structuring container traversals

Nikita Frolov

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Gothenburg, Sweden
2016

Laying Tiles Ornamentally
An approach to structuring container traversals

Technical Report 161L
ISSN 1652-876X
Department of Computer Science and Engineering

Chalmers University of Technology and
University of Gothenburg

SE-412 96 Gothenburg, Sweden
Telephone +46 (0)31-772 1000

Printed at Reproservice, Chalmers University of Technology
Gothenburg, Sweden, 2016

Abstract

Having hardware more capable of parallel execution means that more program scheduling decisions have to be taken to utilize that hardware efficiently. To this end, compilers implement coarse-grained loop transformations in addition to traditionally used fine-grained instruction reordering. Implementors of embedded domain specific languages have to face a difficult choice: to translate operations on collections to a low-level language naively hoping that its optimizer will do the job, or to implement their own optimizer as a part of the EDSL.

We turn ourselves to the concept of loop tiling from the imperative world and find its equivalent for recursive functions. We show the construction of a *tiled* functorial map over containers that can be naively translated to a corresponding nested loop.

We illustrate the connection between *untiled* and tiled functorial maps by means of a type-theoretic notion of *algebraic ornament*. This approach produces an family of container traversals indexed by *tile sizes* and serves as a basis of a proof that untiled and tiled functorial maps have the same semantics.

We evaluate our approach by designing a language of tree traversals as a DSL embedded into Haskell which compiles into C code. We use this language to implement tiled and untiled tree traversals which we benchmark under varying choices of tile sizes and shapes of input trees. For some tree shapes, we show that a tiled tree traversal can be up to 50% faster than an untiled one under a good choice of the tile size.

Acknowledgements

First and foremost, I would like to thank my supervisor John Hughes for his ability to look at problems from unexpected directions. I would like to thank my co-supervisor Koen Claessen for his practical insights and his contagious hacker spirit. I would also like to thank Prof. Sally McKee for persuading me to begin my doctorate and for help with structuring my writing. Last but not least, I would like to thank Guillaume without whom this journey would go in an entirely different direction.

Contents

Chapter 1

Introduction

Functional programming languages equip the programmer with a high level of expressiveness but that expressiveness comes at a price. This price can be too high in a resource-constrained environment, such as embedded hardware. *Pure* languages often require garbage collection in run-time. *Lazy* languages are prone to space leaks. To combine expressiveness and resource awareness, one may design a domain specific language embedded into a general purpose functional language.

An example of such an embedded DSL is Feldspar [6]. Its backend takes all memory allocation decisions during compile-time, producing a C program that does not rely on a complicated run-time environment. A number of memory usage optimizations happen in Feldspar's frontend, for example, *fusion* [50].

This thesis extends memory management options provided by an EDSL further. We introduce a technique known from cache use optimization widespread in imperative language compilers, *tiling*, to functional EDSLs. It allows the programmer to give hints to the compiler about locality of access in data structure traversals.

1.1 Scaling the memory wall

Functional programming using recursive combinators naturally exposes parallelism [17, 37, 16] but the extent to which this parallelism can be exploited depends on the resources of the underlying machine. The latter phenomenon is known as the *memory wall* [54]. For instance, the number of execution units that can be used productively inherently depends on how fast the memory can supply those units with data. Improving the efficiency of memory accesses in a program can thus have a large impact on the program's performance.

Caching is a common approach to efficient utilization of memory bandwidth. Modern computer architectures put a *hierarchy* of caches that differ in size and speed between the processor and the memory. The

fastest caches are the ones that are closer to the processor but they are also the smallest. On the contrary, the largest and slowest caches are closer to the memory.

There are two major approaches to taking advantage of a cache hierarchy. Both build on exploiting *locality of reference*, that is, *temporal* locality or *spatial* locality. Temporal locality means repeating reference of the same data within a short period of time (for example, an accumulator value being updated many times). Spatial locality means reference of data elements that are allocated close to each other in memory (for example, neighboring elements in a sequentially allocated array).

The first approach is to design program transformations to exploit temporal locality by changing the order of memory accesses and spatial locality by changing the layout of data in memory. This approach is known as *cache awareness* because such transformations often require knowledge of cache hierarchy parameters such as the number of cache levels, their size, cache line size, associativity or replacement and coherence policies. The second approach is to design algorithms which perform well without such explicit knowledge. This approach is known as *cache obliviousness*.

The research on principles of caching goes back to performance studies of databases and virtual memory swapping. In those settings, RAMs can be seen as "fast caches" and hard disks as "slow memories". The scope of research on optimal use of cache hierarchies has broadened as the performance gap between processors and memories has increased. This the reason why a large body of recent studies of cache optimization is done for parallel systems, such as multicores and GPUs. [13, 23, 47]

1.2 Cache-aware loop transformations

Adopting a programming style that exploits data locality is one way to utilize the memory hierarchy but this is a tedious and error-prone task. To relieve the programmer from the burden of tuning programs to a particular hardware platform, compilers can implement various program transformations. A number of optimization techniques have been developed to exploit cache locality. Loop-intensive computation kernels can be *tiled*, that is, the loops being fused, fissioned, permuted, or otherwise partitioned [44].

Powerful frameworks for tiling imperative programs are based on *polyhedral* transformations [25, 12]. They bring an elegant approach to tiling by applying affine transforms to imperative programs that are tedious to extract data dependencies from. They are based on identifying four

aspects of a program a loop transformation modifies: iteration space, schedules of statements, array subscripts and data layout. Each aspect is given a matrix representation, so modifications can be expressed as affine (linear) transformations. This makes it possible to represent each loop transformation as a composition of aspect modification and to compose loop transformations themselves.

Translation of programs to the polyhedral representation and back is an area of research in itself. Bastoul [7] develops polyhedral encodings for loop strip-mining and partitioning. He also shows how to minimize the overhead of generated control code by reducing code hoisting which improves utilization of the instruction cache.

1.3 Cache-aware data layout transformations

Although this thesis studies transformations of traversal order, a large literature exists on optimization of access to tree-like pointer-linked structures by changing their layout in memory.

A technique named *clustering* has been developed for packing data structure elements that are likely to be accessed together into single cache blocks during memory allocation. Clustering can be applied both to flat [18] and pointer-based [19] structures, even when access patterns are presumed to be random. It can be further extended to *coloring*, that is, placing data into separate cache regions to avoid associativity conflicts.

Studies of cache behavior of functional programs go as early as [39]. Koopman et al. design a graph reduction abstract machine that represents combinator graphs as self-modifying threaded programs. They study the impact of write caching policy under use of a garbage collector, with *write-no-allocate* policy (which bypasses the cache on writes) being common at the time due to hardware limitations being found inefficient as compared with *write-allocate* policy (which stores the new data in the cache before performing a write to the memory). They also study the improvement of spatial locality resulting from designing the abstract machine to allocate graph nodes on the sequential addresses in the heap.

In [19], Chilimbi et al. improve locality of access with cache-conscious reorganization and cache-conscious allocation. The former approach uses topological properties of tree data structures to specialize their memory layout to a particular memory hierarchy. It copies a sparsely allocated pointer-linked tree-like data structure into a contiguous block of memory while partitioning it into subtrees which are laid out linearly. The applicability of this optimization is limited by existence of external pointers into

the middle of the data structure. The latter approach modifies the heap allocator to produce an optimal data structure layout without copying.

Layout optimization work is not limited to compiler transformations. Techniques to use a generational garbage collector as a memory reorganization tool that produces cache-conscious data layouts have been studied. [20, 30] Yates and Scott [56] improve the performance of the software transactional memory implementation in GHC by changing the internal representation of *transactional variables* (variables that cannot be accessed outside of the transaction scope) removing the need for pointer dereference in many cases.

1.4 Cache-oblivious algorithms

A cache-oblivious algorithm is an algorithm that uses the cache optimally in the asymptotic sense without taking the parameters of the cache (its size, the length of cache lines, associativity, replacement and coherence policies etc) as explicit parameters. Many divide-and-conquer algorithms turn out to be cache-oblivious due to the fact that they divide a problem into smaller subproblems, eventually reaching a subproblem of a size that fits into the cache.

A number of algorithms for a number of common problems, such as matrix multiplication and mergesort, have been shown to be have cache-oblivious implementations. They have been shown to have the same asymptotic cache complexity as cache-aware algorithms. However, in practice, a proper sizing of recursion base cases is necessary to avoid the merge step overhead. Also, cache obliviousness of algorithms *composed* from cache oblivious algorithms cannot be established if the algorithms being composed rely on different data structure layouts. Moreover, it has been shown that some divide-and-conquer algorithms do not possess the cache obliviousness property [15]

In [9], Blelloch et al. describe a cost model for cache miss rate in a multicore processing environment. A multicore processor has at least two levels of cache: a small private cache for each core and a larger shared cache. A broad class of divide-and-conquer algorithm is shown to achieve good cache performance in the presented model.

In [10], Blelloch et al., suggest that a dynamic (run-time) scheduler can be more efficient in exploiting the cache than a static (compile-time) scheduler. Based on a modified cost model from [9], their scheduler works on parallel programs annotated with space requirements. Each recursive subcomputation in an annotated program is a schedulable *task*. The

scheduler is aware of the cache hierarchy and identifies for each task a cache level closest to it in size. Once assigned, the tasks cannot migrate between cache levels. This approach is shown to have a cache miss rate linear in the problem size, cache size and cache line size in the used cost model.

In [11], Blelloch and Harper introduce and study a cost model for analyzing the memory efficiency of programs written in a functional language. The model is defined as an operational semantics for the call-by-value lambda calculus with an explicit store. The store is organized into three parts: a memory, an allocation cache and a read cache. Data in the allocation cache is written back to the memory, and the read cache is filled from the memory. The model does not cover the behavior of the garbage collector. They demonstrate that many algorithms over lists and trees are efficient in this model without explicit specification of data layouts. Then, they prove that the asymptotic bounds of cache miss rate in their model map to the bounds of cache miss rate in the ideal cache model [24].

1.5 Problem statement and contributions

The problem of scheduling imperative programs over arrays for optimal cache performance is well-studied. There exists a large literature on corresponding compiler analyses and transformations. Its findings have been made part of the state-of-art industrial compilers. In this thesis, we describe an approach to cache-optimizing transformation of programs written in pure functional languages that is based on their algebraic properties. In particular, our contributions are as follows:

- Using a well-known representation of algebraic data types (*containers*) as *zippers* (reviewed in Sec .2.2), we demonstrate that the order of traversal over containers can be changed to achieve the desired locality of access. (Section 3.2)
- We define a subclass of traversal order transformations that allows the programmer to provide a *tile shape* of the traversal, that is, a shape of a subcontainer that fits into a cache level. (Section 3.3)
- Using *algebraic ornamentation* (reviewed in Sec .2.2), we show that for each traversal state (position in a container), there exists a position in a subcontainer (tile). (Section 3.4)
- Given an ornamentation of containers with tile positions, we show that by *function lifting across ornaments* a container traversal can

be transformed into a nested one (a *mutumorphism*). (Section 3.5)

- We evaluate the proposed transformation by designing a language of binary tree traversals which compiles to C. (Chapter 4) We use this language to implement and benchmark tiled and untiled tree traversals. The benchmark results illustrate how speedups change under choice of the tiling parameter and the shape and size of trees used as input. Speedups reach 50% under a well-chosen tiling parameter.

Chapter 3 is based on the paper submitted by the author for review to the 28th Symposium on Implementation and Application of Functional Languages (IFL2016) under the title "Laying Tiles Ornamentally". Additionally, we review the necessary background of containers, zippers and ornaments in Chapter 2 and related work in Chapter 5. We pinpoint some possible extensions of the presented work in Chapter 6.

Chapter 2

Background

This thesis relies on a number of existing constructions in functional programming and type theory. We review them in the current chapter. The presentation of this material is largely based on theses by Abbott [1] and Dagand [22]. A number of inspirational ideas come from McBride [42] and Yakushev [55].

2.1 Containers

Algebraic datatypes (ADTs) are a commonly used abstraction in functional languages. ADTs allow the programmer to define his own datatypes in terms of sum types, product types, built-in types and type variables. The programmer can write recursive functions over the values of types defined as ADTs using *pattern matching*. Moreover, recursive functions over ADTs can be defined in a more abstract way using *recursion schemes* [28]. One can transform programs expressed in terms of recursion schemes and prove their properties using a powerful framework of *program calculation*. [8]

The algebraic laws that hold about ADTs and recursion schemes are well-studied in the context of category theory. Categorically, ADTs are represented as *polynomial functors*. Given a category $\mathbb{C}$ with finite products and finite distributive coproducts, a polynomial functor $\mathbb{C} \to \mathbb{C}$ is constructed inductively from the identity functor $\mathrm{id}_{\mathbb{C}}$ (representing type variable positions), constant functors $K_{\mathbb{C}}$ (representing built-in types), product functor $\times$ and coproduct functor $+$. Each polynomial functor can be normalized to the form

$$PX = \Sigma n : \mathbb{N}.A_n \times X^n$$

where the arity of the sum defines the degree of a polynomial. The constant coefficients A_n are represented by types with a finite number of inhabitants, including the empty type (which represents the coefficient 0).

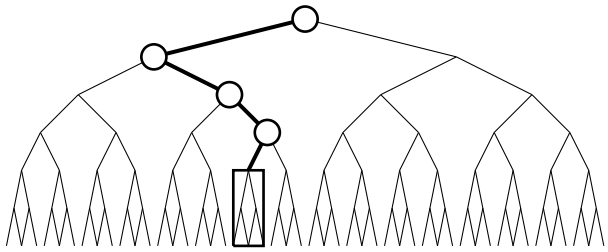


Figure 2.1. A focused subtree in a zipper.

All polynomials can be normalized to this form. [26] This presentation also corresponds to *shapely types* [32].

Container functors (or simply *containers*) are a generalization of polynomial functors

$$CX = \Sigma_A X^B$$

where A is a set of *shapes* of a container type, and $B(a : A) \to X$ is a *position function* mapping positions in a container of a certain shape to values at those positions. In type theory, containers correspond to indexed families $\mathbb{C}^I \to \mathbb{C}^I$ where indices represent container shapes.

2.2 Zippers

In a functional language, morphisms of container types are defined equationally, abstracting away the traversal state. To reason about the latter, an explicit representation is needed. The *zipper* [31] is one such representation. It represents a value of a tree-like type (which containers are) as a pair of the focused subtree and a list of *one-hole contexts*. The list of one-hole contexts can be seen as a path from the tree root to the current position in the tree.

The one-hole context is essentially a node in the process of being traversed where one child is replaced by a tag telling its position among its siblings. For example, given a binary tree type $Tree\ A = \mu X.1 + A \times X \times X$, the corresponding one-hole context type is $A \times 1 \times X + A \times X \times 1$ (where the unit values mark the position of holes) or, equivalently, $2 \times A \times Tree\ A$. The type with two inhabitants 2 is the tag denoting that there are only two choices in placing the focused subtree next to its sibling: to the left or to the right.

An example of a focused tree in a context is given in Fig. 2.1.

One can traverse the zipper by stepping down into a child of the current node or by stepping up to the parent (if there is one). When stepping into a child of a node, that child becomes the new focused subtree, and its siblings and the node label are packed into a one-hole context which becomes the new head of the list of contexts. When stepping up to a parent, a new focused subtree is constructed.

2.2.1 One-hole contexts as derivatives of containers

It was observed by McBride in [41] that the types of one-hole contexts can be computed from polynomial types by applying syntactic rules of partial differentiation known from calculus:

$$\delta_X K = 0 \text{ (if } X \text{ is not free in } K) \text{ where } 0 \text{ is the empty type}$$

$$\delta_X K \times X = K \text{ (if } X \text{ is not free in } K)$$

$$\delta_X (F + G) \cong \delta_X F + \delta_X G$$

$$\delta_X (F \times G) \cong \delta_X F \times G + F \times \delta_X G$$

In [41], McBride conjectures a metaprogram implementing this procedure generically. In [55], Yakushev et al. use then-recently implemented GHC extensions to avoid the need for a metalanguage to implement datatype differentiation generically. First, they define a syntax of datatypes by giving a higher-order functor for each type former, such as constants, recursive variables, variable binders, products and sums. Then they declare a type-level function (a *type family* in GHC parlance) that takes syntactic descriptions of datatypes to Haskell datatypes. An instance of the Haskell type class of higher-order functors implements a functorial map for datatypes declared with this syntax. Another type-level function defines generically the type of initial algebras of the declared datatypes.

At last, Yakushev et al. use the *indexed fixed point* technique [51] to declare the zipper type as a family of mutually recursive types: a tree, a list of one-hole contexts and a product of trees and lists. This family is indexed with values of type with as many inhabitants as there are datatypes in the mutual recursion. We use a similar approach to declare zippers in Section 3.4.

2.2.2 State of traversal as dissection of containers

A zipper represents a position in a data structure but being able to reason about position alone is not enough to reason about traversal orders. To complete the representation of traversal state, one needs to distinguish between visited and unvisited positions in a container. In [42], McBride generalizes the derivative operation on containers to *dissection*. This new operation takes container types to types of one-hole contexts where the elements preceding the hole in a traversal differ in type from the elements succeeding it.

McBride uses dissection to construct in-order left-to-right traversals (a map and a fold) of the type of binary trees. He also remarks that making traversal state first-class data allows one to obtain a finer control over the traversal order. Changing traversal order becomes equivalent to writing a program that manipulates the traversal state explicitly.

2.3 Ornaments

Treatment of datatypes as data in generic programming brings naturally the idea of writing programs that manipulate datatype descriptions. *Ornamentation* is an umbrella term for operations that produce new datatypes from existing ones. We call a type before applying an ornament *base*, and after *ornamented*, or simply an *ornament*, if it is clear from the context what the base type is. In this thesis, we use a representation of ornaments first introduced by Dagand and McBride in [21] and described in detail in Dagand's thesis [22] [1].

While Dagand's thesis and the accompanying Agda library of ornaments constitute the largest single body of work on ornaments , there are others. The work on McBride-inspired ornaments by Ko and Gibbons [38] is also of note, but we do not rely on their presentation in this thesis. Williams et al. [53] have designed an OCaml-like language with syntax for definition of ornaments, ornament elaboration and a preliminary implementation of *lifting* across ornaments. They also note that GADTs in Ocaml are an example of *reindexing* ornamentation of algebraic datatypes. Sijsling [48] uses the reflection mechanism of Agda to allow the programmer to define new types by ornamentation of existing types.

[1] Its complete Agda implementation can be found on Dagand's webpage. `https://pages.lip6.fr/Pierre-Evariste.Dagand/stuffs/journal-2013-patch-jfp/model/html/Readme.html`

2.3.1 A universe of indexed descriptions

Dagand uses the *universe encoding* to represent both base types and ornaments as description codes and interpretation functions that map descriptions to types in the host language (in this case, Agda [2]). He introduces a universe of descriptions which is essentially a datatype of syntax trees that represent definitions of type families indexed by inhabitants of a sort I:

```
data IDesc (I : Set) : Set₁ where
```

This universe contains all the expected components:

- unit types,

  ```
  ‘1 : IDesc I
  ```

- dependent products and sums (Π- and Σ-types),

  ```
  ‘Π : (S : Set )(T : S → IDesc I) → IDesc I
  ‘Σ : (S : Set )(T : S → IDesc I) → IDesc I
  ```

- non-dependent products and sums,

  ```
  ‘σ : (n : ℕ)(T : Fin n → IDesc I) → IDesc I
  _‘×_ : (A B : IDesc I) → IDesc I
  ```

- indexed recursive positions (taken at the index i).

  ```
  ‘var : (i : I) → IDesc I
  ```

Type descriptions are mapped to Agda types using an interpretation function. The interpretation function takes as an argument a description and an type family to which types of recursive positions belong. If read categorically, it takes indexed descriptions to functors from a category of indexed sets to category of unindexed sets:

```
⟦_⟧ : {I : Set} → IDesc I → (I → Set) → Set
```

The unit type is represented with a type with a single inhabitant:

[2] Readers unfamiliar with Agda should keep in mind that Agda syntax makes heavy use of Unicode and mixfix notation, and any substring enclosed by whitespace or special symbols such as parentheses is a token. For example, D^{+} and u^{-1} are identifiers and not partial applications of an operator to a variable. Contrarily, u ⁻¹ i (note the spaces!) *is* an application of an infix operator to two arguments.

```
⟦ '1 ⟧ X = ⊤
```

Π-types are represented by Agda dependent function spaces:

```
⟦ 'Π S T ⟧ X = (s : S) → ⟦ T s ⟧ X
```

Σ-types are represented by dependent pairs:

```
⟦ 'Σ S T ⟧ X = Σ[ s ∈ S ] ⟦ T s ⟧ X
```

Non-dependent products are represented by products from the Agda standard library:

```
⟦ A '× B ⟧ X = ⟦ A ⟧ X × ⟦ B ⟧ X
```

Non-dependent sums are dependent pairs where the first projection is an element of a type with finite number of inhabitants:

```
⟦ 'σ n T ⟧ X = Σ[ k ∈ Fin n ] ⟦ T k ⟧ X
```

Recursive positions are members of the type family $I \to Set$ provided as an argument to the interpretation function taken at the index i:

```
⟦ 'var i ⟧ X = X i
```

IDesc I represent types to be interpreted at a given index. To represent type families, we need functions from indices to descriptions. Due to quirks in Agda unification engine, Dagand wraps these functions in a record with a single field:

```
record func (I J : Set) : Set₁ where
  constructor mk
  field
    out : J → IDesc I
```

A member of an type family can be produced at a given index using the following interpretation function. It takes indexed descriptions to endofunctors on the category of indexed sets, setting the scene for an implementation of a fixed-point operator:

```
⟦_⟧func : {I J : Set} → func  I J → (I → Set) → (J → Set)
⟦ D ⟧func X j = ⟦ func.out D j ⟧ X
```

To make the toolkit of base types complete, a fixed-point representation is used to tie the knot. Note that values of a recursive type must have types of all subterms taken at the indices of the same type I:

```
data μ (D : func I I)(i : I) : Set where
  ⟨_⟩ : ⟦ D ⟧func (μ D) i → μ D i
```

To illustrate the use of the defined universe of descriptions, let us review a description of the type of natural numbers.

Natural numbers are not an indexed type. To encode them in a universe of indexed types, we have to represent them as a trivially indexed type family (that is, indexed by the type with a single inhabitant $\top$):

```
NatD : func ⊤ ⊤
```

Because the type is trivially indexed, we can ignore the index argument of the type family:

```
NatD = func.mk λ _ →
```

The type of natural numbers has two constructors, zero and suc. We encode constructors as injections of a sum with arity 2. Since Dagand's universe of descriptions does not use constructor names, we will distinguish constructors by their labels (inhabitants of a finite type with cardinality equal to the arity of the sum representing constructor choice): first (zero), second (suc zero) and so on.

```
  ‘σ 2
```

The first constructor represents the number 0. We encode it with a unit value.

```
  (λ { zero → ‘1
```

The second constructor represents successor values. The content of a successor value is a natural number being succeeded which we represent by a recursive position with type taken at the trivial index tt:

```
  ; (suc zero) → ‘var tt
```

There are no other injections of the sum. In Agda pattern matching, this is represented by an "absurd case":

```
  ; (suc (suc ())) })
```

Having obtained the type description, we take its fixed point at the trivial index:

```
Nat : Set
Nat = μ NatD tt
```

2.3.2 A universe of ornaments

Ornaments are represented by descriptions, just as base types are. They are also accompanied by interpretation functions but interpretations of ornaments are not host language types but base type descriptions. To obtain a host language interpretation of an ornamented type, one applies the ornament interpretation function first, and the base type interpretation function second. Due to recursive nature of the definition of the universe of ornaments, we will be presenting ornament description codes and their interpretations side by side for clarity.

Descriptions of ornaments are indexed by descriptions of base types they extend taken at an index K. The parameter u is the *reindexing function*. We will explain its meaning and use in the context of *refinement ornaments* later in this section.

```
data Orn {I K : Set}(u : I → K) : IDesc K → Set₁ where
```

⋮

An interpretation function for ornaments takes ornament descriptions to base type descriptions:

```
⟦_⟧Orn : ∀{I K : Set}{u}{D : IDesc I} → Orn u D → IDesc K
```

⋮

Since ornaments can also represent type families, the latter have a representation and an interpretation function similar to func and $[\![\_]\!]_{\text{func}}$ for base descriptions. Again, we encounter the reindexing function parameters u and v which are explained below.

```
record orn {I J K L : Set}(D : func K L)(u : I → K)(v : J → L) : Set₁ where
  constructor mk
  field
    out : (j : J) → Orn u (func.out D (v j))

⟦_⟧orn : {I J K L : Set}{D : func  K L}{u : I → K}{v : J → L}  →
       orn D u v → func  I J
⟦ o ⟧orn = func.mk λ j → ⟦ orn.out o j ⟧Orn
```

Note the case of the subscripts: $[\![\_]\!]_{\text{Orn}}$ takes ornament descriptions to base type descriptions at particular indices, and $[\![\_]\!]_{\text{orn}}$ takes ornament descriptions of type families to base type descriptions of type families.

Identity

In Dagand's universe of ornaments there are three kinds of ornaments that add information to a base type: insertion, deletion and refinement. But before them all, there are identity ornaments that leave a part of a type description unchanged:

⋮

```
‘1 : Orn u ‘1
_‘×_ : ∀{D D'} → (D⁺ : Orn u D)(D'⁺ : Orn u D') → Orn u (D ‘× D')
‘σ : ∀{n T} → (T⁺ : (k : Fin n) → Orn u (T k)) → Orn u (‘σ n T)
‘Σ : ∀{S T} → (T⁺ : (s : S) → Orn u (T s)) → Orn u (‘Σ S T)
‘Π : ∀{S T} → (T⁺ : (s : S) → Orn u (T s)) → Orn u (‘Π S T)
```

⋮

The identity ornament codes mirror the syntax of corresponding base type codes. At the first glance, they share names too but Agda can tell them apart by their type.

The identity ornaments are simply mapped to corresponding descriptions without any change:

⋮

```
⟦ ‘1 ⟧Orn = ‘1
⟦ T⁺ ‘× T'⁺ ⟧Orn = ⟦ T⁺ ⟧Orn ‘× ⟦ T'⁺ ⟧Orn
⟦ ‘σ {n} T⁺ ⟧Orn = ‘σ n (λ x → (λ D → ⟦ D ⟧Orn) (T⁺ x))
⟦ ‘Σ {S} T⁺ ⟧Orn = ‘Σ S (λ x → (λ D → ⟦ D ⟧Orn) (T⁺ x))
⟦ ‘Π {S} T⁺ ⟧Orn = ‘Π S (λ x → (λ D → ⟦ D ⟧Orn) (T⁺ x))
```

⋮

Insertion

Insertion adds a field to an existing type. It takes the type of a field being inserted S and a function D^+ that uses the inserted value to compute the rest of the ornament (it can be a constant function):

⋮

```
insert : ∀{D} → (S : Set )(D⁺ : S → Orn u D) → Orn u D
```

⋮

Insertion is interpreted as a pair type with the first projection being the new field in the type, and the second having the type represented by the second projection with the free variable substituted with the value of the new field:

⋮

```
⟦ insert S D⁺ ⟧Orn = 'Σ S (λ s → ⟦ D⁺ s ⟧Orn)
```

⋮

From naturals to lists An example of an insertion ornament are lists. A list ornament works on the type of natural numbers and keeps its recursive structure [3].. The added field is used to store the elements of the list. The ornament description type will be the following. It is a function from the type of elements to descriptions of ornaments over natural numbers: [4]

```
ListO : Set → orn NatD id id
```

Lists are trivially indexed so the index parameter is ignored:

```
ListO A = orn.mk λ _ →
```

The list type has two constructors, just as the type of natural numbers. This is preserved by using the 'σ identity ornament code for syms. Recall that numerical labels are used in the description codes instead of constructor names. The first constructor does not contain information neither in naturals (zero) nor in lists (nil). It is also left unchanged by the code '1:

```
  'σ (λ { zero → '1
```

[3] Of course, lists are not the only ornament of natural numbers.

[4] The identity functions passed as arguments are simply placeholders for reindexing functions that are used by the refinement ornament described below. They are not used by the insertion ornament.

It is the second constructor of naturals suc that is extended with an additional field to become the second constructor of lists cons. We apply the insert code to add a field of type A:

```
; (suc zero) → insert A (λ _ → `var (inv tt))
```

The lambda expression above is a constant function because we do not use the value of the inserted field to build the rest of the ornament code. In fact, it does not change the base type any further. The code `var has not been introduced by us yet, as we define it in the context of the refinement ornament later. Shortly, its use here means that the recursive position is left unchanged.

There are no other cases:

```
; (suc (suc ())) })
```

A list type in the host language is then obtained by taking a fixed point of the description interpreted at index tt (because lists are trivially indexed):

```
List : Set → Set
List A = μ ⟦ ListO A ⟧orn tt
```

Deletion

Deletion takes a Σ-type and produces a type given by the second projection where the free variable is substituted with the value given in the ornament. It can be seen as specialization of a pair type under a chosen value of the first projection:

⋮

```
deleteΣ : ∀{S T} → (s : S) (T⁺ : Orn u (T s)) → Orn u (`Σ S T)
deleteσ : ∀{n T} → (k : Fin n) (T⁺ : Orn u (T k)) → Orn u (`σ n T)
```

⋮

Interpretation of the deletion ornament works on the type level only. Because of the type of the T^+ code (Orn u (T k)), it is restricted to be a code where k has been substituted for the free variable. The action of the interpretation function on the code terms is simply traversal into their subterms:

⋮

```
⟦ deleteΣ s T⁺ ⟧Orn = ⟦ T⁺ ⟧Orn
⟦ deleteσ k T⁺ ⟧Orn = ⟦ T⁺ ⟧Orn
```

⋮

Refinement

Refinement makes it possible to defined indexed families or extending indices of already defined ones with additional information. Here we finally define *reindexing functions* and explain their use.

A reindexing function (in the current presentation, the parameter `u` of the universe of ornaments) takes indices of a refined type to indices of a base type. The `‘var` ornament code is used to give new, extended indices to recursive positions in a base type. We have seen its trivial use in the list ornament example where it was used to preserve indexing (the reindexing function was identity).

⋮

```
‘var : ∀{i} → (i⁻¹ : u ⁻¹ i) → Orn u (‘var i)
```

Its argument is an inverse image of the reindexing function at the base type index which is a value of the new, extended index type. Inverse images are represented by the type `_⁻¹_`:

```
data _⁻¹_ {A B : Set}(f : A → B) : B → Set where
  inv : (a : A) → f ⁻¹ (f a)
```

Since the new indices are inverse images of the reindexing function, the interpretation of a refined recursive position is a recursive position taken at the new index:

⋮

```
⟦ ‘var (inv i⁺) ⟧Orn = ‘var i⁺
```

From lists to vectors by constraint For example, lists can be refined with indices representing lists. The resulting type is vectors. One possible ornament description taking lists to vectors is defined as follows. It uses the reindexing function `u` which only has to produce a trivial index for any vector length (as lists are trivially indexed):

```
u : ℕ → ⊤
u _ = tt

VecO : orn (ListD A) u u
```

We keep the structure of the list datatype, the type of vectors will still have two constructors (nil and cons):

```
VecO = orn.mk λ n → 'Σ {S = Fin 2}
```

The index value n will be used to construct the type of proofs which witness that the vector length is given by its index. The constructor nil is extended with a proof that the length is zero. Essentially, it means the only way to construct an empty vector is to use the nil constructor. In the other direction, the nil constructor can only be used if a proof of the equality of the length to zero exists.

```
    λ { zero → insert (0 ≡ n) λ _ → '1
```

The constructor nil is extended with a proof that the vector length is non-zero. Moreover, it must be exceed the length of the vector tail exactly by one. To that end, we keep the index m representing the tail length in an inserted field, so we can use later in the recursive position code:

```
      ; (suc zero) → insert ℕ λ m →
      insert (suc m ≡ n) λ _ →
```

We keep the element field unchanged

```
                'Σ λ _ →
```

but indicate that the length of the vector tail must be m, as indicated by an index extended from a trivial one to a natural number:

```
                    'var (inv m)
      ; (suc (suc ())) }
```

We obtain the vector type, again, by interpreting the ornament description. This type we have to interpret at an index representing the vector length:

```
Vec : ℕ → Set
Vec = μ ⟦ VecO ⟧orn
```

From lists to vectors by computation A better way to construct an ornament from lists to vectors does not rely on insertion of proofs. Instead of requiring to supply a proof of length being equal to the index, we can determine which constructors can be used under an index directly. This encoding style reduces redundancy of ornament codes because indices need not to be stored anymore. [14]. This ornament uses the same reindexing function u that always returns trivial indices, and the type signature of the ornament stays the same as in the previous example:

```
VecO : orn (ListD A) u u
```

The novelty is using the vector index to determine available constructors. In the previous example, we have not looked at the index and only used it in the definitions of constraints for each constructor. Here, we pattern-match on the index. If the index is zero (that is, the vector is empty), we use the deletion ornament code to specialize the description to the choice of the nil constructor. Essentially, we do not allow the use of any other constructor than the first one but the rest of the ornament code is left unchanged:

```
VecO = orn.mk λ { zero → deleteΣ zero ‘1
```

If the index is non-zero, we specialize the description to the choice of the cons constructor. The only constructor that can be used in this case is the second one:

```
; (suc n) → deleteΣ (suc zero)
```

The ornament code has to change the index of the recursive position which represents the list/vector tail. We mandate that the length of the tail must be n, that is, one less than the length of the vector constructed with the cons constructor (suc n):

```
(‘Σ λ _ → ‘var (inv n))
```

2.3.3 Algebraic ornaments

An important application of ornaments is algebraic ornamentation. It combines insertion and refinement to create an indexed family where the index represents a property of the indexed value. These properties have to be computable with catamorphism over the value.

An algebraic ornament enriches the index k by pairing it with result x of applying a catamorphism for algebra α over the value t. It also inserts

a field containing a witness that the result of applying the catamorphism $(\!|\ \alpha\ |\!)$ is indeed equal to the new index x:

$$\mu\ D^{\alpha}\ (k : K, x : X\ k) \cong (t : \mu\ D\ k) \times (\!|\ \alpha\ |\!)\ t \equiv x$$

equivalently, in the refinement types notation [3]:

$$\{t \in \mu\ D\ k \mid (\!|\ \alpha\ |\!)\ t = x\}$$

The example of obtaining vectors by ornamenting lists with their length is also an example of an algebraic ornament because the length of a list can be computed with a list catamorphism.

A particular kind of algebraic ornaments are *reornaments*. A reornament is an algebraic ornament by an *ornamental algebra*. An ornamental algebra forgets the extra information introduced by an ornament.

In the running example of natural numbers, lists and vectors, a catamorphism for the ornamental algebra of the list ornament over naturals computes the length of a list. The extra information introduced by the list ornament is list elements. If they are removed, we obtain a natural number built with as many suc constructors as the list has cons constructors.

If the list type is given a new index computed as a catamorphism for the ornamental algebra of the list ornament, that is, the length function, the resulting type is vectors. Therefore, the vector ornament over lists is a reornament of the list ornament over natural numbers.

2.3.4 Reornaments witness coherence properties

If a type is defined as an ornament of another type, the two types share their recursive structure. There will be functions over a base type and its ornament that have similar recursive definitions as well. In the running example of natural numbers and lists, one such pair of functions is addition and concatenation:

```
_+_ : ℕ → ℕ → ℕ
zero  + m = m
suc n + m = suc (n + m)

_++_ : ∀ {A} → List A → List A → List A
[]       ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

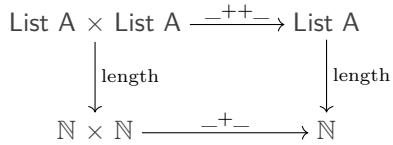


Figure 2.2. Commutative diagram of addition of naturals and list concatenation

Up to constructor names, the only difference in their structure is that concatenation needs to pass around the content of the element field inserted by the list ornament. The relationship between these two functions can be captured in a commutative diagram (Fig 2.2, note uncurrying). This commutative diagram illustrates a free theorem of a particular kind, a *coherence property*.

Recall that vectors are a reornament of lists, and the list length function is a forgetful map associated with the list ornament. The definition of vector concatenation is only different from the definition of list concatenation in its type:

```
_++_ : ∀ {A m n} → Vec A m → Vec A n → Vec A (m + n)
[]       ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

The vector concatenation function with type ∀ {A m n} Vec A m × Vec A n → Vec A (m + n) is *witnessing* coherence between addition of natural numbers and concatenation of lists.

The importance of reornaments lies in establishing *coherence* between functions over base types and functions over ornamented types (Fig. 2.3). The pinnacle of Dagand's library is a set of combinators for *transporting* functions over base types to functions over ornaments. With its help, reornaments become a technique for defining functions *correct by construction*, combining implementation and proof.

In Section 3.5, we use *algebraic ornaments* to extend the type of positions in a tree with additional information to distinguish positions that belong to different *tiles*. Then we apply Dagand's transporting machinery to build tiled traversals as reornaments of untiled ones.

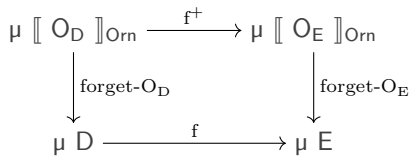


Figure 2.3. A generic coherence property

Chapter 3

Tiling as an ornament

3.1 No Order, No Change

Before we can capture the precise meaning of tiling a computation over an arbitrary polynomial datatype, let us start with a concrete example. Implementing a tiled functorial map over a binary tree will provide us with necessary intuition that we will be able to generalize.

Consider the usual definitions: [1]

```
data Tree (A : Set) : Set where
  leaf : Tree A
  node : Tree A → A → Tree A → Tree A

fmap : ∀ {A B} → (A → B) → Tree A → Tree B
fmap f leaf = leaf
fmap f (node l x r) = node (fmap f l) (f x) (fmap f r)
```

To compile them to a low-level language like C, one has to choose the evaluation order first. We might immediately decide on call-by-value to keep the runtime of the target language simple. Even then, we might prefer the generated program to traverse the tree depth- or breadth-first, in pre-, in-, or post-order. To be explicit about these decisions, we apply the CPS transformation:

```
map : ∀ {A B} → (A → B) → Tree A → Tree B
map f t = mapc f t id
mapc : ∀ {A B}{c : Set} →
  (A → B) → Tree A → (Tree B → c) → c
mapc f leaf k = k leaf
mapc f (node l x r) k =
    (mapc f l (λ l' → mapc f r (λ r' → k (node l' (f x) r'))))
```

[1] This chapter is a Literate Agda document. To reduce the amount of visual noise the reader has to cope with, we will exclude some parts of code from typesetting, for example, many of "absurd cases" that are used in Agda to show exhaustiveness of pattern matching, or some "implicit parameters" that have to be given only to help the unification engine.

Here we have implemented a depth-first post-order traversal. One detail is still left implicit for the C generator to figure out — how to represent continuations. A functional language has closures, but C does not. Because the continuations take limited forms in this example, we can represent them all with a data structure, and that data structure turns out to be the zipper.

3.2 (With A Zipper) Everyone Knows Their Place

Let us assume for a moment that we know how to translate functions over containers[2] to functions over zippers isomorphic to them. (Peek to Sec. 3.4 if the temptation is unbearable.) We can resume our construction of map considering the following slightly unusual presentation of a zipper. The need for a new presentation will become clear in Section 3.6 when we discuss the proof of correctness for the tiling transformation.

A zipper is data structure to represent a current position in a container. It is defined a pair of the focused part of the container and a list of one-hole contexts [2]. One-hole contexts represent a path from the container root to the current position. Their type can be obtained by applying rules similar to the rules of partial differentiation of polynomials from calculus, thus, giving them another name, functor *derivatives*.

To simplify the treatment in Sec. 3.6, we want to define both these as a single type indexed by the sort of its inhabitants, *polynomials* (pln) and *derivatives* (drv), directly.

The reason for a slight change to the presentation is that the latter type is not an inductive but a nested one. In Sec. 3.4, we will use a datatype description technique, ornamentation [22], which only allows for representation of inductive types. Luckily, the usual presentation of the zipper as a pair of a container and a list of container derivatives happens to be a *mutually inductive* datatype. A mutually inducted type can be encoded as a family indexed with sorts; Yakushev et al. [55] use the trick for the zipper itself.

First, we require an index set:

```
data Z : Set where
  pln drv : Z
```

Then, we define a datatype with constructors for both subcontainers and one-hole contexts. The first two constructors generate inhabitants of

[2] I.e., strictly positive functors built with sums, products, constants and recursion.

the pln sort. It corresponds to the definition of the labeled binary tree in our example, given as the least fixed point of $FX = X \times A \times X + 1$.

```
data Zipper (A : Set) : Z → Set where
  leaf  : Zipper A pln
  node  : Zipper A pln → A → Zipper A pln → Zipper A pln
```

⋮

Before we define the type of one-hole contexts, we need a type of tags H to distinguish between holes on the left and holes on the right:

```
data H : Set where
  L R : H
```

The last two constructors generate inhabitants of the drv sort. It corresponds to the derivative $F\prime Y = 2 \times X \times A \times Y$ extended with a termination symbol plug.

⋮

```
  plug :  Zipper A pln  → Zipper A drv
  hole : H → Zipper A pln → A → Zipper A drv → Zipper A drv
```

The zipper operations are implemented only slightly differently. Take the "plugging in" operation, for example. As an operation that moves focus upwards, it might not come to a valid position, if we were in the root of the tree already:

```
up : ∀{A} → Zipper A drv → Maybe (Zipper A drv)
up (plug t)  = nothing
```

There are only two extra cases to treat the termination symbol:

```
up (hole L p x (plug t))   = just (plug  (node p x t))
up (hole R_r p x (plug t)) = just (plug  (node t x p))
```

The remaining cases are standard, constructing a new context out of the focused subcontainer and and a subtree "cast aside" by focusing left or right in the past:

```
up (hole L p x (hole h t y d)) = just (hole h (node p x t) y d)
up (hole R p x (hole h t y d)) = just (hole h (node t x p) y d)
```

We can recursively apply the up operation to a functor until we focus the root.

```
root : ∀{A} → Zipper A drv → Maybe (Zipper A pln)
root z with up z
root z          | just u  = root u
root (plug z)   | nothing = just z
root (hole _ _ _ _) | nothing = nothing - can't happen
```

Zippers don't become structurally smaller when traversed so Agda's termination checker doesn't see that recursive functions taking more than one step actually terminate. We will address this in the following section by giving a more precise type to the zipper operations.

Implementation of moving focus to the left child does not bring surprises either. If the focus is on a leaf, there is nowhere to move further:

```
left : ∀{A} → Zipper A drv → Maybe (Zipper A drv)
left (plug  leaf)        = nothing
left (hole h leaf x d)   = nothing
```

If the child is another node, construct a left-focused context with the right child cast aside:

```
left (plug  (node l x r))  = just (hole L l x (plug r))
left (hole h (node l x r) y d) = just (hole L l x (hole h r y d))
```

Focusing on the right child is only different in that it produces a right-focused context with the left child cast aside.

The implementation of operations to access the root label of the focused subtree, read and write, is straightforward.

As a special case, let us demonstrate how to perform a root label update. This operation will come handy when working with zippers containing labels of *different types at the same time*:

```
update : ∀{A B} → (A → B)
  → Zipper (A ⊎ B) drv → Zipper (A ⊎ B) drv
update f z with read z
... | just (inj₁ x)  = write (inj₂ (f x)) z
... | just (inj₂ x)  = z
... | nothing = z
```

Having rebuilt our tree datatype with the zipper machinery, we can resume the implementation of a tree map. Let us assume a left-to-right traversal. To implement one, it is useful to have a subtree-root operation to discard a chunk of the stack after we have hit a rightmost leaf in a subtree.

For example, in Fig. 3.1, after visiting node 21, traversal needs to continue with node 5 and its right child, node 11. Such subtree-root is implemented similarly to root but has to terminate as soon as a left-focused context is found:

```
subtree-root : ∀{A} → Zipper A drv → Maybe (Zipper A drv)
subtree-root z with z
... | plug _  = nothing
... | hole L _ _ _  = just z
... | hole R _ _ _  = (up z) »= subtree-root
```

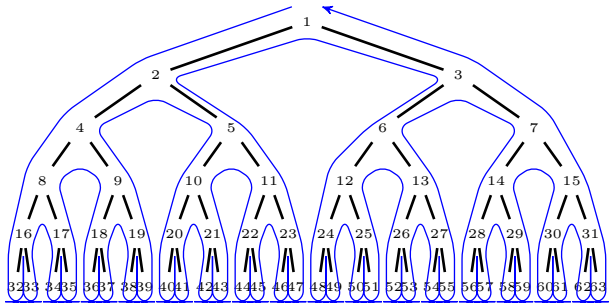


Figure 3.1. Tree traversal order, untiled

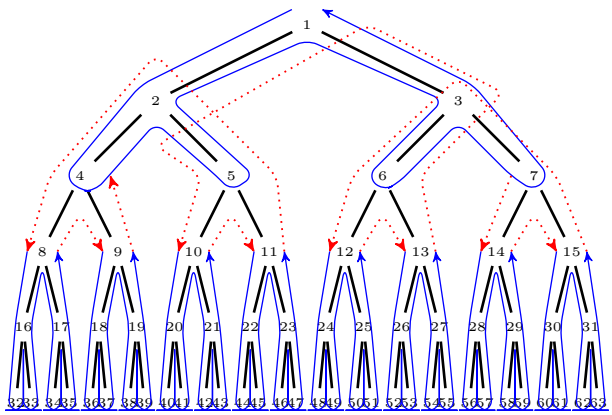


Figure 3.2. Tree traversal order, tiled

Now we are finally equipped to produce a map function. It has to work on zippers containing *elements of two types*. A zipper will have both in the middle of traversal. Just as in case of root and subtree-root , the termination checker cannot be convinced that it terminates:

```
zmap : ∀ {A B} → (A → B)
  → Zipper (A ⊎ B) drv → Maybe (Zipper (A ⊎ B) drv)
```

At any point during traversal we are interested in two things: whether we can still go left and whether we have encountered a rightmost leaf in a subtree:

```
zmap f z with left z | subtree-root z
```

If there is a left child, continue moving. Since we are doing an in-order traversal, the label update will happen afterwards.

```
... | just x  | _  = zmap f x
```

If there is none, we need to return to the root of the current subtree. We can already be in one, if we are focusing a left child already. In any case, the unvisited nodes are to our right. But before we move to a sibling subtree, the node label has to be updated.

```
... | nothing | just x  = up x
  »= right ∘ update f
  »= zmap f
```

If we cannot move to the left, and there is nothing to traverse on the right, we are done. Another way to look at it, we are done when the current subtree equals to the whole tree.

```
... | nothing | nothing = just z
```

3.3 Recursion Must Be Recursive, Ahem, Mutual

Look back at Fig. 3.2. In a tiled traversal, one does not visit nodes in other tiles before visiting all nodes in the current one. We now need to modify the zmap to be explicit not only about the position in the tree but also the current tile. The zipper itself does not have any information about tiles, of course.

Checking tile boundaries requires simultaneous recursion on two arguments, the container and the position in the current tile. The latter can be represented with a zipper containing dummy elements ($\top$), just as the container is represented with a zipper. We can lift zipper operations to change focus in two zippers simultaneously. (Section 3.5 will show how to avoid this by more precise typing of operations.)

Given a type

```
pairstep : ∀{A}
```

and an operation on the zipper containing elements of that type

```
  → (∀ {A} → Zipper A drv → Maybe (Zipper A drv))
```

we transform it to an operation acting simultaneously on that zipper and a position in a current tile with respect to the stack of positions leading to the root:

```
  → Zipper ⊤ drv × List (Zipper ⊤ drv) × Zipper A drv
  → Maybe (Zipper ⊤ drv × List (Zipper ⊤ drv)
    × Zipper A drv)
pairstep f (t , bb , z) with t
... | plug _ = nothing
... | hole L _ _ _ = just (t , bb , z)
```

The operation is attempted in sequence first on the zipper, then on the position, if any of them fails, the cumulative result is nothing.

```
... | hole R _ _ _ =
  f t »= (λ x → f z »= (λ y → return (x , bb , y)))
```

During a tiled traversal, we might encounter subtrees beyond the boundaries of the current tile but we will not traverse them before finishing the tile. When a tile is fully traversed, for example, in an in-order, depth-first, left-to-right fashion, the focus is on a rightmost subtree. We can begin traversal of that subtree with tile boundaries in mind, but what about subtrees that are not rightmost? We need to be able cast aside subtrees with roots at tile boundary and return to them later. To that end, we apply the zipper trick to the zipper itself.

The traversal of a tree begins at the tree root. The root of the first tile coincides with it. We traverse all elements in the tile, checking on every step if we are not crossing its boundary. When all elements in the tile are traversed, the zipper is focused at the rightmost subtree with a root at a boundary of the tile. Note that when covering a tree with tiles, leafs and roots of neighboring tiles overlap.

The traversal can continue with that new tile, resetting the position in the tile to the tile root. But sooner or later there will not be a rightmost subtree to contunue. We will have to continue with a sibling tile, that is, a tile immediately reachable from the one traversed before the current. We can trace our steps back to the previous tile, but we would not know at which of its leafs we ended up. To know that, before we reset the tile position when starting a new tile, we push it on a stack. This way, when returning to a previously traversed tile, we can proceed to the next untraversed subtree.

Fig. 3.3 shows an example of a tiled traversal state. The current tile is

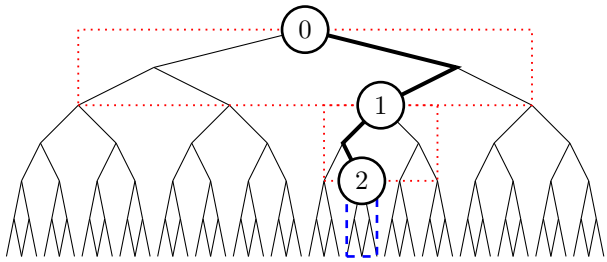


Figure 3.3. Position of a tile

the dashed segment and labeled as 2, The two dotted segments labeled 0 and 1 are fully traversed tiles. Note that the label of each tile corresponds with the number of tiles separating its root from the root of the tree.

The dotted segments represent the unit-valued zippers we have used to check their boundaries and pushed on the stack before moving on to the next tile. The positions where a new tile began lie on bold path going to the tree root. Tiles not yet traversed are to the left of that path.

We can finally implement a tiled zmap. It will be a pair of two mutually recursive functions, one for traversal inside of a tile, another for moving focus to the next untraversed tile. The implementation of the inner traversal is similar to that of an untiled zmap, save for simultaneous recursion on the tree and and on the current tile to stay with tile boundaries. The subtree-root-in function moving focus to the next unvisited node within a tile follows the implementation of subtree-root:

```
mutual
  subtree-root-in : ∀{A} →
    Zipper ⊤ drv × List (Zipper ⊤ drv) × Zipper A drv
    → Maybe (Zipper ⊤ drv × List (Zipper ⊤ drv)
      × Zipper A drv)
  subtree-root-in (t , bs , z) with t
  ... | plug _ = nothing
  ... | hole L _ _ _ = return (t , bs , z)
  ... | hole R _ _ _ = pairstep up (t , bs , z) »= subtree-root-in
```

So does zmap-tiled-inner in respect to zmap, with one slight but significant difference.

```
zmap-tiled-inner : ∀ {A B} → ℕ → (A → B) →
  Zipper ⊤ drv × List (Zipper ⊤ drv) × Zipper (A ⊎ B) drv
  → Maybe (Zipper ⊤ drv × List (Zipper ⊤ drv)
    × Zipper (A ⊎ B) drv)
zmap-tiled-inner n f (t , bs , z) with pairstep left (t , bs , z)
```

```
... | just (x , _ , y) =  zmap-tiled-inner n f (x , bs ,  y)
... | nothing  = maybe′ - tile not done
    (λ x →
    return x
    »= pairstep up
    »= (λ { (t , bs , z) → return (t , bs , update f z) })
    »= pairstep right)
```

When there are no more elements to visit after a rightmost leaf, only the current tile traversal is finished, not traversal of the whole tree. We need to move to the next tile when this happens:

```
- tile done
(zmap-tiled-outer n f (t , bs , z))
- done/not done?
(subtree-root-in (t , bs , z))
```

The outer traversal bears much similarity with the inner one. It is, although, different in that it does not update labels and its traversal direction is symmetrically opposite, right-to-left. The inner traversal stops in the rightmost leaf of a tile, when done from left to right. If there is a subtree beginning right afterwards, that is our next tile. If there is none, we must return to the root of a subtree using the stack of tile positions to find the root of a sibling tile to the left. The subtree-root-in function does exactly that. It discards a chunk of stack with all contexts with a hole on the right in order to find the next unprocessed subtree to the right. Analogously, subtree-root-out discards contexts with holes on the left inside a tile to find the common ancestor to the current tile and the next unprocessed one.

```
subtree-root-out : ∀{A} →
    Zipper ⊤ drv × List (Zipper ⊤ drv) × Zipper A drv
    → Maybe (Zipper ⊤ drv × List (Zipper ⊤ drv)
        × Zipper A drv)
subtree-root-out (plug _ , []  , _) = nothing
subtree-root-out (plug _ , b :: bs , z) =
    up z »= (λ z → return (b , bs , z))
subtree-root-out (hole L t x ctx , bs , z) =
    pairstep up (hole L t x ctx , bs , z)
    »= subtree-root-out
subtree-root-out (hole R t x ctx , bs , z) =
    return (hole R t x ctx , bs , z)
```

Then, zmap-tiled-outer could traverse *from right to left* until the tile

boundary to identify the next tile.

```
zmap-tiled-outer : ∀ {A B} → ℕ → (A → B) →
  Zipper ⊤ drv × List (Zipper ⊤ drv) × Zipper (A ⊎ B) drv
  → Maybe (Zipper ⊤ drv × List (Zipper ⊤ drv)
    × Zipper (A ⊎ B) drv)
zmap-tiled-outer n f (t , bs , z) with pairstep right (t , bs , z)
... | just (x , _ , y)  = zmap-tiled-outer n f (x , bs , y)
... | nothing = maybe′ - tile ahead
  (λ _ → zmap-tiled-inner n f (ztilegen n , t :: bs , z))
  - tile behind
  (maybe′ - tree not done
    (λ x →
    return x
    »= pairstep up
    »= pairstep left)
    - tree done
    (return (t , bs , z))
    - done/not done?
    (subtree-root-out (t , bs , z)))
  - ahead/behind?
  (right z)
```

3.4 The Applied Art of Ornamentation

The code we have written in Sec. 3.2 and 3.3 is quite verbose. We had to account for many ways a traversal can fail and implement error handling accordingly. Even having done that, we still do not know if a subtle mistake is not lurking somewhere! One way to exclude undesired behavior of a program is to give it a richer type.

In his recent work on constructing search trees [43], McBride shows how an invariant can be "baked" into a datatype, step by step. His method rests upon the idea of *refinement* of datatypes with folds over them that calculate the property to be enforced [3]. We will attempt a similar undertaking but with help of a framework of type refinement coauthored by McBride and Dagand [21].

The framework lets us to define *codes* for datatypes and their refinements. The datatypes subject to refinement are referred to as *base* types, and their refinements *ornamented* types (or simply *ornaments*). Given a function between base types and a function between corresponding ornamented types, one can obtain a *coherence certificate* witnessing a

naturality property. We will use this machinery to show equivalence of tiled and untiled traversals of arbitrary container datatypes definable in the universe of codes in Sec. 3.6. But even before that, ornaments will be helpful to demonstrate how container datatypes are related to their zippers.

We use Dagand's library of ornaments in our construction. It contains definitions of a universe of indexed families and a universe of ornaments. It also provides combinators for writing recursive functions over types in these universes and combinators for *transporting functions across ornaments*, a technique for writing functions correct by construction that we will use in Sec. 3.6.

A base type We begin with a code for an unlabeled binary tree type. In Dagand's library, the func I J type represents descriptions indexed with inhabitants of J and with recursive occurences indexed by I. Recursive datatypes can be obtained by taking a fixed point of a description where both index sets coincide. The single constructor mk of the type func takes as an argument a function producing descriptions indexed by I from inhabitants of J.

The pattern functor of a binary tree is trivially indexed (that is, its type is $(1 \to Set) \to (1 \to Set)$). The type of the tree description needs to indicate that both domain and codomain of the pattern functor are indexed by elements of ⊤ which has only one inhabitant.

```
TreeD : func ⊤ ⊤
```

The code ‘σ represents non-dependent sum types. We use it to represent the choice between two constructors, leaf and node:

```
TreeD = mk λ _ → ‘σ 2
```

The universe of codes does not provide syntax for arbitrary labels, so numbers are used instead. At the constructor labeled zero (or leaf), there is no interesting data. The unit code (‘1) expresses exactly that.

```
  (λ { zero → ‘1
```

At the constructor labeled suc zero (or node), there is a pair of subtrees. The ‘var code means "an inhabitant of the member of the datatype family being described, at the given index". In this case of a trivially indexed datatype, the index tt is the only inhabitant of ⊤.

```
  ; (suc zero) → ‘var tt ‘× ‘var tt
```

There are no other constructors, which is indicated in Agda by

```
  ; (suc (suc ())) })
```

This is the first and last time we typeset an absurd case in this chapter.

By taking a (least) fixed point of this functor, we obtain the type of unlabeled binary trees.

```
Tree : Set
Tree = μ TreeD tt
```

Zippers as ornaments of trees A code for the unlabeled binary tree type can be extended into a code for the corresponding zipper type with an appropriate ornament. The datatype we will obtain will not be Tree × [Tree'] but isomorphic to it. To avoid confusion with the usual presentation of zippers, let us call the datatype slightly differently, Pipper ("position zipper"). A Pipper marks a "position" in a binary tree. What else can a zipper-like datatype represent? This will come a few paragraphs later.

In the ornamental setting, producing a type with a *richer* index that the base one is known as *refinement*. A refinement ornament does not add new information to the terms inhabiting a type but to the index of the type. The connection between the new index containing more information to the old index is made by a function which erases this new information. In the example of refining a type into a mutually inductive one, we need a *reindex* function that takes the sort of mutually inductive types and returns a trivial index.

```
u : Z → ⊤
u _ = tt
```

Just as datatypes, ornaments are given as a universe of codes. The index of the ornamented type is related to the index of the base type with the reindex function.

```
PipperO : orn TreeD u u
PipperO = mk λ
```

We add one additional sort fcs to the index set Z to represent the pair of a tree and a list of tree derivatives (a context). We piggyback on the node code of the base tree to represent it and throw away the leaf by applying deleteσ. Then, we refine both recursive occurences of the tree to represent fragments of the zipper type we are interested in. We need to provide new sorts of the recursive occurences. This is done by constructing a proof of the fact that application of the reindex function to the new sorts yields an "old" index value. Evidence of existence of an inverse image of function u at a has type ((a : Z) → u $^{-1}$ a). This type has a single constructor inv which takes a domain value of the reindex function as its single argument.

```
{ fcs → deleteσ (suc zero) ('var (inv pln) '× 'var (inv drv))
```

To obtain the fragment of the ornamented type representing trees, no more than the identity ornament is needed. The codes of identity ornaments reuse the names of corresponding description codes. Thus, the definition of an identity ornament representing the pln sort in Pipper looks the same as the description of Tree.

```
; pln  → ‘σ (λ { zero → ‘1
    ; (suc zero) → ‘var (inv pln) ‘× ‘var (inv pln)
    ; (suc (suc ())) })
```

To encode the list of derivatives, we keep both constructors with the identity ornament. The leaf will serve as the empty list, and node as the cons cell. Now we have to change what is stored in the cells.

The derivative of the functor $FX = X \times X + 1$ is $2 \times X$. The unit is not there to begin with, and one of the recursive occurences we refine to the recursive occurence of the list. Another recursive occurence we refine to store a subtree. We only need to insert a field containing the hole tag into the code of the tree type with the insert ornament. The hole tags are represented by the type H which is isomorphic to $1 + 1$. It has two inhabitants L and R, to represent holes on the left and on the right correspondingly.

```
; drv → ‘σ (λ { zero → ‘1
    ; (suc zero) → insert H (λ _ → ‘var (inv pln) ‘× ‘var (inv drv))
    ; (suc (suc ())) }) }
```

We complete the construction by interpreting the ornament code as a type in the host language (Agda) and taking a least fixed point.

```
Pipper : Z → Set
Pipper = μ ⟦ PipperO ⟧orn
```

Zipper operations Now we can repeat the construction of up, left and right for the Pipper. We will introduce a small change — we will not use the Maybe monad to handle impossible steps in the zipper. We will simply return the zipper unchanged if it was not possible to move left, right or up. As we will see by the end of this section, a zipper traversal can be built without monadic error handling.

We express each operation as a fold with an appropriate algebra. The three algebras act similarly on the pln and drv fragments of the datatype that represent focused subtrees and lists of one-hole contexts correspondingly That action is identity, and we will only review its implementation for pleft.

The type of all three algebras indicates two things. They must act on a type obtained by interpreting the PipperO ornament as a type in

the host language. The result of applying any of the three algebras has a type isomorphic to Pipper: a product of Pipper and the unit type. The need for this will become clear in the next section when we will ornament not only types but functions on them. To that end, we will use a universe of function type descriptions which uses the unit type as the termination symbol of descriptions.

```
pleft-α : Alg ⟦ PipperO ⟧orn (λ x → Pipper x × ⊤)
pleft-α {pln} (zero , tt) = ⟨ zero , tt ⟩ , tt
pleft-α {pln} (suc zero , (l , tt) , r , tt) =
  ⟨ suc zero , l , r ⟩ , tt
pleft-α {drv} (zero , tt) = ⟨ zero , tt ⟩ , tt
pleft-α {drv} (suc zero , h , (c , tt) , cs , tt) =
  ⟨ suc zero , h , c , cs ⟩ , tt
```

The difference between the three algebras lies in the action on the fcs fragment which represents pairs of focused subtrees and lists of one-hole contexts. In the case of pleft, we pattern-match on the focused subtree, and if it consists of a single leaf, we return it immediately paired with the unchanged list of one-hole contexts.

```
pleft-α {fcs} ((⟨ zero , tt ⟩ , tt) , cs , tt) =
  ⟨ ⟨ zero , tt ⟩ , cs ⟩ , tt
```

If the focused subtree has at least a node, we have the left subtree to move to. We construct a new list of one-hole contexts from the right subtree that is "casted aside" and the old list. The left subtree becomes the new focused subtree.

```
pleft-α {fcs} ((⟨ suc zero , l , r ⟩ , tt) , cs , tt) =
  ⟨ l , ⟨ suc zero , L , r , cs ⟩ ⟩ , tt
```

The action pright on pairs of focused subtrees and one-hole contexts is similar. The difference is that it is the left subtree being "cast aside" and the right subtree becoming the new focus. We do not repeat this construction for brevity.

The action of pup is opposite to that of pleft and pright. It is not an inverse though, due to the latter behaving as identities on zippers that focus on leaves. Here, we pattern-match on the list of one-hole contexts instead of pattern-matching on the focused subtrees. If the list is empty, we do nothing.

```
pup-α {fcs} ((f , tt) , ⟨ zero , tt ⟩ , tt) =
⟨ f , ⟨ zero , tt ⟩ ⟩ , tt
```

If the list is non-empty, we construct a new focused subtree by "plugging in" the old focused subtree on the left of on the right of its head correspondingly.

```
pup-α {fcs} ((f , tt) , ⟨ suc zero , L , c , cs ⟩ , tt) =
  ⟨ ⟨ suc zero , f , c ⟩ , cs ⟩ , tt
pup-α {fcs} ((f , tt) , ⟨ suc zero , R , c , cs ⟩ , tt) =
  ⟨ ⟨ suc zero , c , f ⟩ , cs ⟩ , tt
```

Canonical forms of zippers Even though pleft, pright and pup are defined as folds, they can also be defined non-recursively. To illustrate how one would write a recursive function over the Pipper, we implement an operation recovering the tree that a Pipper value represents. This operation, proot, is an equivalent to root defined in Sec. 3.2. It "plugs in" the focused subtree recursively into all one-hole contexts. The result is the canonical form of the zipper, that is, the tree a position in which a given zipper represents put into an empty context. This operation will come in handy in the next section when we will be discussing zippers representing a tree of a certain size.

To define the algebra of proot, we need to define the types of the results of its application. There is a slight twist here. The pln and fcs sorts can be expected to have a tree as their canonical form. Indeed, the focused subtrees are already in the canonical form, and pairs of focused subtrees and lists of one-hole contexts must have one.

```
proot-sig : Z → Set
proot-sig pln = Pipper pln
proot-sig fcs = Pipper pln
```

But what about the drv sort? How can a list of one-hole contexts have a canonical form? Even if we "plug in" all contexts into each other, we will still have one empty hole! We must speak of the canonical form of the list of one-hole contexts as a one-hole context itself. But it will not make sense to speak of the position of the hole in such a context as "on the left" or "on the right". The hole will be at a leaf in a tree. To resolve this dilemma, we define the canonical form of the list of one-hole contexts as a function which will take canonical forms of focused subtrees to canonical forms of zippers.

```
proot-sig drv = Pipper pln → Pipper pln
proot-α : Alg {Z} ⟦ PipperO ⟧orn proot-sig
```

As we have just mentioned, the canonical form of subtrees coincides with them by definition. The action on the pln fragment is identity.

```
proot-α {pln} (zero , tt) = ⟨ zero , tt ⟩
proot-α {pln} (suc zero , l , r) = ⟨ suc zero , l , r ⟩
```

We decided to represent one-hole contexts as functions. We "plug in" them into each other to produce a function between canonical forms of

subtrees and canonical forms of zippers.

```
proot-α {drv} (zero , tt) = λ x → x
proot-α {drv} (suc zero , L , c , cs) = λ x → cs ⟨ suc zero , x , c ⟩
proot-α {drv} (suc zero , R , c , cs) = λ x → cs ⟨ suc zero , c , x ⟩
```

Finally, we "plug in" the focused subtree into the list of contexts by a simple function application.

```
proot-α {fcs} (f , cs) = cs f
proot : Pipper fcs → Pipper pln
proot = fold ⟦ PipperO ⟧orn (λ {i} → proot-α {i})
```

Untiled traversal We are ready to implement a traversal on Pipper. Each Pipper value represents a position in a tree of some shape. If that shape is known, we can a priori know what the sequence of primitive steps will make a traversal of it. Let us define what such a sequence must be for any tree shape.

An empty tree does not require any traversal at all, so its traversal function is identity.

```
ptraverse-α : Alg TreeD (λ _ → Pipper fcs → Pipper fcs × ⊤)
ptraverse-α (zero , tt) = (_, tt)
```

To traverse a non-empty tree, we need to step into its left child, traverse it, step up and repeat the same on the right child.

```
ptraverse-α (suc zero , l , r) = pup ∘' r ∘' pright ∘' pup ∘' l ∘' pleft
ptraverse : Tree → (Pipper fcs → Pipper fcs × ⊤)
ptraverse = fold TreeD ptraverse-α
```

Because the step operations return a Pipper paired with a unit value, we need a slightly modified function composition operator ∘' that will strip that unit value.

```
_∘'_ : ∀ {A : Set} →
  (A → A × ⊤) → (A → A × ⊤) → (A → A × ⊤)
f ∘' g = g ∘ proj₁ ∘ f
```

Tiled traversal If we know the shape of the tree to be traversed and the tile shape to change the traversal order, we can generate a sequence of steps for the tiled traversal too. In Sec. 3.2, we have required the pairstep function to keep track of the position in the current tile. Similarly, we will use recursion on two arguments here as well. To that end, we define the product of two trees, the first projection will be used to denote the position in the current tile, and the second to denote the tree being traversed.

```
Tree×TreeD : func ⊤ ⊤
Tree×TreeD =
  mk (λ _ → func.out TreeD tt ‘× func.out TreeD tt)

Tree×Tree : Set
Tree×Tree = μ Tree×TreeD tt
```

As we have mentioned earlier in Sec. 3.2, the inner traversal corresponds to a traversal of an untiled container. Essentially, we repeat the construction from the previous paragraph in a new setting. To traverse a tree leaf, no further action is needed.

```
ptraverse-α-tiled : Tree
  → Alg Tree×TreeD (λ _ → Pipper fcs → Pipper fcs × ⊤)
ptraverse-α-tiled t (_ , zero , tt) = (_, tt)
```

To traverse a tree node, we descend to the left child, traverse it and ascend back. The right child is traversed in the same way.

```
ptraverse-α-tiled t (_ , suc zero , l , r) =
  pup ∘' r ∘' pright ∘' pup ∘' l ∘' pleft
```

Recall that the outer traversal corresponds to traversal of a container of subcontainers. Essentially, outer traversal must revisit all leafs. Thus, the outer traversal is an inversion of the inner traversal.

```
ptraverse-β-tiled : Tree
  → Alg Tree×TreeD (λ _ → Pipper fcs → Pipper fcs × ⊤)
ptraverse-β-tiled t (_ , zero , tt) = (_, tt)
ptraverse-β-tiled t ((zero , tt) , suc zero , il , ir) =
  pup ∘' il ∘' pleft ∘' pup ∘' ir ∘' pright
ptraverse-β-tiled t ((suc zero , ol , or) , suc zero , il , ir) =
  pup ∘' ol ∘' pleft ∘' pup ∘' or ∘' pright
```

Now we are ready to build a tiled traversal as a mutumorphism. First, we need to parametrise the algebra of the tiled traversal with the tile shape.

```
ptraverse-α×β-tiled : Tree
  → Alg Tree×TreeD
  (λ _ → (Pipper fcs → Pipper fcs × ⊤)
    × (Pipper fcs → Pipper fcs × ⊤))
```

Regardless of the position in the tile, an empty tree traversal is an identity function, as before.

```
ptraverse-α×β-tiled t (_ , zero , tt) = (_, tt) , (_, tt)
```

The nodes that lie on the tile boundary do not have children in the same tile. Their contribution to the inner traversal is an identity step. However, the children that lie beyond the boundary need to be visited in the future.

The contribution to the outer traversal are steps to the children outside the tile and inner traversals of those children.

```
ptraverse-α×β-tiled t ((zero , tt) , suc zero , (il , ol) , ir , or) =
  (_ , tt) , pup ∘' il ∘' pleft ∘' pup ∘' ir ∘' pright
```

The inner traversal is built up similarly to the non-tiled case. The outer traversal needs to "lead the way" between tiles, retracking the steps made by the inner traversal.

```
ptraverse-α×β-tiled t ((suc zero , tl , tr)
    , suc zero , (il , ol) , ir , or) =
  pup ∘' ir ∘' pright ∘' pup ∘' il ∘' pleft
  , pup ∘' ol ∘' pleft ∘' pup ∘' or ∘' pright
ptraverse-tiled : Tree → Tree×Tree →
(Pipper fcs → Pipper fcs × ⊤) × (Pipper fcs → Pipper fcs × ⊤)
ptraverse-tiled t = fold Tree×TreeD (ptraverse-α×β-tiled t)
```

3.5 Do Not Measure, Demand

In the previous section, we have began to put the intuition of tiled traversals being pairs of mutually recursive inner and outer traversals (Sec. 3.3) onto formal grounds, by defining an appropriate algebra of a mutumorphism. The next step in the formalization is to show that all tiled traversals that implement functorial mappings visit each node in the tree being traversed exactly once. We will do that by defining a family of tiled traversals indexed by tile shapes. Such definition will, on one hand, serve to explain why a tiled traversal has to be a mutumorphism and, on another hand, why the choice of a tile shape does not change the semantics of a functorial map.

The mutumorphism algebra ptraverse-α×β-tiled has two projections, ptraverse-α-tiled and ptraverse-β-tiled. They both act on a pair of partial traversals, inner and outer. ptraverse-α-tiled only inspects the inner traversals. ptraverse-β-tiled inspects both because it needs to know what an inner traversal will be after descending beyond the boundary of a tile.

To review outer traversals in detail, we need to understand when to begin an outer traversal at all. At a first glance, an outer traversal begins when an inner one finishes. Given that an inner traversal in a tiled setting is implemented as an untiled traversal, we may think of the termination condition as being focused on the rightmost position in a tile. But the rightmost position in a tile is not generally the rightmost position in the complete tree. We need a way to restrict our building blocks of the step operation (pleftmost, pupdown, psubtree-root) to never consider the parts

of the tree outside the current tile. To that end, we will build a zipper type that incorporates knowledge of where tile boundaries lie.

The refinement technique showcased by McBride [43] works for properties computable as a fold, and proot is one. To demonstrate it, we will implement proot with help of the induction combinator included in Dagand's library of ornaments.

Tiles have shapes; while traversing a tile, we never visit nodes beyond its boundary, which is given by its shape. Since tile shapes are essentially trees, and we work with zippers, we need a notion of a zipper that represents trees of a given shape. When that tree shape is accompanied with an empty list of one-hole contexts, we call it the *canonical form* of a zipper. We can obtain the canonical form of a zipper by moving focus to its root, that is, "plugging in" all one-hole contexts into each other. We have implemented this operation in the previous section under the name of proot.

Algebraic ornaments Having obtained a *measure* of zippers, we can bake it into the type by refinement. In the ornamental setting such refinement is implemented as an *algebraic ornament*. Algebraic ornament is enriching the index of a datatype by adding to it the result of a fold over a value of said datatype with some appropriate algebra. When the index is enriched, there has to be evidence that values of the ornamented type satisfy the property given by the enriched index.

We define a code for the VipperO ornament because we will need it later to define type signatures over Vippers ("vector zippers"). [3]

```
VipperO : orn ⟦ PipperO ⟧orn proj₁ proj₁
VipperO = Orn.AlgebraicOrnament.Func.algOrn
  {Z} {proot-sig} ⟦ PipperO ⟧orn (λ {i} → proot-α {i})
```

The Vipper type is a family indexed by Pipper. Here we begin to see that the index of a "sized container" might not necessairy express "size and shape" of the container at the same time. When looking at vectors as lists indexed by natural numbers, the index is both determining the number of elements in the vector and the outermost constructor of the vector. For zippers, *two* separate indices are needed to provide this information to the typechecker. Luckily, dependent algebraic ornaments add one *along the way* when asked for another.

To compute the canonical form ("the size") of a Pipper value with a (dependent) fold proot, the value v itself ("the shape") has to be present in type.

[3] Agda issue #1662

```
Vipper : Pipper pln → Set
Vipper v = μ (Orn.AlgebraicOrnament.Func.algOrnD
  {Z} {proot-sig} ⟦ PipperO ⟧orn (λ {i} → proot-α {i}))
  (fcs , v)
```

Tile shapes as indices If the shape of zipper values is reflected in their types, correctness of zipper operations can established by invariants in the domain and codomain of operation types. But more information about the traversal state can be encoded in types. Remember how we presented a tiled traversal as pair of mutually recursive traversals in Section 3.2. It required maintaining not only a position in the tree, but also in the current tile, and the exit positions of all tiles connecting the current one to the tree root. It turns out we can bake in this property into the zipper type as well by using the refinement technique we have used to construct Vipper.

We will use the same algebraic ornament machinery as before. We only need to find a fold that will capture the property we need. The resulting type Tipper ("tiled zipper") will be a zipper indexed by positions in the current tile represented by Vipper.

The fold we are looking after computes a position in the current tile for each position in the complete tree. The t gives the tile shape.

```
ptile-α : ∀ {t} → Alg ⟦ PipperO ⟧orn (λ _ → Vipper t)
```

We will not use what is computed for the trees.

```
ptile-α {t}{i = pln} x = makeVipper t
ptile-α {t}{i = fcs} (_ , t) = t
```

If the list of contexts is empty, we are at the root of a tile. We can construct a tile focused at its root straight from the index.

```
ptile-α {t}{i = drv} (zero , tt) = makeVipper t
```

When we are at a leaf of a tile, we cross the tile boundary.

```
ptile-α {t}{i = drv} (suc zero , L , _ , ⟨ _ , _
  , ⟨ (zero , tt) , _ ⟩ , _ ⟩) = makeVipper t
```

If not, we move down in the tile structure, for example left. Since we work on the Vipper structure, we have to move around bits of the proof by hand.

```
ptile-α {t}{i = drv} (suc zero , L , _
  , ⟨ (f , cs) , eq₁
  , ⟨ (suc zero , l , r)
  , eq₂ , vl , vr ⟩ , vcs ⟩) =
  ⟨ (l , (λ x → cs ⟨ suc zero , x , r ⟩)))
```

```
  , subst (λ x → cs x ≡ t) (sym eq₂) eq₁
  , vl , ⟨ (suc zero , L , r , cs) , refl , vr , vcs ⟩ ⟩
```

The case for descent to the right is analogous. There are no other cases.

Tipper can now be defined as an algebraic ornament of Pipper.

```
Tipper : ∀ {t} → Vipper t → Set
Tipper {t} v = μ (Orn.AlgebraicOrnament.Func.algOrnD
  {Z} {λ _ → Vipper t} ⟦ PipperO ⟧orn
    (λ {i} → ptile-α {t}{i}))
  (fcs , v)
```

Transporting functions across ornaments We need to implement the building blocks of the step operations for the tiled setting too: tleftmost, tupdown, tsubtree-root. Their implementations do not have to be written by hand from scratch thanks to *induction lifting* combinator liftind. It *transports* functions expressed as folds across ornaments, that is, given a fold over a base type, it produces a fold over an ornamented type. More specifically, it constructs an inhabitant of the type of ornamented functions *coherent* to the corresponding base functions by construction. In the following section, we review how functions are transported across the TipperO ornament.

3.6 Naturality of tilings

By now, we have established three facts. The description of a binary tree datatype can be taken to the description of a binary tree zipper datatype by ornamentation. The binary tree zipper datatype can be constrained by refinement to contain only inhabitants corresponding to positions in a tree of a given shape. Further refinement can enrich positions to represent not only the current tree node but also the current tile in the tree. Our final step is to establish that mapping traversals over the refined datatypes are *coherent* to traversals over the base ones (Fig. 3.4).

Coherence properties have witnesses in form of *reornaments*. A reornament is a special case of an algerbraic ornament (Sec. 3.5). It is built used an *ornamental algebra*, that is, an algebra which forgets information added to a type by an ornament. Each ornament induces an ornamental algebra. Using the running example of natural numbers ornamented to lists, the ornamental algebra is a list length function.

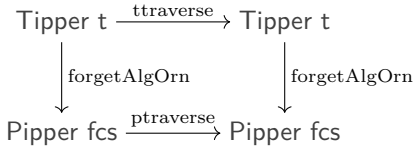


Figure 3.4. Coherence of tiled and untiled mapping traversals

The basic idea behind using reornaments as witnesses of coherence of a function is to use the base values obtained by the ornamental algebra as properties of the ornamented values. Then, a function on base values expresses an invariant. For example, when transporting addition of natural numbers to concatenation of lists, the reornament of the list type is the vector type. The coherence of the list concatenation and addition of natural numbers is encoded in vector indices, their length. These two functions are coherent because the sum of two lists being concatenated is equal to the length of the result list.

Dagand's library of ornaments provides combinators for transporting datatype constructors (lift-constructor) and pattern matching constructs (lift-case). By using them, one can transport a function expressed as a catamorphism by defining a *lifting* of the algebra it uses. A lifting of an algebra of a functor defined by ornamentation of a description code is an algebra of a functor defined by *reornamentation* of that ornament.

We shall illustrate how one component of the mapping traversal, the left child focus pleft is transported to a function on Tippers tleft. The other components, pright and pup are transported similarly.

To transport a function across an ornament, one needs to provide data for the new fields introduced by the ornament (packed together as a value of a type constructed for the ornament at hand by the Extension type constructor in the Dagand's library) and the recursive structure of the refined datatype (as constructed by the Structure type constructor).

We lift the algebra of pleft case by case. The type of the lifted algebra is constructed by the liftAlg from the ornament, the original algebra and the type of its application result.

```
tleft-α : ∀ {t} → liftAlg (TipperO {t})
  (λ {i} → pleft-α {i})
  (mk (λ k → μ⁺ (TipperO {t}) [ inv k ]× 'T))
```

The case we review is the one acting on the pair of the focused subtree and the list of one-hole contexts (represented by the fcs sort).

```
tleft-α {t}{i = (fcs , _) , ⟨ ⟨ zero , tt ⟩ , _ ⟩}
  ((_ , cs) , refl , (⟨ _ , eq , refl , tt ⟩ , tt) , vcs , tt) =
  lift-constructor (TipperO {t})
```

The new data introduced by the ornament is the proof (refl) that the position in a tile is indeed computed from the list of one-hole contexts (cs).

```
  ((makeVipper t , cs) , refl , (tt , tt))
```

The new recursive structure is determined by the recursive structures of the focused subtree and the list of one-hole contexts.

```
  (⟨ ((zero , tt) , (refl , (refl , tt))) ⟩ , vcs) tt
```

3.7 Composition of tilings

Having obtained the construction of tiled container traversal on two levels, we can use it to represent traversal on arbitrary number of levels. Traversal on two levels comes from ornamention of Vipper to Tipper. We can add more levels by repeatedly applying the Tipper ornament. This leads us to a generalization of Tipper that will reindex the base container type not with one tile shape but a vector of them. Being a coherent ornament, whose semantics preserving properties we have discussed in the previous section, this generalization establishes compositionality of tilings as well.

Chapter 4

Evaluation

In Ch. 3, we have constructed a model of a tiled tree traversal. In the current chapter, we illustrate the use of this model to implement a tree map.

4.1 Evaluation methodology

We study the performance of the four tree map implementations in C (handwritten untiled, handwritten tiled, generated untiled, generated tiled) on two hardware platforms. Each implementation/platform combination is run with input of randomly generated binary trees labeled with double-precision floats.

For each hardware platform and input data, we measure the running time of untiled and tiled implementations. Since the benchmarking is performed on a multitasking OS (Linux), we run each implementation 10 times on each generated input and pick the smallest times. To account for the shape distribution of generated trees, we generate 20 trees for each set The speedup is calculated as the ratio of average running times after 100 executions.

4.1.1 Implementations

All four implementations share the C definition of the binary labeled tree type. The definition is inspired by the sums-of-products-style algebraic datatypes. A binary tree is a tagged union of two types that represent leaves and nodes. A leaf type does not contain any data. A node type is a C struct that contains the node label (a double-precision float) and pointers to the two child trees.

All implementations use the same input data generator described in Sec. 4.1.2.

Handwritten C code The handwritten untiled tree map implementation follows the textbook in-order depth-first traversal. At each node, the traversal continues with the left child and puts the pointer to the right child on a stack. Having reached a leaf, the traversal accesses a node pointed to by the topmost element of the stack. Its label is processed (that is, in a tree map, a function is applied to it, and the result is saved as the new label), then the traversal continues with its right child. Traversal terminates when there are no more nodes accesible through pointers in the current one, and the stack is empty.

In the tiled version, the modification of the order of traversal is made. The depth-first traversal is performed inside tiles only but entering new tiles is breadth-first. To that end, instead of a stack, a queue of pointers is used. When an element of the tree is reached which has distance to the current tile root not greater than given by the *depth* parameter, pointers to its children (if there are any) are added to the end of queue. Before the maximum allowed depth is reached, the pointers to the (right) children of the node visited are put in the beginning of the queue, as before.

Generated C code The traversal model we have presented in Sec. 3.2 does not use stacks of pointers. To represent the state of traversal, we have used zippers. Since each move of focus in a zipper requires allocation either of a tree node or an one-hole context, the performance is lower than that of the handwritten implementation using stacks of pointers.

To generate the code of the zipper-style implementations, we have created a language of binary trees and tree traversals which we have used to express traversals as described in Sec. 3.2. The abstract syntax of this language is defined using the `Syntactic` [4] library. The user interface is implemented as a shallow embedding over the abstract syntax and includes constructors and pattern-matching functions for the binary tree and one-hole context types, together with the fixed point combinator. The compiler produces C code from abstract syntax with help of the `embedded-edsl` library [5].

4.1.2 Input data parameters

The input tree generator takes several parameters, allowing us to evaluate the traversal implementations on a range of tree shapes.

Tree size is the number of nodes in a tree. For each node a double precision float value is randomly generated. The leaves are unlabeled.

Branch balance determines the ratio between sizes of children of a node. A perfectly balanced tree has a branch balance equal to 1. A tree with a branch balance under 1 is biased to the right. A tree with a branch balance over 1 is biased to the left.

Branch stride controls the depth of the generated tree. It is a probability of generating a node with two node children. If set to 1, all nodes will have node children, except in subtrees with less than three nodes.

Tile size The distance from the root of a tile to its boundary. It can be seen as the maximum depth of the subtree that is covered by a tile. It serves as a proxy parameter for the number of nodes in a tile. With a tile size parameter equal to n, the number of nodes in a tile can vary from n in a degenerate tree to $2^n - 1$ in a perfectly balanced tree.

4.1.3 Hardware platforms

We evaluate the benchmarks on two systems which represent different tradeoffs in the design of the memory hierarchy.

A laptop computer equipped with an Intel Core i7-2677M processor which has 32 KB of L1 data cache, 256 KB of per-core L2 cache and 4 MB of shared L3 cache. The processor frequency is set to the maximum of 2.9 GHz. The system has 4 GB of RAM and runs an OS based on Linux kernel 4.4.31.

A Raspberry Pi board built around the Broadcom BCM2835 system-on-chip. It includes an ARM11 CPU core running at 700 MHz with 16 KB of L1 data cache and 128 KB of L2 cache. The board includes 512 MB of RAM and runs an OS based on Linux kernel 4.4.16.

4.2 Evaluation results

The handwritten, pointer-based traversal benchmarks have not shown an advantage of the tiled implementation over the untiled, with the best cases showing no speedup, and the worst cases suffering from 7% slowdown. This slowdown can be explained by the fact that in both pointer-based traversals all tree nodes are only accessed once, taking no benefit from changing memory access patterns On the other hand, in zipper traversals, nodes are accessed even after being visited when the focus is being moved

up from subtrees. Exchanging long ascent/descent paths between subtrees in an untiled implementation to shorter paths in a tiled implementation gives the benefit of temporal locality. The rest of the section only refers to the results of the generated, zipper-based benchmarks.

Figures 4.1–4.3 show speedups measured in a number of experiments. Results presented in Sections 4.2.1 and 4.2.2 are obtained for perfectly balanced trees. In Section 4.2.3, the trees are not balanced but the number of nodes and the sizes of tiles are fixed. In each experiment, we generate 20 random trees for each set of tree generation parameters. We run each implementation 10 times on each generated tree and take the smallest traversal time as the outcome to account for possible interferene We average the smallest traversal times for all generated trees to account for the variation in their shape.

4.2.1 Finding an optimal tile size

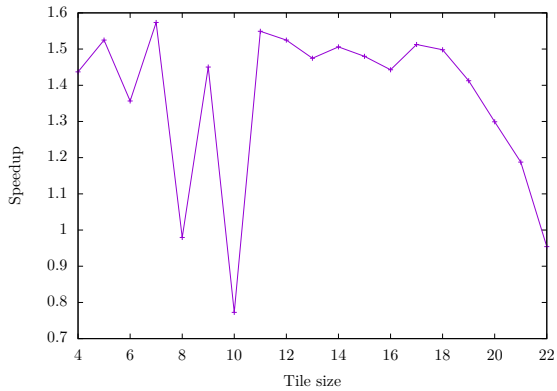
The efficacy of a tiling optimization depends on the choice of tile size. Optimal tile size is determined by cache size. In a tiled tree traversal, if tiles do not occupy the entire cache, their parent tiles are not evicted. This enables a fast move of focus from a traversed tile at the bottom of a tree to its sibling. In a perfectly balanced binary tree, one half of tiles are at the bottom, so tiling pays off.

As seen in Fig. 4.1a, the speedup on the Core i7 processor can be as high as 1.57× under a well-chosen tile size. The negative effect of the tile size crossing a cache boundary is also clearly visible on this plot. The best speedup (1.57×) is achieved under with tiles of depth 7. If the depth is increased to 8 (that is, the number of nodes in a tile is doubled), L1 cache cannot fit but L2 cache is still underutilized which causes a drop in performance. Increasing the depth to 9 (that is, doubling the tile size once more) enables efficient use of L2 cache, and increasing the depth to 9 crosses the boundary of L2 cache as well. The boundary of L3 cache is crossed starting from depth 17.

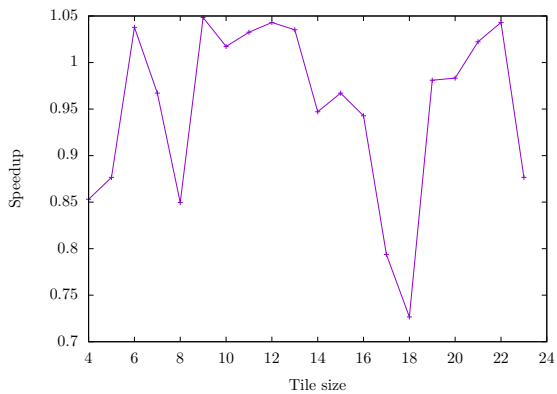
Speedup on the ARM11 processor is less pronounced (1.05×). The cache boundary effect is observed only twice (at depths 8 and 14) because the memory hierarchy has only three levels on this platform.

4.2.2 Trees of different size

If a tree is deep, the nodes closer to the root will be evicted from cache when the tiles closer to the leaves are traversed. Figure 4.2 shows this effect starting from 13000000 nodes on the Core i7 processor and from

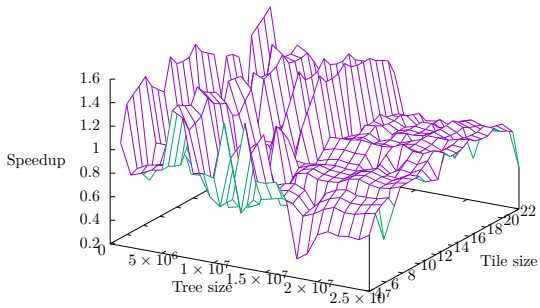


(a) Core i7, 6000000 nodes

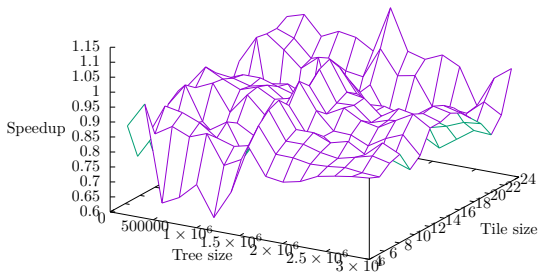


(b) ARM11, 500000 nodes

Figure 4.1. Speedups with different tile sizes



(a) Core i7



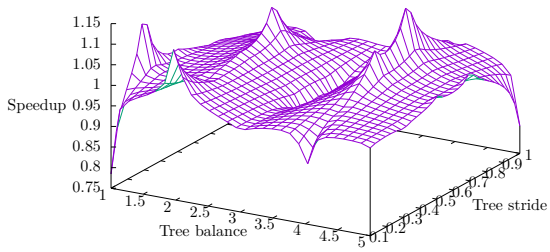
(b) ARM11

Figure 4.2. Speedups with different tree sizes

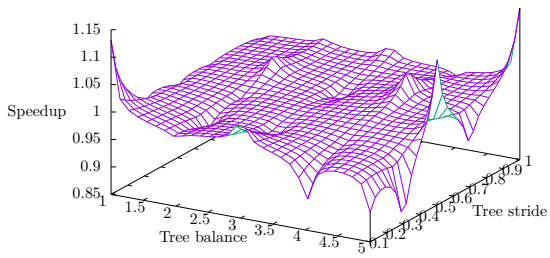
1600000 nodes on the ARM11 processor.

4.2.3 Exploring the space of possible inputs

As seen in Fig. 4.3–4.3, the speedups achieved with optimal tile depths from Section 4.2.1 are not optimal. Traversal of unbalanced trees usually requires a different tile depth parameter than traversal of perfectly balanced trees because in an unbalanced tree the number of nodes in a tile will be smaller if the depth is kept the same as with balanced trees. To achieve comparable performance improvement with unbalanced trees, the tile depth parameter has to be larger than with balanced trees.



(a) Core i7, 6000000 nodes, tile depth 17



(b) ARM11, 500000 nodes, tile depth 12

Figure 4.3. Speedups with varying tree generation parameters

Chapter 5

Related work

This thesis stands at the intersection of reasoning about programs in type theory and low-level compiler optimization. In the current section, we review existing literature that has strong connection to the present work.

5.1 Traversal splicing

Jo and Kulkarni have produced a large body of work on enhancing locality of access in traversals of pointer-linked structures. In [34], they note that optimization of regular data structure traversals relies on regular memory allocation, that is, on *spatial* locality. Traversal of irregularly allocated data structures, such as trees and graphs, can be optimized by exploiting *temporal* locality, that is, changing traversal order. To that end, they introduce a technique they call *point blocking*. It changes a depth-first tree traversal into a combination of a depth-first and a breadth-first traversals. The technique is developed for Java programs and works only on recursive traversals of recursive data structures with particular structure, which it needs to identify. The programs that can be optimized with this technique compute *point* values as the result of tree traversals. A point value is a result returned by a subcomputation that traverses parts of a recursive data structure. Elements of a data structure might be accessed many times during computation of many points. Points with overlapping traversals can be identified by an algorithm-specific procedure and then assigned to *blocks*. Traversals to compute points within one block are performed depth-first, but traversal of blocks is done by *levels*, which makes it breadth-first.

In [35], Jo and Kulkarni improve the point blocking technique and rebrand it as *traversal splicing*. The improved technique removes the requirement of algorithm-specific sorting by partitioning traversal space instead of point space. For example, in ray tracing algorithms, objects in a scene are stored in a bounding volume tree [36]. Finding intersections of each ray with scene objects requires a recursive traversal of the tree.

Traversals for different rays might visit the same subset of tree nodes. For each point, a partial traversal involving a particular subset of tree nodes is scheduled together with other partial traversals of that subset, thus exploiting temporal locality. This change to the traversal order requires more sophisticated bookkeeping, as the state of each point traversal must be maintained. Jo and Kulkarni describe how to store traversal state in the tree nodes themselves.

In [33], traversal splicing is used as the scheduling stage for vectorization.

5.2 Vectorization in Haskell

A compiler aware of the target machine's vector instructions can save a bit of thinking about cache behavior on the programmer's side. A vector unit works on data several machine words long which have to be aligned. This guarantees the occupation of one or more cache lines by data that is accessed together.

Vectorization can be seen as transforming a loop into a loop nest where the inner loop has as many iterations as the number of words a vector unit can process.

In [46], Petersen et al. extend the `Core` internal representation of GHC with array operations that read and write several array elements at once as vectors. These operations are commonly known as *gathering* and *scattering*. This work does not rely on the properties of the high-level datatype operations and requires a dependency analysis, although the authors use array immutability to simplify their dependency analysis.

5.3 *Strategies* in Haskell

Memory hierarchy use optimizations, such as cache-aware compiler transformations and cache-oblivious algorithms, build on the common idea of representing a computation as a collection of subcomputation. Another manifestation of this idea is the use of higher-order functional programming to separate algorithms from the specifics of their execution on a particular machine. To our knowledge, there is no previous work that applies it to cache optimizations. We should note that identification of subcomputations and deciding on order of their execution are two different problems, and our work is mostly concerned with the former.

In the context of parallel programming, a library of *strategies* was proposed in [52] to separate denotational semantics of a program from its

operational semantics. Strategies are higher-order functions that capture patterns of parallel evaluation. The programmer can `par` and `seq` combinators to annotate a program. The meaning of the `par` combinator is to provide a hint to the run-time scheduler that a computation can be evaluated in its own thread. The `seq` combinator introduces an explicit evaluation order for two computations that could otherwise be executed differently in a lazy language. Strategies build on these two combinators to simplify the task of annotation by capturing common parallel patterns and enabling composition.

These combinators enjoy the run-time support of the Glasgow Haskell Compiler [40]. They are used to make the thread scheduler aware of coarse control dependencies and exploit parallelism dynamically without suffering much overhead. Although explicit sequencing introduced by strategies can remove some of the overhead of laziness, a Haskell programmer still does not enjoy control over memory allocation and access patterns.

5.4 Program calculation

Identification of subcomputations was also studied from the category-theoretical perspective. Our work builds on the fact that functional programs are mathematical objects equipped with numerous algebraic laws. The bulk of these laws comes from giving datatypes semantics as initial (terminal) (co)algebras and their morphisms. [27] Use of laws to transform programs was given a name of *program calculation* by Bird [8].

To divide a computation into independent subcomputations, Hu et al. [29] introduce *J-homomorphisms*, functions on lists that distribute over list concatenation. This abstraction can be used to identify subcomputations in a program that can be performed in parallel. Then they introduce calculational rules to translate a program expressed as a fusion of list homomorphisms to a J-homomorphism.

5.5 Data layout polymorphism

Shinkarev's thesis [49] introduces a type system with data layout polymorphism into the Single Assignment C language (SAC). A data layout is a mapping of a data structure into flat memory. With type inference, this type system is capable of generating all valid data layouts for a well-typed program. His thesis studies transformations of array traversals under changed layouts which are represented in the types as tile sizes. In

comparison, our work considers tree-like data structures with information (tile sizes) that impacts the order of their traversals also encoded in the types.

The ultimate purpose of Shinkarev's work is to facilitate SIMD vectorization. The vector types in SAC are annotated with data layouts. A n-dimensional vector has n possible layouts in a scheme where an additional dimension of a constant length is introduced and one of original dimensions is transformed by dividing its length by that constant length. The constant is chosen to match the width of the SIMD units.

The purpose of type layout annotations is to enable type inference that can identify layout choices suitable for vectorization. In Shinkarev's work, the type inference algorithm is a top-down traversal of a program applying a layout rule to the type of every term, thus propagating layout constraints. Compositionality of data layouts, while crucial for reasoning about program behavior under a multilevel memory hierarchy, is not yet part of the described type system.

Chapter 6

Conclusion and future work

We have shown an approach to decomposing a traversal of a data structure into smaller traversals. Such decomposition improves temporal locality of access which leads to a more efficient use of cache. Locality optimizations have been long used for imperative array-processing programs. There are much less studies of this class of optimizations in the setting of functional programs operating on tree-like data structures.

McBride has observed in [42] that the key to transformation of a traversal order is representing traversal state as data. We build on that observation by using Dagand's function ornaments (Ch. 9 in [22]) to capture the relationship between tiled and untiled traversal orders. That relationship is coherence between functions over base and ornamented datatypes where the ornament extends the traversal state with the distance from the current position in a tile to the tile root. Using Dagand's machinery for transporting functions across ornaments, we obtain a tiled tree map traversal implementation from an untiled one.

We have demonstrated that tiling transformations are a direction to be explored in order to improve performance of recursive traversals. This work is not an end in itself but a foundation for a larger implementation that could be a basis for a larger study similar to [35]. There are steps to be made in several directions.

6.1 Structured graphs

The majority of work on tiling has been done in the context of high-performance array computations. Changing the order of array traversals often requires a dependency analysis. An array can be represented as a cyclic graph where array elements are vertices and there is an edge between each pair of neighbouring array elements. Oliveira and Cook [45] introduce an encoding for cyclic graphs based on parametric higher-order abstract syntax. This encoding can be used to represent arrays as algebraic datatypes. Then our tiling approach can be applied to arrays to

obtain a tiled array traversal that is correct by construction, without the need for dependency analysis.

6.2 Zoo of morphisms

We have only demonstrated tiling for functorial maps but other recursive schemes can benefit from it as well. A tiled recursive scheme is essentially a mutumorphism which uses an additional algebra for each level of tiling to keep track of a position in the correponding tile. Hinze, Wu and Gibbons [28] unite numerous recursive schemes under a single category-theoretical framework. This framework can be used to derive tiled versions of all of them.

6.3 Feldspar

To measure the performance effect of the proposed tiling transformation, we have implemented a small EDSL of trees and tree traversals. We used the `Syntactic` and `imperative-edsl` libraries [4, 5] to implement its compiler. These libraries were originally developed to serve as foundation of the Feldspar language [6]. Our implementation work can be continued to extend Feldspar with algebraic datatypes, recursive schemes and tiling and implement larger benchmarks in it.

Appendices

Appendix A

The language of tree traversals

A.1 Syntax

```
{-# LANGUAGE DeriveFunctor          #-}
{-# LANGUAGE FlexibleContexts       #-}
{-# LANGUAGE FlexibleInstances      #-}
{-# LANGUAGE GADTs                  #-}
{-# LANGUAGE PartialTypeSignatures #-}
{-# LANGUAGE ScopedTypeVariables    #-}
{-# LANGUAGE StandaloneDeriving     #-}
{-# LANGUAGE TemplateHaskell        #-}
{-# LANGUAGE TypeFamilies           #-}
{-# LANGUAGE TypeOperators          #-}
{-# LANGUAGE UndecidableInstances   #-}

module Frontend where

import          Data.Typeable
import          Language.Syntactic
import          Language.Syntactic.Functional
import          Language.Syntactic.Functional.Tuple
import          Language.Syntactic.Sugar.BindingTyped ()
import          Language.Syntactic.Sugar.TupleTyped   ()
import          Language.Syntactic.TH
import          Prelude                         hiding (curry, uncurry)

-- INTERNAL REPRESENTATION (DEEP EMBEDDING)

-- the standard definition of binary trees and their lists of one-hole contexts
data Tree a = Leaf | Node (Tree a) a (Tree a)
deriving instance (Show a)     => Show (Tree a)
deriving instance (Typeable a) => Typeable (Tree a)
```

```
data Context a = Plug | Hole Bool a (Tree a) (Context a)
deriving instance (Show a)     => Show (Context a)
deriving instance (Typeable a) => Typeable (Context a)

-- a shortcut borrowed from Emil Axelsson's examples
class    (Typeable a, Show a) => Type a
instance (Typeable a, Show a) => Type a

data Pln a where
   DLeaf :: Pln (Full (Tree a))
   DNode :: Type a => Pln (Tree a :-> a :-> Tree a :-> Full (Tree a))
   MatchPln :: (Type a, Type b)
            => Pln (Tree a :-> b
                   :-> (Tree a -> a -> Tree a -> b) :-> Full b)

deriveSymbol ''Pln
deriveRender id ''Pln
instance StringTree Pln

instance Eval Pln where
   evalSym DLeaf = Leaf
   evalSym DNode = Node

data Drv a where
   DPlug :: Type a => Drv (Full a)
   DHole :: Type a => Drv (Bool :-> a
                       :-> Tree a :-> Context a :-> Full (Context a))
   MatchDrv :: (Type a, Type b) =>
            Drv (Context a :-> b
                :-> (Bool -> a -> Tree a -> Context a -> b) :-> Full b)

deriveSymbol ''Drv
deriveRender id ''Drv
instance StringTree Drv

data Fix a where
   Fix :: Type a => Fix (((a -> a) -> (a -> a)) :-> a :-> Full a)

deriveSymbol ''Fix
```

```
deriveRender id ''Fix
instance StringTree Fix

type DemoDomain = Typed (Construct :+: BindingT
                  :+: Tuple :+: Pln :+: Drv :+: Fix)

newtype Demo a = Demo { unDemo :: ASTF DemoDomain a }

-- sugared syntax to AST and back
instance Type a => Syntactic (Demo a)
  where
    type Domain (Demo a)   = DemoDomain
    type Internal (Demo a) = a
    desugar = unDemo
    sugar   = Demo

-- another handy shortcut from Emil
class    (Syntactic a,
          Domain a ~ DemoDomain, Type (Internal a)) => Syntax a
instance (Syntactic a,
          Domain a ~ DemoDomain, Type (Internal a)) => Syntax a

-- USER INTERFACE (SHALLOW EMBEDDING)

-- a small "prelude"
value :: Syntax a => Internal a -> a
value a = sugar $ injT $ Construct (show a) a

pair :: (Type a, Type b) => Demo a -> Demo b -> Demo (a, b)
pair = sugarSymTyped Pair

uncurry :: (Syntax c, Type a, Type b) =>
  (Demo a -> Demo b -> c) -> Demo (a, b) -> c
uncurry f p = f (sugarSymTyped Fst p) (sugarSymTyped Snd p)

false :: Demo Bool
false = value False

true :: Demo Bool
true = value True
```

```
(?) :: forall a . Syntax a => Demo Bool -> (a,a) -> a
c ? (t,f) = sugarSymTyped sym c t f
  where
    sym :: Construct (Bool :-> Internal a :-> Internal a
                      :-> Full (Internal a))
    sym = Construct "cond" (\c t f -> if c then t else f)

-- now the actual things begin
leaf :: Type a => Demo (Tree a)
leaf = sugarSymTyped DLeaf

node :: Type a => Demo (Tree a) -> Demo a -> Demo (Tree a)
  -> Demo (Tree a)
node = sugarSymTyped DNode

matchtree :: (Syntax b, Type a)
          => Demo (Tree a) -> b
          -> (Demo (Tree a) -> Demo a -> Demo (Tree a)-> b) -> b
matchtree = sugarSymTyped MatchPln

plug :: Demo ()
plug = sugarSymTyped DPlug

hole :: Type a => Demo Bool -> Demo a -> Demo (Tree a)
     -> (Demo (Context a)) -> (Demo (Context a))
hole = sugarSymTyped DHole

matchcontext :: (Syntax b, Type a) => Demo (Context a)
  -> b -> (Demo Bool -> Demo a -> Demo (Tree a)
  -> Demo (Context a) -> b) -> b
matchcontext = sugarSymTyped MatchDrv

fix :: Syntax a => ((a -> a) -> (a -> a)) -> a -> a
fix = sugarSymTyped Fix

-- An example expression (a tree map)
maptree :: Type a => (Demo a -> Demo a)
        -> Demo (Tree a) -> Demo (Tree a)
maptree f = fix (\mf t -> matchtree t leaf
```

```
(\l x r -> node (mf l) (f x) (mf r)))
```

A.2 Semantics

```
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE GADTs            #-}
{-# LANGUAGE RankNTypes       #-}
{-# LANGUAGE TypeFamilies     #-}
{-# LANGUAGE TypeOperators    #-}

module Backend where

import          Control.Monad.Reader
import          Data.Map                        (Map, empty, insert,
                                                 lookup)
import          Unsafe.Coerce
import          Frontend
import          Language.Embedded.Backend.C
import          Language.Embedded.CExp          (CExp (..), evalCExp)

import          Language.Embedded.Imperative
import          Language.Syntactic              ((:&:) (..), (:->),
                                                 AST (..), ASTF, Args (..),
                                                 DenResult (..), Full,
                                                 Project (..),
                                                 Syntactic (..), renderSym,
                                                 simpleMatch)
import          Language.Syntactic.Functional
import          Language.Syntactic.Functional.Tuple

import          Prelude                         (Maybe (..), Show (..),
                                                 error, show, undefined, flip,
                                                 ($), (++), (.))

-- TRANSLATION TO C (SEMANTICS)

type CMD = RefCMD :+: ControlCMD :+: C_CMD

type Target = ReaderT Env (Program CMD (Param2 CExp CType))
```

```
translate :: (Type a, CType a) => Demo a -> Target (CExp a)
translate = simpleMatch go . unDemo where
  go :: CType (DenResult sig) =>
    DemoDomain sig -> Args (AST DemoDomain) sig
                   -> Target (CExp (DenResult sig))
  -- Binding
  go var Nil | Just (VarT v) <- prj var = lookAlias v
  -- Pair
  go pair (a :* b :* Nil)
    | Just Pair <- prj pair = Two <$> go a <*> go b
  -- Pln
  go leaf Nil                 | Just DLeaf <- prj leaf =
                                lift . callFun "new_leaf" $ []
  go node (l :* x :* r :* Nil) | Just DNode <- prj node =
                                lift . callFun "new_node" $ []
  go matchpln (tree :* caseLeaf :* (lamL :$ (lamX :$ (lamR :$ body))) :* Nil)
    | Just MatchPln  <- prj matchpln
    , Just (LamT lv) <- prj lamL
    , Just (LamT xv) <- prj lamX
    , Just (LamT rv) <- prj lamR
    = do ReaderT $ \env -> do
         res <- newRef
         isLeaf <- callFun "is_leaf" [caseLeaf]
         l <- callFun "node_l" [lv]
         x <- callFun "node_x" [xv]
         r <- callFun "node_r" [rv]
         iff isLeaf
           (flip runReaderT env (translate (sugar caseLeaf)) >> break)
           (flip runReaderT env (localAlias lv l $
                                 localAlias xv x $
                                 localAlias rv r $
                                 translate (sugar body)) >>= setRef res)
         getRef $ res

  -- Drv
  go plug Nil                   | Just DPlug  <- prj plug =
                                  lift $ callFun "new_plug"
  go hole (h :* u :* c :* cs :* Nil) | Just DHole  <- prj hole =
                                  lift $ callFun "new_node"
  go matchdrv (ctx :* casePlug
```

```
          :* (lamH :$ (lamX :$ (lamU :$ (lamC :$ body)))) :* Nil)
  | Just MatchDrv <- prj matchdrv
  , Just (LamT hv) <- prj lamH
  , Just (LamT xv) <- prj lamX
  , Just (LamT uv) <- prj lamU
  , Just (LamT cv) <- prj lamC
  = do ReaderT $ \env -> do
       res <- newRef
       isPlug <- callFun "is_plug" [casePlug]
       h <- callFun "ctx_h" [hv]
       x <- callFun "ctx_x" [xv]
       u <- callFun "ctx_u" [uv]
       c <- callFun "ctx_c" [cv]
       iff isLeaf
         (flip runReaderT env (translate (sugar caseLeaf)) >> break)
         (flip runReaderT env (localAlias lv h $
                               localAlias xv x $
                               localAlias xv u $
                               localAlias rv v $
                               translate (sugar body)) >>= setRef res)
       getRef $ res

-- Fix
go fix ((lamf :$ fbody) :* a :* Nil)
  | Just Fix      <- prj fix
  , Just (LamT fv) <- prj lamf
  = do ReaderT $ \env -> do
       acc <- initRef =<< (flip runReaderT env $ translate (sugar a))
       while (flip runReaderT env $ translate true) $ translate fbody
       getRef $ acc

go s _      = error $ "translate: no handling of symbol " ++ renderSym s

-- translation environment, borrowed from RAW-Feldspar

data CExp' where
  CExp' :: CExp a -> CExp'

newtype Env = Env { unEnv :: Map Name CExp' }
```

```
localAlias :: MonadReader Env m => Name -> CExp a -> m b -> m b
localAlias v e = local (\env -> Env (insert v (CExp' e) (unEnv env)))

lookAlias :: MonadReader Env m => Name -> m (CExp a)
lookAlias v = do
  env <- asks unEnv
  return $ case lookup v env of
    Nothing -> error $ "lookAlias: variable " ++ show v ++ " not in scope"
    Just (CExp' e) -> unsafeCoerce e
```

Appendix B

Benchmarks

We have edited the generated benchmarks for readability while leaving their structure unchanged. All benchmarks use the following definitions of the tree datatype and the tree generation function.

```
enum TREE_TYPE { LEAF, NODE };
enum TREE_EDGE { LEFTMOST, RIGHTMOST, ROOT, NONE };

typedef struct tree {
    enum TREE_TYPE type;
    enum TREE_EDGE edge;
    int depth;
    union {
        void *leaf;
        struct {
            struct tree *l;
            double       x;
            struct tree *r;
        } node;
    } value;
} tree_t;

tree_t* gen_tree_unbalanced(int size, int ratio, double bias) {
    tree_t* tree = malloc(sizeof(tree_t));
    double skip = (double) rand() / (double) RAND_MAX;
    double twist = (double) rand() / (double) RAND_MAX;
    if (size == 0) {
        tree->type = LEAF;
        tree->value.leaf = NULL;
    } else {
        if (skip < bias) {
            tree->type = NODE;
            tree->value.node.x = rand();
```

```
            if (twist > 0.5) {
                tree->value.node.l = gen_tree_unbalanced(0, ratio, bias);
                tree->value.node.r = gen_tree_unbalanced(size, ratio, bias);
            } else {
                tree->value.node.l = gen_tree_unbalanced(size, ratio, bias);
                tree->value.node.r = gen_tree_unbalanced(0, ratio, bias);
            }
        } else {
            int size_l, size_r;
            if ((size-1) % ratio == 0) {
                size_l = (size-1) / ratio;
                size_r = (size-1) - size_l;
            } else {
                size_l = ((size-1) - (size-1) % ratio) / ratio;
                size_r = (size-1) - size_l;
            }
            tree->type = NODE;
            tree->value.node.x = rand();
            if (twist > 0.5) {
                tree->value.node.l = gen_tree_unbalanced(size_l, ratio, bias);
                tree->value.node.r = gen_tree_unbalanced(size_r, ratio, bias);
            } else {
                tree->value.node.r = gen_tree_unbalanced(size_r, ratio, bias);
                tree->value.node.l = gen_tree_unbalanced(size_l, ratio, bias);
            }
        }
    }
    return tree;
}
```

B.1 Untiled tree traversal

B.1.1 A high-level implementation

This implementation follows one described in Sec. 3.2.

```
module MapZipperUntiled where

import      Frontend
import      Prelude  hiding (curry, uncurry)
```

```
type Zipper a = (Tree a, Context a)

update :: Type a => (Demo a -> Demo a)
  -> Demo (Zipper a) -> Demo (Zipper a)
update f = uncurry (\t c -> matchtree t (pair t c)
                        (\l x r -> pair (node l (f x) r) c))

left :: Type a => Demo (Zipper a) -> Demo (Zipper a)
left = uncurry (\t c -> matchtree t (pair t c)
                                (\l x r -> pair l (hole false x r c)))

right :: Type a => Demo (Zipper a) -> Demo (Zipper a)
right = uncurry (\t c -> matchtree t (pair t c)
                                 (\l x r -> pair r (hole true x l c)))

up :: Type a => Demo (Zipper a) -> Demo (Zipper a)
up = uncurry (\t c -> matchcontext c (pair t c)
                      (\h x u cs -> pair (h ? (node t x u, node u x t)) cs))

subtreeRoot :: Type a => Demo (Zipper a) -> Demo (Zipper a)
subtreeRoot = fix (\sr -> uncurry (\t c ->
   matchcontext c (pair t c) (\h _ _ _ -> h ? (pair t c , sr . up $ pair t c))))

mapzipper :: Type a => (Demo a -> Demo a)
  -> Demo (Zipper a) -> Demo (Zipper a)
mapzipper f = fix (\mz -> uncurry (\t c ->
   matchtree t (matchcontext c (pair t c) -- done
         -- ascent, update, descent right
         (\_ _ _ _ -> mz . right . update f . subtreeRoot $ pair t c))
       -- descent left
       (\_ _ _ -> mz . left $ pair t c)))
```

B.1.2 Generated C code

```
void map_zipper_untiled(double (*f) (double), tree_t* tree) {
    stack_t* context = NULL;
    stack_t* txetnoc = NULL;
    tree->edge = ROOT;
    while (1) {
        if (tree->type == LEAF) {
            if (context == NULL) { // done
```

```
                break;
            } else {
                // ascent, update, descent right
                if (tree->edge == RIGHTMOST)
                    break;
                subtree_root(&tree, &context, &txetnoc, LEFT);
                up(&tree, &context, &txetnoc);
                tree->value.node.x = f(tree->value.node.x);
                count_nodes++;
                right(&tree, &context, &txetnoc);
            }
        } else {
            // descent left
            left(&tree, &context, &txetnoc);
        }
    }
    while (context)
      up(&tree, &context, &txetnoc);
    assert(!context);
    return;
}
```

B.1.3 Handwritten C code

```
void map_tree_untiled(double (*f) (double), tree_t* tree) {
    stack_t* s_head = NULL;
    stack_t* s_tail = NULL;
    while (tree->type == NODE || s_head != NULL) {
        if (tree->type == NODE) {
            push(&s_head, &s_tail, tree);
            tree = tree->value.node.l;
        } else {
            tree = pop(&s_head, &s_tail);
            tree->value.node.x = f(tree->value.node.x);
            count++;
            tree = tree->value.node.r;
        }
    }
}
```

B.2 Tiled tree traversal

B.2.1 A high-level implementation

This implementation follows one described in Sec. 3.3.

```
module MapZipperTiled where

import      Frontend
import      MapZipperUntiled
import      Prelude      hiding (curry, uncurry)

type Tipper a = (Zipper a, Zipper a)

pairstep :: Type a => (Demo (Zipper a) -> Demo (Zipper a))
  -> Demo (Tipper a) -> Demo (Tipper a)
pairstep f = uncurry (\tile tree -> pair (f tile) (f tree))

subtreeRootIn :: Type a => Demo (Tipper a) -> Demo (Tipper a)
subtreeRootIn = fix (\sr -> uncurry (\tile tree -> uncurry (\t c ->
    matchcontext c (pair (pair t c) tree)
    (\h _ _ _ -> h ? (pair (pair t c) tree
                    , sr . pairstep up $ pair (pair t c) tree)))))

subtreeRootOut :: Type a => Demo (Tipper a) -> Demo (Tipper a)
subtreeRootOut = fix (\sr -> uncurry (\tile tree -> uncurry (\t c ->
    matchcontext c (pair (pair t c) tree)
    (\h _ _ _ -> h ? (sr . pairstep up $ pair (pair t c) tree)
                    , pair (pair t c) tree))))

maptipperInner :: Type a => (Demo a -> Demo a)
  -> Demo (Tipper a) -> Demo (Tipper a)
maptipperInner f = fix (\mz -> uncurry (\tile tree -> uncurry (\t c ->
    matchtree t (matchcontext c (maptipperOuter f (pair (pair t c) tree))
          (\_ _ _ _ ->
             mz . right . update f . subtreeRootIn $ pair (pair t c) tree))
        (\_ _ _ -> mz . left $ pair (pair t c) tree))))

maptipperOuter :: Type a => (Demo a -> Demo a)
  -> Demo (Tipper a) -> Demo (Tipper a)
maptipperOuter f = fix (\mz -> uncurry (\tile tree -> uncurry (\t c ->
```

```
    matchtree t (matchcontext c (maptipperOuter f (pair (pair t c) tree))
        uncurry (\t2 c2 -> matchtree t2 (matchcontext c2
              (maptipperInner f (pair (pair t c) (pair t2 c2)))
          (\_ _ _ _ -> mz . left . update f
            . subtreeRootOut $ pair (pair t c) (pair t2 c2))
        (\_ _ _ -> mz . right $ pair t c)))))
```

B.2.2 Generated C code

```
void map_zipper_tiled(double (*f) (double), tree_t* tree, int depth) {
    stack_t* context = NULL;
    stack_t* txetnoc = NULL;
    tree->depth = 0;
    tree->edge = ROOT;
    while (1) {
        while (1) {
            assert(tree->depth <= depth);
            if (tree->type == LEAF) {
                if (context == NULL || tree->depth == 0) {
                    break;
                } else {
                    if (tree->edge == RIGHTMOST)
                        break;
                    subtree_root(&tree, &context, &txetnoc, LEFT);
                    up(&tree, &context, &txetnoc);
                    tree->value.node.x = f(tree->value.node.x);
                    count_nodes++;
                    right(&tree, &context, &txetnoc);
                }
            } else {
                if (tree->depth < depth)
                    left(&tree, &context, &txetnoc);
                else {
                    assert(tree->depth == depth);
                    tree->value.node.x = f(tree->value.node.x);
                    count_nodes++;
                    if (tree->edge == RIGHTMOST)
                        break;
                    subtree_root(&tree, &context, &txetnoc, LEFT);
                    up(&tree, &context, &txetnoc);
                    tree->value.node.x = f(tree->value.node.x);
```

```
            count_nodes++;
            right(&tree, &context, &txetnoc);
        }
    }
}
assert(tree->depth <= depth);
count_tiles++;
if (tree->type == LEAF) { // no tile ahead, go back
    // get out of the current tile
    while (tree->depth != 0)
        up(&tree, &context, &txetnoc);
    while (1) {
        assert(tree->depth <= depth);
        // find a pivot to an unvisited tile
            subtree_root(&tree, &context, &txetnoc, RIGHT);
            if (context == NULL) // tree done
                return;
        // pivot
            up(&tree, &context, &txetnoc);
            left(&tree, &context, &txetnoc);
        // can it be a new tile?
            if (tree->depth > depth) {
                tree->depth = 0;
                tree->edge = ROOT;
            break;
            }
        // go to an unvisited tile
        while (tree->depth < depth && tree->type == NODE)
            right(&tree, &context, &txetnoc);
        // tile ahead, go there
        if (tree->depth == depth && tree->type == NODE) {
            right(&tree, &context, &txetnoc);
            tree->depth = 0;
            tree->edge = ROOT;
            break;
        }
    }
} else { // tile ahead, go there
    assert(tree->depth == depth);
    right(&tree, &context, &txetnoc);
```

```
            tree->depth = 0;
            tree->edge = ROOT;
        }
        assert(tree->depth == 0);
        assert(tree->edge == ROOT);
    }
    assert(!context);
    return;
}
```

B.2.3 Handwritten C code

```
void map_tree_tiled_depth(double (*f) (double), tree_t* tree, int depth) {
    stack_t* s_head = NULL;
    stack_t* s_tail = NULL;
    tree->depth = 0;
    while (tree->type == NODE || s_head != NULL) {
        if (tree->type == NODE) {
            if (tree->depth < depth) {
                push(&s_head, &s_tail, tree);
                tree = tree->value.node.l;
                tree->depth = ((tree_t*) (s_head->x))->depth + 1;
            } else {
                // remember work not done
                if (tree->value.node.l->type == NODE) {
                    tree->value.node.l->depth = 0;
                    append(&s_head, &s_tail, tree->value.node.l);
                }
                if (tree->value.node.r->type == NODE) {
                    tree->value.node.r->depth = 0;
                    append(&s_head, &s_tail, tree->value.node.r);
                }
                // continue in the current tile
                tree->value.node.x = f(tree->value.node.x);
                count++;

                tree = pop(&s_head, &s_tail);
                tree->value.node.x = f(tree->value.node.x);
                count++;
                tree = tree->value.node.r;
            }
```

```
        } else {
            tree = pop(&s_head, &s_tail);
            tree->value.node.x = f(tree->value.node.x);
            count++;
            tree = tree->value.node.r;
        }
    }
}
```

Bibliography

[1] Michael Abbott. *Categories of containers*. PhD thesis, University of Leicester, 2003.
— One citation on page 7

[2] Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Derivatives of containers. In *Typed Lambda Calculi and Applications*, pages 16–30. Springer, 2003.
— One citation on page 26

[3] Robert Atkey, Patricia Johann, and Neil Ghani. Refining inductive types. *Logical Methods in Computer Science*, 7(2:9), 2012.
— 2 citations on pages 21 and 34

[4] Emil Axelsson. A generic abstract syntax model for embedded languages. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, pages 323–334. ACM, 2012.
— 2 citations on pages 50 and 62

[5] Emil Axelsson. Compilation as a typed edsl-to-edsl transformation. *arXiv preprint arXiv:1603.08865*, 2016.
— 2 citations on pages 50 and 62

[6] Emil Axelsson, Koen Claessen, Gergely Dévai, Zoltán Horváth, Karin Keijzer, Bo Lyckegård, Anders Persson, Mary Sheeran, Josef Svenningsson, and András Vajdax. Feldspar: A domain specific language for digital signal processing algorithms. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 169–178. IEEE, 2010.
— 2 citations on pages 1 and 62

[7] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages

7–16, 2004.
— One citation on page 3

[8] Richard S. Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126, 1989.
— 2 citations on pages 7 and 59

[9] Guy E Blelloch, Rezaul A Chowdhury, Phillip B Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 501–510. Society for Industrial and Applied Mathematics, 2008.
— One citation on page 4

[10] Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, and Harsha Vardhan Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 355–366. ACM, 2011.
— One citation on page 4

[11] Guy E Blelloch and Robert Harper. Cache and i/o efficent functional algorithms. In *ACM SIGPLAN Notices*, volume 48, pages 39–50. ACM, 2013.
— One citation on page 5

[12] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 101–113, 2008.
— One citation on page 2

[13] Silas Boyd-Wickizer, Robert Morris, M Frans Kaashoek, et al. Reinventing scheduling for multicore systems. In *HotOS*, 2009.
— One citation on page 2

[14] Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In *International Workshop on Types for Proofs and Programs*, pages 115–129. Springer, 2003.
— One citation on page 20

[15] G. S. Brodal and R. Fagerberg. On the limits of cache-obliviousness. In *Proceedings of the thirty-fifth annual ACM symposium on Theory*

of computing, STOC '03, pages 307–315, 2003.
— One citation on page 4

[16] M. M. T. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the 6th workshop on Declarative aspects of multicore programming*, DAMP '11, pages 3–14, 2011.
— One citation on page 1

[17] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data parallel haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, DAMP '07, pages 10–18, 2007.
— One citation on page 1

[18] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of the 13th international conference on Supercomputing*, ICS '99, pages 444–453, 1999.
— One citation on page 3

[19] T.M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 1–12, 1999.
— One citation on page 3

[20] Trishul M Chilimbi and James R Larus. Using generational garbage collection to implement cache-conscious data placement. *ACM SIGPLAN Notices*, 34(3):37–48, 1999.
— One citation on page 4

[21] P.-E. Dagand and C. McBride. Transporting functions across ornaments. *Journal of Functional Programming*, 24(2-3):316–383, 2014.
— 2 citations on pages 10 and 34

[22] Pierre-Evariste Dagand. *A cosmology of datatypes: reusability and dependent types*. PhD thesis, University of Strathclyde, 2013.
— 4 citations on pages 7, 10, 26, and 61

[23] Alexandra Fedorova, Margo I Seltzer, Christopher A Small, and Daniel Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. Technical Report, 2005.
— One citation on page 2

[24] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 285–297, 1999.
— One citation on page 5

[25] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34:261–317, 2006.
— One citation on page 2

[26] Ryu Hasegawa. Two applications of analytic functors. *Theoretical Computer Science*, 272(1):113–175, 2002.
— One citation on page 8

[27] Ralf Hinze. Adjoint folds and unfolds—an extended study. *Science of Computer Programming*, 78(11):2108–2159, 2013.
— One citation on page 59

[28] Ralf Hinze, Nicolas Wu, and Jeremy Gibbons. Unifying structured recursion schemes. In *ACM SIGPLAN Notices*, volume 48, pages 209–220. ACM, 2013.
— 2 citations on pages 7 and 62

[29] Zhenjiang Hu, Tetsuo Yokoyama, and Masato Takeichi. Program optimizations and transformations in calculation form. In *Generative and Transformational Techniques in Software Engineering*, pages 144–168. Springer, 2006.
— One citation on page 59

[30] Xianglong Huang, Stephen M Blackburn, Kathryn S McKinley, J Eliot B Moss, Zhenlin Wang, and Perry Cheng. The garbage collection advantage: improving program locality. *ACM SIGPLAN Notices*, 39(10):69–80, 2004.
— One citation on page 4

[31] Gérard Huet. The zipper. *Journal of functional programming*, 7(05):549–554, 1997.
— One citation on page 8

[32] C Barry Jay and J Robin B Cockett. Shapely types and shape polymorphism. In *European Symposium on Programming*, pages 302–316. Springer, 1994.
— One citation on page 8

[33] Youngjoon Jo, Michael Goldfarb, and Milind Kulkarni. Automatic vectorization of tree traversals. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 363–374. IEEE Press, 2013.
— One citation on page 58

[34] Youngjoon Jo and Milind Kulkarni. Enhancing locality for recursive traversals of recursive structures. *ACM SIGPLAN Notices*, 46:463–482, 2011.
— One citation on page 57

[35] Youngjoon Jo and Milind Kulkarni. Automatically enhancing locality for tree traversals with traversal splicing. In *ACM SIGPLAN Notices*, volume 47, pages 355–374. ACM, 2012.
— 2 citations on pages 57 and 61

[36] Timothy L Kay and James T Kajiya. Ray tracing complex scenes. In *ACM SIGGRAPH computer graphics*, volume 20, pages 269–278. ACM, 1986.
— One citation on page 57

[37] G. Keller, M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. In *Proc. of the 15th ACM SIGPLAN international conference on Functional programming*, volume 45, pages 261–272, September 2010.
— One citation on page 1

[38] Hsiang-Shang Ko and Jeremy Gibbons. Modularising inductive families. In *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming*, pages 13–24. ACM, 2011.
— One citation on page 10

[39] Philip J Koopman Jr, Peter Lee, and Daniel P Siewiorek. Cache behavior of combinator graph reduction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14(2):265–297, 1992.
— One citation on page 3

[40] S. Marlow, P. Maier, H.-W. Loidl, M. K. Aswad, and P. Trinder. Seq no more: better strategies for parallel haskell. In *Proceedings of the 3rdACM Haskell symposium on Haskell*, Haskell '10, pages 91–102, 2010.
— One citation on page 59

[41] Conor McBride. The derivative of a regular type is its type of one-hole contexts. *Unpublished manuscript*, pages 74–88, 2001.
— One citation on page 9

[42] Conor McBride. Clowns to the left of me, jokers to the right (pearl): Dissecting data structures. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 287–295, 2008.
— 3 citations on pages 7, 10, and 61

[43] Conor McBride. How to keep your neighbours in order. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 297–309, 2014.
— 2 citations on pages 34 and 43

[44] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.
— One citation on page 2

[45] Bruno CdS Oliveira and William R Cook. Functional programming with structured graphs. In *ACM SIGPLAN Notices*, volume 47, pages 77–88. ACM, 2012.
— One citation on page 61

[46] Leaf Petersen, Dominic Orchard, and Neal Glew. Automatic simd vectorization for haskell. *ACM SIGPLAN Notices*, 48(9):25–36, 2013.
— One citation on page 58

[47] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 72–83. IEEE Computer Society, 2012.
— One citation on page 2

[48] Yorick Sijsling and Wouter Swierstra. Implementing ornaments through reflection. Technical Report, 2016.
— One citation on page 10

[49] Artjoms Sinkarovs and Sven-Bodo Scholz. Type-driven data layouts for improved vectorisation. *Concurrency and Computation: Practice and Experience*, 28(7):2092–2119, 2016. cpe.3501.
— One citation on page 59

[50] Josef Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *ACM SIGPLAN Notices*, volume 37, pages 124–132. ACM, 2002.
— One citation on page 1

[51] S Doaitse Swierstra, Pablo R Azero Alcocer, and Joao Saraiva. Designing and implementing combinator languages. In *International School on Advanced Functional Programming*, pages 150–206. Springer, 1998.
— One citation on page 9

[52] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + strategy = parallelism. *J. Funct. Program.*, 8(1):23–60, January 1998.
— One citation on page 58

[53] Thomas Williams, Pierre-Évariste Dagand, and Didier Rémy. Ornaments in practice. In *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming*, pages 15–24. ACM, 2014.
— One citation on page 10

[54] Wm.A. Wulf and S.A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, March 1995.
— One citation on page 1

[55] Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, pages 233–244, 2009.
— 3 citations on pages 7, 9, and 26

[56] Ryan Yates and Michael L Scott. Improving stm performance with transactional structs. Technical Report, 2016.
— One citation on page 4