Compiling Tree Transforms to Operate on Packed Representations

Michael Vollmer[1], Sarah Spall[2], Buddhika Chamith[3], Laith Sakka[4], Chaitanya Koparkar[5], Milind Kulkarni[6], Sam Tobin-Hochstadt[7], and Ryan R. Newton[8]

1 **Indiana University, Bloomington, IN, USA**
vollmerm@indiana.edu
2 **Indiana University, Bloomington, IN, USA**
sjspall@indiana.edu
3 **Indiana University, Bloomington, IN, USA**
budkahaw@indiana.edu
4 **Purdue University, West Lafayette, IN, USA**
lsakka@purdue.edu
5 **Indiana University, Bloomington, IN, USA**
ckoparka@indiana.edu
6 **Purdue University, West Lafayette, IN, USA**
milind@purdue.edu
7 **Indiana University, Bloomington, IN, USA**
samth@indiana.edu
8 **Indiana University, Bloomington, IN, USA**
rrnewton@indiana.edu

Abstract

When written idiomatically in most programming languages, programs that traverse and construct trees operate over pointer-based data structures, using one heap object per-leaf and per-node. This representation is efficient for random access and shape-changing modifications, but for traversals, such as compiler passes, that process most or all of a tree in bulk, it can be inefficient. In this work we instead compile tree traversals to operate on **pointer-free pre-order serializations of trees**. On modern architectures such programs often run significantly faster than their pointer-based counterparts, and additionally are directly suited to storage and transmission without requiring marshaling.

We present a prototype compiler, Gibbon, that compiles a small first-order, purely functional language sufficient for tree traversals. The compiler transforms this language into intermediate representation with explicit pointers into input and output buffers for packed data. The key compiler technologies include an effect system for capturing traversal behavior, combined with an algorithm to insert destination cursors. We evaluate our compiler on tree transformations over a real-world dataset of source-code syntax trees. For traversals touching the whole tree, such as maps and folds, packed data allows speedups of over 2× compared to a highly-optimized pointer-based baseline.

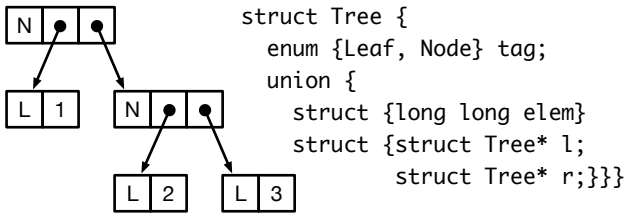
1998 ACM Subject Classification D.3.3 Language Constructs and Features

Keywords and phrases compiler optimization, program transformation, tree traversal

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.97

1 Introduction

Programs that traverse and construct trees are widely used across all domains of computer science, ranging from compiler passes, to the browser Document Object Model, to particle



(a) Standard representation of a tree structure in C: by default, word-sized tags *and* pointers.

(b) Serialized version of the same tree. Not to scale: tags take one byte and integers eight.

Figure 1 Standard and serialized representations of trees

simulations with space-partitioning trees. Yet almost all modern programming languages and compilers represent trees and their traversals identically. Each node of the tree is a heap object, followed by fields for child nodes or leaf values. This representation has not changed since early LISP systems and is shared across source languages with diverse type systems—whether algebraic data types or class hierarchies, statically or dynamically typed. The deviations from this consensus are found within limited high-performance scenarios where complete trees can be laid out using address arithmetic with no intermediate nodes.

We submit that this consensus is premature. In numerical computing it is an axiom that you cannot treat the numbers in a matrix as individual heap objects. Rather, the emphasis is on bulk efficiency. Likewise, many tree traversals process trees in bulk, reading or writing them in one pass. On such workloads, traditional tree representations are not favored by current trends in computer architecture. Pointer-chasing implies randomized memory access patterns. While previous work addresses spatial locality for tree data [4], much memory is still wasted both in pointers themselves and in tags on nodes (e.g. distinguishing "interior" vs "leaf" objects). For example, a C compiler uses 96 bytes of memory to represent the tree shown in Figure 1(a). On the other hand, if we are sending the tree over the network, we would naturally use a more compact form in serializing it, as shown in Figure 1(b). In the latter version, we use the same 24 bytes for the data in the leaves, but only 5 bytes for the spine (capturing the "tags" of the 5 nodes in the tree), rather than 72. Further, a tree traversal processing this memory representation follows a precisely linear memory access pattern, because the data is already laid out in a preorder traversal. On architectures with inexpensive unaligned access, such as modern x86, this is a desirable in-memory representation as well as a serialization format.[1]

Indeed, if we can compile programs to operate directly on this serialization, we follow a precedent of using serialization formats jointly as memory formats. For example, Cap'N Proto [28] makes it ergonomic for C++ code to operate directly on the Protobuf serialization format in memory. Likewise "data baking"[2]. is an established practice in video games—caching assets on disk in a format that allows them to be `mmap`'d into memory and used without further conversion. As a general example of this capability, the Glasgow Haskell Compiler

[1] Even restricted to aligned access, we would still shrink from 72 bytes to 20 by switching to a packed format.

[2] Described here `http://nullprogram.com/blog/2016/11/15/`

(GHC) recently added the capability to store any closed subgraph of the heap as a *Compact Normal Form* (CNF) [29]—a contiguous memory region that is treated as a kind of "super heap object", never traced by the GC and collected only when there are no pointers into any of the sub-parts of the CNF.

The packed tree format above is precisely a dense encoding of a CNF—a transitive closure of heap objects with no escaping pointers, in this case, no pointers *at all*. GHC's CNF support—like related efforts at region [26] or pool memory management [16]—colocates heap objects without changing their representation. Code accessing the data can remain unchanged. In contrast, the dense tree format *requires a complete rearrangement of the compiled code that operates on the data*. This rearrangement is fundamental to the space savings and format simplicity.

In this paper, we take a first step towards compiler support for packed tree data types *without* changing the source program. Packed representations aren't always appropriate, and we don't automate the choice of *when* to use them, but rather automate the necessary code transformations to transparently use packed representations for selected data types. Henceforth, we use *tree traversals* or *tree transforms* to refer to programs that walk over an immutable tree, building an output tree of size proportional to the input tree, without substantially relying on *sharing* in the representation. We also address a limited class of *tree searches* that require random access within a tree. We make the following contributions:

- We present a compiler, dubbed Gibbon, that can compile a range of tree transforms, written in a minimal functional language, to be more than twice as fast as standard techniques (Section 3). We evaluate Gibbon against both a number of existing compilers and its own best performance (without packing) in Section 6.
- We present compilation algorithms for data packing (Section 4), including a method for determining when a function reaches the end of its input(s), and for converting to a destination-cursor-passing style, which supports operating on data in dense byte streams.
- In an additional evaluation, we show that not only can tree traversals become faster in the packed representation, but that they are still amenable to parallel speedup (Section 7.2.1). To leverage parallelism, we need random access and thus extra layout information in dense encodings—a feature that also allows tree searches to be expressed in our framework, such as a point correlation application evaluated in Section 7.2.2.

2 Background and Example

We begin our study of packed tree representations with perhaps the simplest example: binary trees with integer leaves. In a language with algebraic data types, a recursive walk on the tree would typically use pattern matching, which we demonstrate with the following function that increments each integer leaf by one.

```
data Tree = Leaf Int | Node Tree Tree

add1 t = case t of
           Leaf n   → Leaf (n+1)
           Node x y → Node (add1 x) (add1 y)
```

Here we use a Haskell-like syntax, but in fact the small, strict, first-order, purely functional language of tree traversals we consider in this paper is already a subset of most existing languages. The above program is not substantially different in C, Haskell, ML, F#, Scala,

Swift, Rust, etc. Only the details of switching on sum types (tagged unions) differ, as well as the syntax for constructing an object while initializing its fields, here: `Node e1 e2`.

The first problem for tree-walks such as this is memory management, as `add1` can easily become a malloc or garbage collector benchmark. For instance, the following C code is over twice as slow as the same implementation in Java or a good functional compiler, thanks to overhead in `malloc`.

```
Tree* add1(Tree* t) {
  Tree* tout = (Tree*)malloc(sizeof(Tree));
  tout→tag = t→tag;
  if (t→tag == Leaf) {
    tout→elem = t→elem + 1;
  } else {
    tout→l = add1(t→l);
    tout→r = add1(t→r);
  }
  return tout;
}
```

But even if we assume bump-pointer allocation in an arena, and *no header objects*—even if we go further and enable the `__packed__` attribute for our structs to save tag space—the performance of the above code is still several times below what is achievable. The main observation of this paper is that bulk tree walks are efficient if done directly on a pre-order serialization of the tree, and that it is possible to automate the translation of recursive functions, such as `add1` above, into code that directly manipulates data buffers containing serialized trees.

For our simple example, this buffer-passing code isn't complicated to write by hand in C, as pictured on the right. Yet this approach cannot scale—it quickly becomes tedious and error prone. Clearly, no one would use a technique like this for building a non-trivial tree processing program such as a compiler or a web browser!

This C program is similar to the output produced by the Gibbon compiler we describe in this paper. We refer to the input and output pointers as *cursors*, and one of the primary jobs of the compiler is to insert them automatically.

```
char* add1(char* tin, char* tout) {
  if (*tin == Leaf) {
    *tout = Leaf;
    tin++; tout++;
    *(int*)tout = *(int*)tin + 1;
    return (tin + sizeof(int));
  } else {
    *tout = Node;
    tin++; tout++;
    char* t2 = add1(tin,tout);
    tout += (t2 - t);
    return add1(t2,tout);
  }
}
```

2.1 Challenges and Limitations

At a basic level, the remainder of the paper describes how to generate efficient, but complex, cursor-passing C code automatically from the simple functional tree-walking program we began with. However, this code generation process is not as easy as our initial example makes it seem. Our compiler must solve several challenging problems: ensuring complete traversal to consume the stream in order, tracking the state of cursors into the tree, and more. We begin by outlining some of those challenges, and delve into their solutions in subsequent sections. Of course, many challenges can be overcome with extensions to the data

format, and in Sections 2.1.2 and 7 we will explore various extensions to the basic preorder serialization. But we begin with the most basic scenario, where all data for a tree resides in one buffer, contiguously.

2.1.1 Ensuring complete traversal

Our `add1` function is well-behaved and easy to compile. But many real programs, even very simple ones, pose more challenges. For example, consider the following two seemingly-similar functions:

```
left t = case t of
           Leaf n   → n
           Node x _ → left x
```

```
right t = case t of
            Leaf n   → n
            Node _ y → right y
```

These functions are isomorphic to each other in a pointer-based representation. But with a preorder, packed representation there is the stark difference between them. The `left` function only needs access to left branches, which are serialized immediately after the tag for `Node`. But the `right` function needs to *skip over* that left child, to reach the right child. Our prototype adopts a simple solution for this problem: generate a *dummy traversal* that walks the left child to reach the right. This of course is inefficient for many applications, if the tree traversal need only consider a small portion of the tree. But in bulk processing where most of the tree is visited, dummy traversal is simple and fast, preserving the linear memory access pattern favored by modern processors. However, adopting this strategy is not straightforward—the compiler must determine when these extra traversals are needed. This requires the addition of an effect system to track how much of the input buffer is *read*, corresponding to the effect of moving a cursor in the resulting code (Section 4.1).

2.1.2 Extensions

There are many possible extensions to the basic preorder format. For example, we can include *indirections*, which use a distinct tag in the serialized stream to insert a pointer to another buffer or portion of the existing buffer. We can also selectively use alternative constructors that include size information and allow random access. Note that we can *still* save space even while storing size (layout) information. For instance, the `Node` record above could be laid out as: `NodeTag <size_left> <left> <right>`.

Whereas a pointer based representation would spend *two words* for the left and right pointer (16 bytes[3]), if we assume individual tree values are less than 4GB, we need only four bytes for the size of the left tree, and we needn't store the size of the right tree at all! Indeed, we plan to explore the tradeoff between density of encoding, and computational overhead. A dense encoding in the style of UTF8 would enable us to store small values of `<size_left>` in as little as one byte.

We return to the topic of extending the basic format in Section 7, and we present preliminary experiments using layout and indirection extensions in Section 7.2.1 and Section 7.2.2. Further, in the future, it makes sense to fully explore the spectrum of representations between packed and pointer-based. In this paper, to simplify the exposition, we present our core language plus our compilation algorithms in the setting of the simple, *completely* serialized representation.

[3] One basic advantage that we leverage here is that 64 bit platforms have become wasteful of memory, using 8 bytes for every pointer, even though most of the time it is unneeded.

2.2 Related Work

One line of closely related work focuses on managing data layout in trees and other data structures to promote spatial locality [4, 5, 27, 16, 6], by modifying garbage collection to co-locate objects [6], modifying memory allocators to proactively place objects with similar access patterns together [16, 4], or modifying the internal layout of objects to place hot fields near each other [5]. These approaches attempt to "pack" data together, using various techniques, into cache lines to improve spatial locality, and hence have some resemblance to our packed representations, which gain some performance benefits from packing tree data into a compact format that promotes spatial locality.

Perhaps the most closely related of these is Chilimbi et al.'s *cache conscious structure layout* [4]. They propose a cache-conscious data placement scheme where, given a traversal function, tree-structured data will be laid out in memory in a *clustered* manner: nodes from small subtrees will be placed on single cache lines. By matching the tree layout to a specified traversal order, spatial locality is improved when the tree is traversed in that order. A key difference between our packed representation and Chilimbi et al.'s work is that this work focused on object layout, without changing the internal *representation* of the objects. Leaving the object representation of tree nodes the same allows code that manipulates the objects to remain the same, but incurs costs: there is no opportunity to reduce the space or instruction overhead incurred by pointers linking nodes in the tree (see Figure 1), as exploiting that opportunity requires code transformation. Most of the aforementioned spatial locality work makes the same tradeoff.

One exception is Chilimbi et al.'s work on *automatic structure splitting* [5], where objects are transformed into split representations, allowing hot fields from multiple objects to be co-located on a single cache line while those objects' cold fields are placed elsewhere. Because this layout optimization changes the internal representation of the object, Chilimbi et al. develop a compiler pass that automatically transforms code to work with the split representation. The transformations for structure splitting concern how to access object fields, and hence, unlike our work, do not require deeper transformations to remove the pointer dereferences inherent in traversing linked data structures. Indeed, neither this work nor cache-conscious structure placement affect the behavior of pointers in data structures.

Lattner and Adve's *automatic pool allocation* identifies memory allocations that, roughly, correspond to different data structures so that objects from disjoint data structures can be allocated into separate pools [16]. This approach does not change the internal layout of data structures (and hence does not require substantial modifications to the way a data structure is used) nor does it do any further layout optimization to promote locality. However, it does enable a *compression* step. Because pointers *internal* to data structures point to other objects in the same pool, these pointers do not need to point to arbitrary addresses, and can instead use fewer bits to represent the target [17].

Hsu looks at a representation of abstract syntax trees that uses a matrix layout, allowing operations to be specified in a data-parallel manner without traversing pointers [14]. While this representation shares a goal with ours of avoiding pointers, it is not "packed"—the representation requires a dense representation of a sparse matrix—and hence does not yield the type of space savings we target.

In the high-performance computing community, linearizing trees and tree traversals for improved performance has been a common technique [18, 9]. These linearizations tend to be *ad hoc*, written specifically for a given application, and each application must be re-written by hand to benefit. This contrasts with our compiler-based approach which allows programmers to write using idiomatic traversal algorithms, relying on the compiler to synthesize the packed

representation as well as the algorithm to traverse that representation.

Similar *ad hoc* layout transformations have recently been pursued in the context of vectorization [20, 22, 23]. Meyerovich et al. discuss different linearization schemes that can promote packed SIMD loads and stores, improving vectorization efficiency [20]. These layouts have the implicit effect of eliminating pointer dereferences, as in our packed representations, but rely on index arithmetic to traverse formerly-linked nodes, rather than encoding particular traversal orders. Ren et al. look at a wide range of tree layouts for vectorization, each targeted at different traversal patterns [22, 23]. These layouts are chosen to match the traversal patterns of an application, enabling the removal of pointers, as in our layouts. Ren et al. use a library-based approach: applications are written using high-level tree interfaces, with specific layouts chosen based on hardware and application considerations. In contrast, our work focuses on compiler-driven transformations of both the tree layout and the code that traverses the tree.

3 The Gibbon Input Language

To demonstrate the compilation technique we propose, we use a typed programming language simple enough to present briefly in a paper, and featureful enough to express some interesting tree-manipulating functions, such as compiler passes.

The syntax is given in Figure 2—it is simply a standard first-order functional language. Programs consist of a series of data type declarations and function declarations. Similar to most functional programming languages, programmers may define *algebraic data types*, and dispatch on them with a `case` form (called `match` or `switch` in some languages). For example, a data type for Peano numbers would have two cases: `Zero` and `Successor`.

Data types declared with `data` are automatically and implicitly *packed* in this language. In this basic design, the only non-packed data types are tuples $(\mathtt{e}_1, \ldots, \mathtt{e}_n)$, accessed with $\mathtt{e}.n$. Note, however, that tuples are sufficient for functions to take and return arbitrary numbers of packed data types. When we perform cursor translation in our compiler, this will mean passing multiple *output cursors* to a function in order to provide buffers for it to write its results to.

Other language features are standard: tuple access, let binding, conditionals, and primitive operations. Conditionals are included to avoid the need for `Bool` to be packed data (because `case` operates on packed data only). Standard primitive values are included such as integers, booleans, and symbols. Finally, Gibbon provides dictionaries (not shown) to support more sophisticated operations such as bulk transformations—substitution on an abstract syntax tree is one example. A fuller language would support richer data types, more operations, and data structures such as arrays and lists, but the crucial elements for expressing tree-shaped data and transformations on trees are present.

Rather than moving directly from a high-level functional language to cursor-oriented low-level C code, our compiler transforms programs first into an intermediate language which captures the crucial invariants. These additional forms are presented in Figure 3 and described in Section 4.

Using Gibbon

Gibbon is implemented as a language built on Racket [8], using Racket's language implementation and extension facilities. Gibbon's type checking support is implemented by compiling to Typed Racket [25]. A programmer can develop and test a Gibbon program using the

$K \in$ Data Constructors, $T \in$ Type Constructors, $v \in$ Variables

Program	$prog$	$::= \overline{dd}\ ;\ \overline{vd}\ ;\ \overline{fd}\ ;\ e$
Packed Data Declarations	dd	$::= \texttt{data}\ T\ =\ \overline{K\ \tau}$
Value Declarations	vd	$::= v\ :\ \tau\ ;\ v\ =\ e$
Function Declarations	fd	$::= f\ :\ \tau\ \rightarrow\ \tau\ ;\ f\ (\ v\ )\ =\ e$
Expressions	e	$::= v \mid n \mid \texttt{True} \mid \texttt{False} \mid e \odot e \mid f\ e$
		$\mid (e_1, \ldots, e_n) \mid e\ .\ n \mid \texttt{let}\ \ v\ :\ \tau\ =\ e\ \ \texttt{in}\ \ e$
		$\mid \texttt{case}\ e\ \texttt{of}\ \overline{K\ \overline{v}\ \Rightarrow\ e} \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$
Types	τ	$::= T \mid (\ \tau_1, \ldots, \tau_n) \mid \texttt{Int} \mid \texttt{Bool} \mid \ldots$
Prim Ops	$\odot$	$::= + \mid - \mid * \mid \ldots$

Figure 2 Grammar for source language.

Expressions	e	$::= \ldots \mid \texttt{switch}\ v\ \texttt{of}\ \overline{K\ (v) \Rightarrow\ e} \mid \texttt{toEnd}\ (e) \mid \texttt{fromEnd}\ (e)$
		$\mid \texttt{write}\ (\text{`}K\text{'},\ v\ ) \mid \texttt{write}\ (\ n\ ,\ v\ ) \mid \texttt{read}\ (\ v\ ) \mid \texttt{finish}\ (e)$
Types	τ	$::= \ldots \mid T_\ell \mid Needs([\overline{\tau}], \tau) \mid Has([\overline{\tau}]) \mid End(\hat{\ell})$
Extended vars	v	$::= \text{v} \mid \texttt{end}_{\text{v}} \mid \texttt{start}_{\text{v}}$
Location vars	ℓ	$::= \alpha \mid \beta \mid \ldots$

Figure 3 Extensions to the core language for cursor-inserting compilation. Here we read and write word-sized (or smaller) values from byte streams. And switch is a low-level mechanism to read and case on the next tag byte from a stream.

DrRacket IDE and tools, which include code coverage, syntax highlighting, on-the-fly type checking, etc.

Given a working Gibbon program, it can then be compiled using our compiler via a C backend and a standard C compiler. These backends apply the techniques described in subsequent sections to automatically use packed data to represent all types declared using the `data` form.

4 Compilation Algorithms

Gibbon's approach is to convert programs into a form of *destination passing style* [15], where destinations are not managed per-heap-object (i.e. per-data-constructor), but rather for entire trees or subtrees. This approach implies function calls producing data types do not generally call the allocator, for example, even a simple function such as `f` below (on the left) is transformed to take a destination cursor argument, as shown on the right:

```
data Foo = MkFoo Int
f() = MkFoo 3
```

```
f ptr = let p2 = write('MkFoo',ptr)
        in write(3,p2)
```

We say that data types like `Foo` are *packed* types, whereas `Int`, `Bool`, `Symbol`, etc. are not. As we will see in this section, during compilation the data constructors for packed types (`MkFoo`) will themselves come to require destination cursor arguments, before eventually ending up in the final state (shown above) of writing directly to input and output data

streams. We insert these cursors using the extended language of Figure 3, which includes an extended type-system for safely dealing with cursors (currently used only by the compiler, and not exposed to the user).

Functions do not, however, merely have the effect of writing destination memory. Sometimes functions will need to allocate new memory regions as well. We treat tuples `(e1,e2)` as value types, so they don't account for allocation. But consider expressions `(e :: T)`, where `T` does not contain packed values, yet subexpressions of `e` have types which do. For instance:

```
g n = (case MkFoo n of MkFoo i → i) + 4
```

If the optimizer does not eliminate this silly expression, then `MkFoo` *must* be given a destination, even though the constructed data does not escape the function `g`. For this purpose, we will use a very simple form of region allocation which takes advantage of the purely functional nature of the Gibbon language. Namely, we know that the case expression of type `Int` above can have no other visible effect or communication than producing an `Int`, so thus we can *region allocate* the `MkFoo` constructor inside a buffer that is freed when the expression returns (in the implementation, this resembles stack allocation). This follows the precedent of other languages such as UrWeb [7], as well as previous work on region types [26, 11].

This matter of destination routing is the primary function of the Gibbon compiler. However, to support it, other analyses are required. For instance, determining the destination cursor for a field within a data record requires determining an *end witness* for the field before it—that is, a pointer to the position in the buffer that marks the end of one field and the start of the next. If we recursively unpack adjacent fields without storing a pointer to the later fields, we must rediscover those downstream fields as a side effect of *traversing* their upstream ones. (For example, in our binary tree data type, to discover the start location of `y` in `Node x y`, we must first scroll through `x` in the preorder packed data.) Thus we begin with an inter-procedural analysis of which functions are able to traverse their inputs.

The overall structure of the compiler, covered in the rest of this section, is:

1. Infer traversal effects (Section 4.1).
2. Generate additional traversals as necessary to reach input ends (Section 4.2).
3. Route end-of-value witnesses as additional function returns (Section 4.3).
4. Switch to destination cursor-passing with additional function arguments (Section 4.4).
5. Code generation (Section 4.5).

4.1 Inferring traversal effects

To reason about traversals, we associate with every packed type an *abstract location*. This is different from a region variable in prior work, because it is a symbolic value representing the *exact memory location* that a value starts at. No two distinct data constructors can share the same location, whereas two values can share the same "region". The type signature of `add1` becomes:

```
add1 :: Tree_α → Tree_β
```

This is read "function f takes a tree at location α and produces one at location β." Note that a function of type $\texttt{Tree}_\alpha \rightarrow \texttt{Tree}_\alpha$ is *necessarily* the identity function. Next, if `f` examines all the bytes in α, we say it has the effect $traverse(\alpha)$ and we write its type as:

```
add1 :: Tree_α -α-> Tree_β
```

We write end_α to signify the location after the last byte of α, or $\hat{\alpha}$ for short. One way of looking at a function that traverses α is that it can *witness* end_α. At runtime, this witness is merely a pointer value. Ultimately we will rewrite the function to return such a witness. For now, the goal of the effect inference pass is to determine a consistent traversal type for all functions jointly. Of course, if `f` calls `g`, whether `f` reaches (witnesses) the end of its input may depend on whether `g` does likewise.

A lattice of locations

The locations used above, α, β, are *metavariables* that can range over different locations, depending on what the (location-polymorphic) function `f` is applied to. Intuitively, we expect *outputs* to be polymorphic in location, corresponding to the as-yet-undetermined destination parameter. Conversely, inputs already exist in memory at a fixed location. This includes lexically-bound variables introduced by λs or pattern matching. For example, the variables `tr`, `x`, and `y` from `add1` below.

```
add1 :: Tree → Tree
f(tr) = case tr of Node x y → ···
```

In fact we name these fixed locations after their lexical variables, simply: ℓ_{tr}, ℓ_{x}, ℓ_{y}. In contrast, `let`-bound variables take on the locations of their right-hand-side. Every data constructor in the program introduces a *fresh* location. Fixed variables only unify with themselves, but fresh variables unify with any other (non-tuple) location. Together with tuple locations (ℓ, ℓ), these fresh and fixed locations form a lattice under unification. For example, $(\ell_1, \ell_2) \sqsubseteq (\ell_3, \ell_4)$, if and only if there exists a substitution on metavariables that ensures $\ell_1 = \ell_3 \wedge \ell_2 = \ell_4$. Such a substitution assigns fixed locations to metavariables, and does *not* allow metavariables to range over entire tuple locations.

In this lattice, non-packed values such as integers always have location $\bot$. On the other hand a top value ($\top$) is reached when two locations are incompatible. For example, the following term has location $\top$ because it attempts to unify two fixed locations ℓ_{x} and ℓ_{y}.

```
(case p of Node x y → if _ then x else y) :: SomePackedDatatype
```

Indeed, we cannot statically know what *location* this expression will return, even symbolically. (We have no notion of disjunction locations in our definition: e.g., $\ell_{\text{x}} \vee \ell_{\text{y}}$.) Finally, ends are always distinct locations from starts: $\forall \ell. end(\ell) \neq \ell$.

Analysis and fixed point

We use the lattice of locations above to perform a program analysis, assigning a location to each subexpression, as well as a set of traversal effects. The basic idea is that an expression `case e of ...`, creates a traversal effect for the location of `e` provided that all the branches of the case traverse the (non-statically-sized) arguments of their data constructors. This stage of the process is *optimistic*, in that it assumes that any additional traversals that are *necessary* but not *present* will subsequently be inserted later. For example:

```
  case v of K (y :: Tree) (x :: Int) → x
```

Here, when reading data type `K` from a a preorder serialization in a buffer, accessing the simple scalar `x` requires somehow traversing `y` to witness $\hat{\ell_{\text{y}}}$, where $\hat{\ell_{\text{y}}} = \ell_{\text{x}}$. During the infer effects phase, we optimistically assign the traverse effect, $traverse(\ell_v)$, to the above code, *assuming* that a dummy traversal will later be inserted (Section 4.2). If it were not, this program couldn't compile!

Even with this assumption, determining the traversal effect signature for each function is nontrivial because of interdependencies between functions. Thus we design this pass as a traditional program analysis that iterates to a fixed point. We begin with every function having a *maximum* traversal signature—we assume it reaches the end of *every* packed input. Then, this set monotonically decreases in every round until the fixed point is reached.

The running `add1` example does not contain mutual recursion, so it takes only one iteration to reach a fixed point. But the reasoning is still recursive (inductive)—add1 is only able to traverse its input because its recursive call sites traverse their (subtree) inputs:

```
add1 :: Tree_α -α-> Tree_β
add1 t =          — when the polymorphic type is instantiated, α ↦ ℓ_t
 case t of        — case has traverse(ℓ_t), because all branches do
   Leaf n    -> Leaf (n+1)    — fresh location; γ, static size, thus traverse(ℓ_t)
   Node x y -> let x' = add1 x in — x' at fresh loc; call's effect: traverse(ℓ_x)
               let y' = add1 y in — y' at fresh loc; call's effect: traverse(ℓ_y)
               Node x' y' — traverse(ℓ_y) implies traverse(ℓ_t)
```

Here the compiler has also performed a bit of standard flattening, introducing temporaries. Inferring the traverse effect for the `Leaf` case is trivial, because once we know `t` is a `Leaf`, we know its exact byte size, and can compute $\hat{\ell_t} = \ell_t + 9$ bytes. In the `Node` case, because of the polymorphic signature, $(\forall \alpha\beta.\ \texttt{Tree}_\alpha \xrightarrow{\alpha} \texttt{Tree}_\beta)$, the lexical variables `x'` and `y'` have fresh, unrestricted locations, but, more importantly the recursive call gets the effect $traverse(\ell_y)$, due to the effect annotation on the function's type ($\xrightarrow{\alpha}$).

4.2 Copy and traversal insertion

During analysis, we generated all the information we need not only to *label* traversal effects in function signatures, but to recognize where they are needed, but missing, and where destination-location constraints conflict. Next we need to *repair* the program to fix these problems. With the inter-procedural traversal types settled, we reprocess the program and repeat the same location analysis, but this time, we mark wherever we are (1) *missing* a witness of a field stored within a packed buffer, or (2) have *conflicting* constraints where a packed value flows to two incompatible destinations (sharing).

First, a missing end-witness can always be restored, if necessary, by inserting a call to a dummy traversal function. For example, the program fragment from the previous subsection (with a missing traversal) would take the following form after a dummy-traversal insertion:

```
 case v of K (y :: Tree) (x :: Int) ->  — Here we know ℓ_x = ℓ̂_y
   let end_y = traverseTree y in x
```

Here, `traverseTree` is synthesized by the compiler based on the structure of the type definition. The call to `traverseTree` may look like dead code, but it's dead code with the correct location, which lets the compiler pass described in the next section reuse the end of `y` as the start address of `x`.

Second, a *conflicting* destination location can always be resolved by inserting a copy function[4]. A simple example of a program that forces a copy is one that introduces sharing:

[4] More generally, we can perform a *program synthesis* here to fix the program by generating a recursive call that meets that constraint. Copies work, but so does inlining. Ultimately, when we consider indirection extensions to the data format (Section 7), the program repair process interacts with data-structure layout choices, because sharing can be addressed by adding (limited) indirections back in.

```
let x = f t in  Node x x
```

In later extensions (Section 7), we will use these missing traversals and conflicts to go back and *change the data format* (i.e. use packed records augmented with indirection nodes, rather than the most straightforward preorder serialization). But, for *completeness*, it always suffices to naively insert copies or dummy traversals. Copy insertion for the above program would break the sharing:

```
let x = f t in  Node x (copyTree x)
```

Here the call to `x` can flow to the destination location right after the `Node` constructor, and can, from there, be copied to occur a second time in the output buffer. Inlining can also resolve these conflicts, producing `Node (f t) (f t)`, in which the two calls flow to different destations in the output buffer. Our current prototype compiler prefers inlining where possible (because it enables subsequent optimizations), and uses copy-insertion otherwise.

4.3 Routing end-of-value witnesses

After all traversal constraints are satisfied by recursive calls or compiler-inserted traversals, we then transform the program in a type-directed way, to include additional return-values: end-witnesses.

```
add1  :: Tree_α  -α->  Tree_β                  — Before
add1' :: Tree_α  ->  ( End(α̂), Tree_β )        — After
```

Here the type of the end-witness is $End(\hat{\alpha})$, which signifies a cursor (pointer) to the end of a value, which is not useful by itself. Rather, it is useful if it witnesses the *start of another value*. This brings us to the topic of our **type system for cursors**. Cursors are internal to the compiler, rather than exposed to the user. We use a typing discipline resembling *session types* [13] to ensure their correct handling in the compiler's intermediate language—specifically, the types ensure that data is read from and written to buffers in the correct order.

We add three new cursor types: the *End* type, as mentioned above, *Has* cursors for reading, and *Needs* cursors for writing. These will be described further in Section 4.4. In brief, $Has([A, B])$ is an input pointer that, when read from, yields a value of type A as well as a pointer of type $Has([B])$. The *Has* type is parameterized by a list of types A, B, ..., which correspond to the types of values that must be read in a particular order from the buffer. $Needs([A, B], C)$ is an output pointer that requires a value of type A be written to the pointer, followed by B, after which a fully initialized value of type C can be read from the buffer. A given *Needs* cursor must be used linearly, after the address is written to, writing it again would clobber existing data.

During the routing pass, we use these cursor types to insert additional bindings in the program that explicitly encode facts about how to reach the end of a given location. This uses $\texttt{start}_v$ and $\texttt{end}_v$ as special variables to refer to the physical start and end locations of other variables. ($\texttt{start}_v$ is roughly `&v` in C.) Namely:

- One field's end becomes its successor's start. This becomes a binding, such as:
  `let` $\texttt{start}_y$`=` $\texttt{end}_x$.
- Fields of static sizes have known offsets, such as:
  `let` $\texttt{start}_y$`=` $\texttt{start}_x$`+ offset`.
- In `case a of K b1...bn` $\rightarrow$ ..., the end of last field `bn` is also the end of `a`, thus
  `let` $\texttt{end}_a$`=` $\texttt{end}_{bn}$`in` $\cdots$

We could record these facts in program metadata, but in our current approach we instead manifest them explicitly as let bindings. Note, however, that they may refer to (temporarily) unbound end-variables! We solve this later with a pass that reorders these bindings.

Performing this transformation on `add1` yields a program with extra bindings as well as the additional end-address-of-input return values.

```
add1' :: Tree_α → ( End(α̂), Tree_β ) ;
add1' tr = case tr of
    Leaf n →  let end_n = toEnd(start_n + 8) in
              let end_tr = end_n in
              (end_tr, Leaf n+1)
    Node x y →  let start_y = fromEnd(end_x) in
                let (end_x,x') = add1' x in
                let (end_y,y') = add1' y in
                let end_tr = end_y in
                (end_tr, Leaf x' y') ;
```

Just as with the dummy traversal example earlier, the compiler at this phase does not use the $\texttt{start}_y$ binding. Later, when we switch to using explicit cursors into input and output byte streams (Section 4.4), we lose direct access to fields beyond the first one, and the $\texttt{start}_y$ binding then replaces the binding for `y`.

Further, to make the injected bindings above type check, the compiler must insert *coercions* between *Has*/*Needs* types on the one hand and *End* types on the other. The `toEnd`/`fromEnd` forms are coercions. The compiler ensures the correctness of these coercions and offset computations. For instance, given $\texttt{start}_n$ `:: Has(Int)`, we know that $\texttt{start}_n$`+8` is a valid offset (8 is the size of `Int`), but that 7 would not be.

Lastly, before we proceed, note that the original textual order of the program does not effect the results of traversal inference or end-witness discovery. This is because the compiler aggressively reorders programs in order to connect end-witnesses with their consumers. (Starting with purely functional programs makes this easier.) For example, the following two programs for summing the leaves of a tree are equivalent to the compiler.

```
sum1 t = case t of Leaf n → n | Node x y → sum1 y + sum1 x
sum2 t = case t of Leaf n → n | Node x y → sum2 x + sum2 y
```

4.4 Output cursor insertion

Next we are ready for the core translation in the compiler—switching to destination-cursor-passing calling conventions. This proceeds in two phases:

- First, perform a dataflow analysis and mark every data constructor, K, or function call which returns packed data, with a *destination*. A destination is a static source location of another constructor application, or is one of the output terminals of the enclosing function definition, i.e. location β in a $\texttt{Tree}_\beta$ output. Copy-insertion will have guaranteed a unique destination for each such value (i.e. no sharing).
- Second, perform a type-directed, type-preserving cursor-insertion pass. This augments functions with additional inputs (output cursors), and changes their return value convention to return additional end witnesses for outputs as well as inputs. That is, rather than conventionally returning the start address of an output value, the function now returns the end-address of that same value.

For example, the `add1` function becomes:

```
add1'' :: (Has([Tree_α]), Needs([Tree_β],γ)) →( End(α̂), End(β̂)) ;
add1'' cin cout =
  switch cin of
    LEAF(cin1) → let cout2    = write(LEAF,cout) in
                 let (cin2,n) = read(cin1) in
                 let cout3    = write(cout2, n+1) in
                 (toEnd(cin2), toEnd(cout3))
    NODE(cin1) → let cout2         = write(NODE,cout) in
                 let (end1, cout3) = add1'' cin1 cout2 in
                 let (end2, cout4) = add1'' fromEnd(end1) cout3 in
                 (end2, toEnd(cout4))
```

The new `switch` form reads one byte from an input buffer and dispatches based on the contained tag. Each case of the switch statement binds a cursor pointing to the beginning of the first field of the matched data—naturally these cursors have different *Has* types based on the types of fields in the respective data constructor. The return value of the function has turned into a *End* cursor, whereas the inputs have turned into read and write cursors respectively (*Has* and *Needs*). These behave much like typed channels with protocols. We use the extensions in Figure 3 to write and read cursors:

```
write :: (Needs(a : rst, b), a) → Needs(rst, b)
read  :: Has(t : rst) →(Has(rst), t)
```

Here we use Haskell-style list syntax at the type-level, so single-colon is "cons", and the list literal `[a,b]` is shorthand for `a : b : []`. *Needs* tracks a list of values its *waiting for*. For instance, given a data type, `data Foo = MkFoo Int Int`, after we write a tag for `MkFoo` to an output buffer, the output cursor has type `Needs([Int,Int],Foo)`. The second argument of the *Needs* is the type of the value which will be completed only after all the obligations have been satisfied. Once the list of needed-values is empty, retrieving the completed value can be accomplished with `finish`:

```
finish :: Needs([],T) → Has([T])
```

In the context of the above example, if `cout` :: $Needs([Tree_\ell], \gamma)$, then the expression `write(cout,LEAF)` has type $Needs([Int], \gamma)$, whereas `write(cout,NODE)` has type $Needs([Tree, Tree], \gamma)$, corresponding to the different number and type of the fields for those respective data constructors.

Locally dilated representation of packed values

Sometimes the end-witness of a given value is computed, say, underneath a conditional. Thus we may need to change the types of expressions to (locally) tack on additional return values. In order to accomplish this, our cursor-inserting transformation internally switches to a *dilated* representation of every packed value. Inside the local scope of a function body, a subexpression that originally returned $\texttt{Tree}_\alpha$, must instead return a pair $(Has(\texttt{Tree}_\alpha), End(\hat{\alpha}))$. The transformed program routes these tuple values throughout the function body, making it possible for the compiler to directly produce $End(\hat{\alpha})$, in the tail position of the function body, to satisfy the calling convention by returning an end-witness. Note that the inter-procedural calling conventions do not change to reflect this dilated representation, rather, mediation happens at the call sites.

One surprising aspect of the cursor-passing output language is that it is *still purely functional*. Rather than directly encoding effects, we have created a purely functional interface where `write` returns a new cursor, and all *Needs*() cursors must be used in a *linear*, but pure, way. (For instance, in the Gibbon interpreter we use for debugging, evaluating programs after every compiler pass, we model cursors as *lists*, where write is "cons".)

The fate of constructors and case expressions

Here we cover the `switch` form in more detail. The cursor-insertion pass lowers constructors to operate directly on destination cursors. Thus `MkFoo 55` becomes a multi-step operation, where we first initialize the `MkFoo` structure (returning a cursor pointing to its `Int` field), then write the `55` to that cursor. We capitalize data constructor names when they are used as simple one-byte "enum" values:

```
let cur2 = write(MKFOO,curs) — 1 byte tag
    cur3 = write(cur2,55)    — 8 byte int
in (curs, toEnd(cur3))  — Return dilated start/end pair. cur3 = curs+9
```

The above program is well-typed, following the protocol on output cursors. The "value" of the resulting data constructor is now equivalent to the pointer location at which it was written, i.e. `curs`, which we return in the body, together with an end-witness to match the dilated convention. Note again that cursors themselves are persistent, not mutated, which is why each operation with side-effects on a buffer (e.g. `write`), returns a fresh cursor representing the new value of the cursor.

Finally, what becomes of the `case` expression? `case v of K x y → e2`, if both `x` and `y` are of fixed size, is ultimately translated to:

```
switch start_v of
  K(cur) → let (cur2,x) = read(cur)
               (cur3,y) = read(cur2)
           in e2
```

Here `switch` reads one byte from the cursor given as its scrutinee, and then dispatches based on that tag (just like C's switch statement). It is a binding form only insofar as it binds a cursor, `cur`, to the position just after the tag—i.e., the start of `K`'s fields. Then, we generate explicit code to read the fields one at a time from the appropriate positions in the byte stream. Not that the *Has* type for `cur` will contain *two* values, `cur2` one, and `cur3` zero remaining values.

4.5 Code generation

The final step for Gibbon compiler is to generate native code. Any backend target would do (LLVM, native code, etc.), but we presently generate C code. Because the current Gibbon design is a first order language, this is straightforward. We generate C code in static single-assignment form.

The compiler eliminates tuples through "unarization", except at function call returns where structs encoding tuples are returned. Tuple function arguments become multiple arguments. Conditionals that return tuples instead write multiple destination variables. The compiler walks through the program to accumulate all remaining anonymous tuple types, and emits C struct declarations for each.

The cursors become merely `char*` pointers, and `switch` statements closely correspond to C switches, while `read` and `write` are open-coded as pointer operations. Because `read`

returns two values, it generates multiple statements rather than creating a tuple, and thus need not create a struct. Finally, the generated C code is linked with a simple run time system that includes code for (de)allocation and initialization.

5 Implementation

In the next section we evaluate a prototype Gibbon compiler, implemented in Haskell, exposed through a Racket `#lang` mode, and generating code via C. This compiler implements the algorithms of the prior section, with a few current limitations.

Packed & Pointer, malloc & bumpalloc

The Gibbon C backend supports multiple modes of code generation, which we compare in the next section. The first distinction is between two primary implementation strategies:

- **packed**: Generate code using all the compiler passes of Section 4. Data of packed type can be read from disk in human readable or binary/packed formats, but in memory it stays always in preorder serialized representation.
- **pointer**: Use traditional C struct representations. This mode provides a baseline for comparison. It shares the front and back of the compiler with packed mode. But, in pointer mode, we skip the transformations that introduce cursors and packed representations. Rather, we use a traditional pointer graph of heap objects to represent all data. This mode uses the default policies of the C compiler for `struct` layout.

The packed mode manages tree data by allocating large buffers to serve as output destinations (and an additional large region for scoped allocations). In the future, we will employ the standard technique for a block-structured heap, where a linked list of blocks provides growable storage areas for destination cursors. The current performance should be representative of an implementation strategy that uses large regions capped by guard pages to enable unbounded growth.

Within the **pointer** mode, we allocate regular heap objects and thus need an allocation strategy. One policy is to use the system `malloc` implementation, but this does not typically perform well given the large numbers of fine-grained allocations incurred by out-of-place tree transformations. A second strategy is to use a custom arena-allocation method for storing heap objects. This doesn't change the internal layout of heap objects, but it does pack them densely within cache lines and provides a near-optimal memory management strategy—about the best you can do without going to packed. We use a simple arena implementation where a single global variable stores the heap pointer, which is incremented upon allocation.

For each of these implementation strategies, no garbage collection is required. In both packed and pointer mode, we can use coarse-grained arena deallocation. Our region inference is currently quite simple compared to a compiler like MLKit [26], and is not yet suitable for programs complex programs with complex lifetimes. Our present benchmarks allocate regions for input and output trees, and temporary packed data that does not escape a lexical scope is freed at the end of the scope. Finally, for comparison purposes, we also generate code for a fine-grained malloc/free mode of the pointer-based backend, which substitutes a recursive "free tree" function in place of arena deallocation.

For benchmarking, we add an `iterate(e)` form to the language which runs an expression multiple times and reports the time for all iterations together. Iterate *also* resets the state of the arena allocator, after each iteration but the final one, in order to "undo" the effects of previous iterations and avoid leaking memory. Thus, when we benchmark a traversal with

`iterate(treetraversal(tr1))`, we repeatedly walk `tr1` to produce `tr2`, such that `tr2` is allocated into the same memory region on each iteration. This optimizes our use of the cache if both input and output trees fit in memory. It is this optimized version of the "bumpalloc" pointer-based mode that provides the most competitive baseline against which to evaluate our proposed packed-mode compiler pipeline in Section 6.

Embedding Gibbon

Ultimately the ideas in Gibbon should either be ported to a mature compiler for existing general purpose languages, or the prototype Gibbon compiler should be used as an embedded domain-specific language (EDSL) from within such a host. In the latter scenario, we would write tree-traversals in a subset of the host language that corresponds to Gibbon, and those traversals would then be compiled to a shared object file and linked back into the host program for transparent interoperability. Tree data would be marshaled at the boundary, as usual, in this case converting from pointer graphs to packed representations. Indeed, this arrangement is similar to that used by existing EDSLs for, e.g., GPU programming [19, 3, 24], except that those languages are typically focused on arrays and matrices and exclude recursive sum types and recursion—which are Gibbon's emphasis.

Currently, we've taken the first steps to making Gibbon available as an embedded language in the host language Racket. Our front-end Gibbon is available as a custom `#lang gibbon` mode in Racket. This provides IDE support via DrRacket, while enforcing all the specific restrictions of our minimal language (including using Typed Racket to enforce the type system with good error messages and source locations). What remains is to enable in-calls and out-calls between Racket and Gibbon. Indeed, these are already possible using a Gibbon backend which simply expands Gibbon (during macro expansion) to run as native Racket code. Eventually, the C backend will likewise be supported without modifying the program.

In the next section we compare to the Racket backend as a baseline for a high-level language with significant overheads. This information is useful, but the more relevant data for evaluating the packing technique is to compare the different modes supported by the C backend (packed and pointer).

6 Evaluation

We evaluate the performance of our approach in three ways. First, we examine the performance of packed vs. pointer-based tree walks in idealized microbenchmarks. We also use these microbenchmarks to examine the state of the art in several existing compilers. And while we find significant variation between compilers, no existing system we're aware of comes close to matching Gibbon's packed mode. Second, in Section 6.2, we evaluate an important class of tree traversals—AST traversals, as found in a compiler. We use ASTs gathered from real programs to ensure realistic shape and depth. Specifically, these tree benchmarks operate on Racket's intermediate representation, and show substantial speedup using the packed representation. Finally, in Section 7, we look ahead to what a future compiler will be able to achieve *if* it extends the basic preorder packed format, including indirections or layout information to enable parallel traversals.

All benchmarks were conducted on a cluster of identical, dedicated Intel machines in a two socket configuration with Xeon E5-2670 CPUs at 2.60 Ghz, 20MB cache, and 32GB RAM, running Ubuntu 14.04 LTS. All C code is compiled with GCC 5.3 and `-O3`.

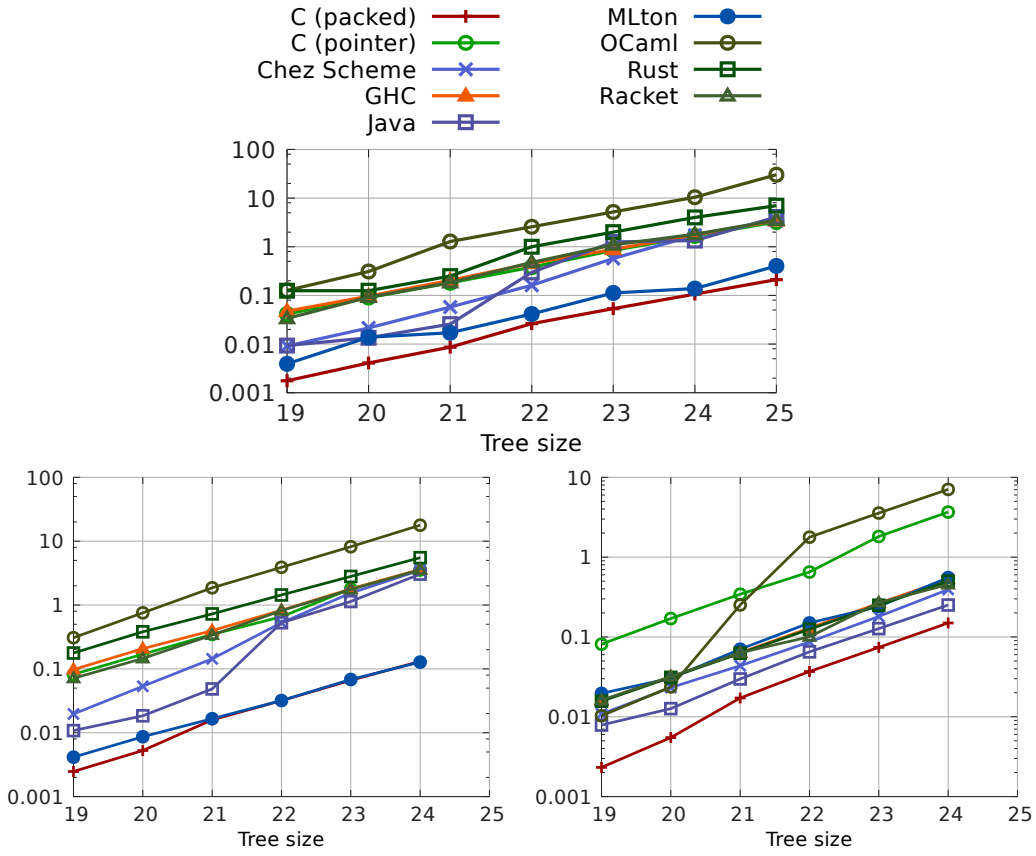


Figure 4 Performance when building (left), mapping add1 (top), and summing (right) a tree respectively. Traditional compiler approaches vs. the packed approach. All handwritten implementations. X axis is tree *depth*, implying 2^N leaves. Y axis shows time in seconds.

6.1 Microbenchmarks

Our first benchmarks return to the example from Section 2: simple binary trees. We implement three operations: constructing a tree, incrementing the values in a tree, and summing the elements of a tree. To understand the performance of packed data representations, we implemented these three operations in multiple ways across a variety of languages: with pointer-based trees in Racket, Chez Scheme, MLton, GHC Haskell, and C (using both `malloc` as well as a fast bump-pointer allocator).

Figure 4 shows the results for these three microbenchmarks on purely *handwritten* implementations, while varying tree size. The results show a clear advantage for packed representations (note the log scale), in some cases with 100x speedup over pointer-based representations in garbage-collected languages. All competing implementations use their default memory management settings, including GC parameters as well as the "C (pointer)" using the system malloc.

We next implement the tree building and summing benchmarks in Gibbon, and use our prototype cursor-inserting compiler to generate code over packed representations. Figure 5 shows the results of comparing this generated code to both Racket and the handwritten C version of the packed representation. We see that our compiler generated code is competitive with, and occasionally exceeds, hand-written C code performing *packed* tree traversals.

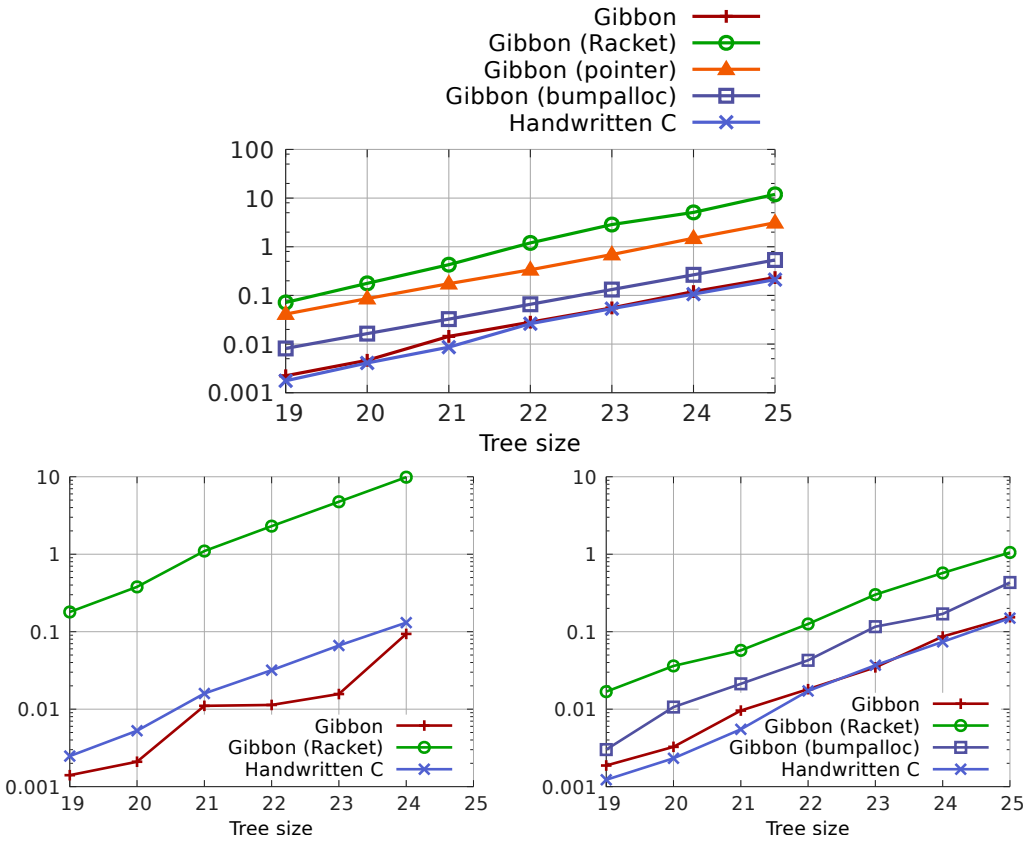


Figure 5 Cursor-inserting compiler's performance compared to handwritten cursorized C implementation. Tree building (left), tree summing (right), and mapping add1 over the tree (top). The Gibbon prototype is currently embedded in Racket, so we show its Racket backend as well, as different modes of its C backend (packed, pointer, bumpalloc).

Figure 5 shows building, mapping over, and summing a tree, separately. Here we introduce a couple of additional variants, which we will carry into the next section. First, the **pointer** version of Gibbon, as explained in Section 5, uses the same code generator, but does not pack trees, and uses system malloc and free to manage memory. This version is faster than the Racket backend, but much slower than packed. Also, over these benchmarking runs, at these tree sizes, the pointer-based implementations consume 6× more memory than packed ones.

However, there is one more mode of the Gibbon code generator, **bumpalloc**, also described in Section 5, which shrinks the gap further. "bumpalloc" uses the same representations as "pointer", but approximates optimal memory management with cheap arena allocation rather than simple malloc. Still, it remains the case that on add1, packed yields a geomean speedup of 1.75× over bumpalloc, and 18× over the malloc-based pointer code.

The influence of leaf size

In our simple tree example, we have thus far used a single `Int` as the payload of the leaf. This implies a certain, fixed ratio of payload bytes to the memory used for storing the *structure* of the tree—i.e., the tags in the packed representation, or tags and pointers in a traditional representation. We would expect that increasingly "heavy" nodes, with many scalar fields,

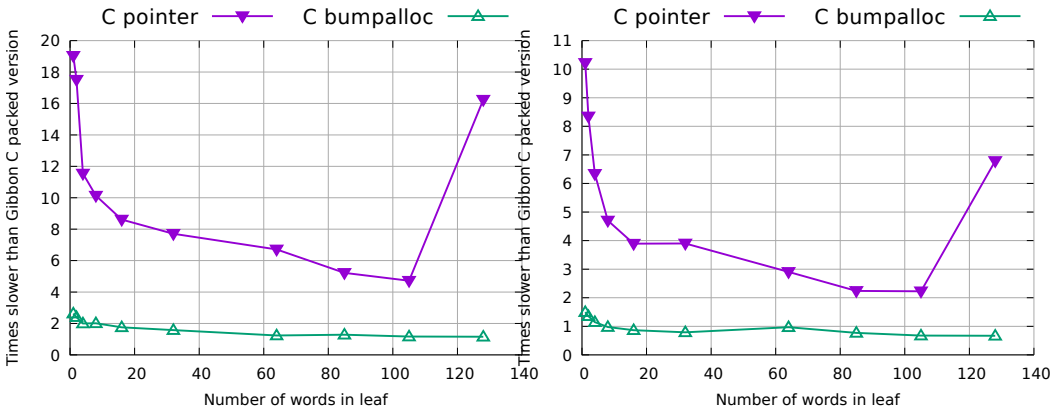


Figure 6 The factor *slowdown* of competing approaches compared to a baseline of Gibbon's packed mode. The malloc-based implementation performs especially badly when given large structs of over 800 bytes each.

would directly reduce the advantage of the packed representation. To verify this hypothesis, we ran a simple parameter study where we generated alternate versions of the `Tree` data type and add1 traversal over it, varying the number of `Leaf` fields. Figure 6 shows the results. As expected, the best performance of the packed approach is with *zero* leaves, and the performance of the bumpalloc version catches up as the scalar payload of leaves increases.

Pathological cases

Because Gibbon fixes a traversal order, it is possible to write programs that exhibit pathalogically bad performance when compiled with the packed approach. A simple example is a function that traverses a binary tree to return its right-most leaf. With the pointer approach, the function need not ever inspect the left child of any node, while with the packed approach the compiler must traverse both left and right children of every node, leading to asymptotically worse run-time complexity. For example, when run on trees of height 12, the generated packed code runs 150× slower than the pointer code (and arbitrarily slower on progressively larger trees). In Section 7, we propose a solution to this problem: the addition of indirection in packed buffers.

6.2 Compiler passes on realistic inputs

While our microbenchmarks demonstrate the potential of the packed representation, and also demonstrate Gibbon's ability to automatically generate code that operates on the packed representation from idiomatic implementations, they don't demonstrate a large savings of programmer effort, because directly implementing functions on simple data in a packed representation is tractable.

More challenging, however, is to operate on trees that have more complex structure, such as the abstract syntax trees (ASTs) that arise in full blown programming languages: (i) the trees themselves do not have homogeneous structure, so the location of a particular tree node in a packed buffer is intimately related to the types of the other nodes in the tree; and (ii) the operations on the tree nodes are not homogeneous, so the structure of computations (including how to extract particular fields from a serialized representation of a tree node) varies based on the type of the tree node. In this setting, writing a tree

traversal that operates directly over a packed representation is complex and error prone. On the other hand, writing such a traversal in an idiomatic style using pattern matching is fairly straightforward. This, then, is an ideal use case for Gibbon's approach.

Benchmarks

In this portion of the evaluation, we look at the performance of two classes of tree walk on full Racket Core syntax, an AST definition which is excerpted in Figure 8. These benchmarks consume a Racket abstract syntax tree as input and produce either (1) a count of nodes, or (2) a new abstract syntax tree. While we only evaluate two simple treewalks, we note that these traversals contain the two major operation types that might be performed on trees: `map`s, where the output tree is structurally similar to the input tree but with a function applied to each node, and `fold`s, which in this context is transforming an entire subtree into some differently-structured result. Seen at this high level, all compiler passes on ASTs are roughly similar, differing mostly in the work done near the leaves of trees. For example, substitution, copy-propagation, and constant folding all traverse a tree and "act locally". In general, many transformations only transform a small fraction of the input and spend most of their time simply walking over syntax.

We write both benchmarks in Gibbon. We then generate versions of each benchmark, as before, one using Gibbon's *pointer-based* backend (which generates passes over pointer-based ASTs in C), and one using Gibbon's *packed* backend. By letting the implementations differ only in the backends used to generate them, we isolate the performance differences to those that arise from the difference in representation. Because Gibbon allows tree traversals to be written using standard data type match operations, this evaluation also serves to confirm our ability to generate packed implementations from idiomatic code.

We generated a dataset of inputs by collecting all of the (macro-expanded) source code from the main Racket distribution, which contains 4,456 files consuming 1GB of code which drops to 485MB when stripped of whitespace and comments, and 102MB once packed in our dense representation. We benchmark on this entire dataset, but report only on a subset, sampling from a spectrum of sizes. The largest single file was 1.4MB. To simulate larger programs (as would be found in whole-program compilation), we combined the largest K files into one, varying K from 1 to 4,456. This is representative of a whole program compiler, which would indeed need to load these modules as one tree.

Results

First, our benchmark methodology is to traverse each input tree N times, doubling N until the run takes at least two seconds. This gives us a uniform way of measuring both traversals over very small trees and very large ones.

Figure 7 shows the performance of Gibbon's packed mode vs gibbon's pointer (malloc) and bumpalloc modes, expressed as slowdowns of the pointer-based approaches over packed. We measured the last level cache reference and cache misses and found dramatic improvements in these for the packed approach (and modest differences in the number of instructions executed). Nevertheless the performance of pointer-based approaches is good at small sizes: (1) trees are small and fit in cache, (2) the single-threaded workload can acquire all of the last level cache, not contending with other threads on the 16-core machine. The end result is that the system is able to mask the bad behavior of these implementations at these sizes. When the input/output tree sizes exceed the cache size, however, we see a phase shift. Once we need to stream trees from memory, the smaller memory footprints and linear access patterns

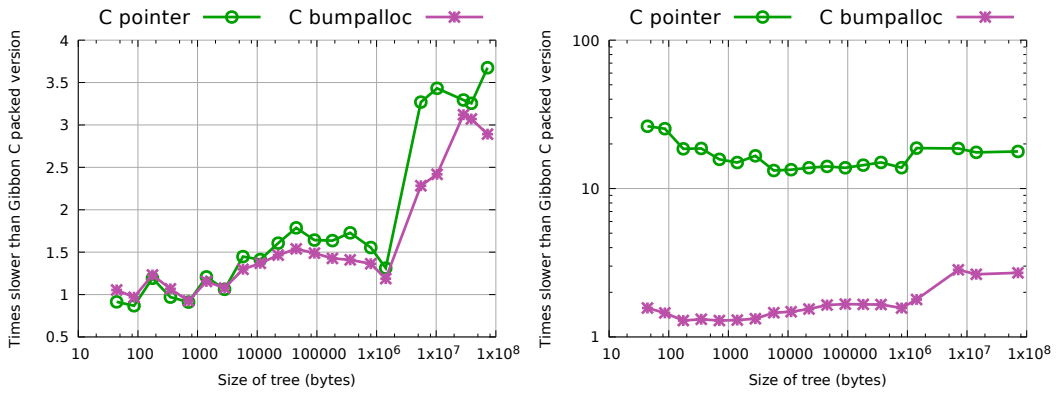


Figure 7 The factor *slowdown* of competing approaches compared to a baseline of Gibbon's packed mode. The X axis is the size in bytes of the (packed) input tree. Left is the fold benchmark which counts the AST nodes in the tree. On the right is a map over the tree.

```
data Toplvl = DefineValues ListSym Expr | DefineSyntaxes ListSym Expr
            | BeginTop ListToplvl       | Expression Expr
data Expr = VARREF Sym | Top Sym  | Lambda Formals ListExpr  | App Expr ListExpr
      | CaseLambda LAMBDACASE     | If Expr Expr Expr        | SetBang Sym Expr
      | Begin ListExp             | Begin0 Expr ListExpr     | Quote Datum
      | QuoteSyntax Datum         | QuoteSyntaxLocal Datum
      | LetValues LVBIND ListExpr | LetrecValues LVBIND ListExpr
      | WithContinuationMark Expr Expr Expr
      | VariableReference Sym | VariableReferenceTop Sym | VariableReferenceNull
...
```

Figure 8 Excerpt of Racket Core AST definition in Gibbon., which follows `https://docs.racket-lang.org/reference/syntax-model.html`. There are nine data types total.

of Gibbon's packed approach yield speedups of 2.5-3× for fold and 2× for map.

7 Extensions

This section evaluates two extensions to Gibbon that enable more complicated traversals and expose more opportunities for performance.

Our benchmarks up until now focus on "full" treewalks: traversals that visit every node of the input tree, in order. While this assumption is accurate for most compiler passes, there are some scenarios and applications where this may not be true:

- If a traversal exploits *truncation*. Some tree traversals, such as those of space-partitioning trees [10] gain asymptotic improvements by *truncating* traversals of subtrees. Based on some condition (for example, that a given subspace in a space-partitioning tree is unimportant), traversal of a node's entire subtree is skipped, and the traversal continues on to the sibling of the current node. This optimization means that not all of the tree is visited by the traversal.
- If a traversal is *parallelized*. To run a traversal in parallel, multiple threads collaborate to walk over a tree. In many traversals, this parallelism is natural: walks over different subtrees are independent of each other. In such a scenario, a single thread may not walk

over the entire tree and, indeed, may not even start its tree walk at the beginning of the buffer holding the tree.

For both of these cases, our current Gibbon compiler is insufficient, because it does not support non-full treewalks. It assumes that the cursor moving through the buffer runs by each node in the tree during the tree walk, and the transformations that ensure that cursors get routed correctly assume the same. The key distinction here is that in both the truncation case and the parallelism case, we need some way to move a cursor to (or generate a cursor at) some later point in a packed tree buffer *without* walking over the intermediate tree nodes.

This section describes an extension to Gibbon's packed representation—*layout information*—that enables these more sophisticated traversals, as well as a evaluation of two benchmarks that use this extended representation.

7.1 Adding Layout Information for Indirection

As described in Section 4.2, our current approach for handling traversals where we need a cursor position (e.g., the position of a right child) without an accompanying traversal that generates it—in other words, if we need to skip over a portion of the tree—is to insert a dummy traversal that traverses the portion of the tree we are skipping. This dummy traversal generates the required cursor position to continue with the "real" traversal. However, this approach can be inefficient if the amount of work done by the dummy traversal is large. In some situations, these dummy traversals can turn $O(\log n)$ operations into $O(n)$ ones, an unacceptable increase in complexity (consider, for example, the `right` example from Section 2.1).

Our solution to this problem is the introduction of *indirections* in the packed representation. These are, effectively, values stored in the packed tree that can be used to generate necessary cursor positions without performing traversals. This layout information amounts, essentially, to adding pointers to our packed representation (albeit ones that only have to be used in lieu of dummy traversals). However, they still preserve some of the space benefits of the packed representation for three reasons. First, indirections are not necessary for the first child, as it is placed immediately after its parent. Second, indirections are only necessary during some portions of traversal; if a particular type of node does not have computations that require skipping subtrees, there is no need to add indirections to that type of node. Third, even if indirections *are* required everywhere, if they are only offsets within the buffer, full (64-bit) pointers are not required, enabling space savings [16].

The particular type of indirection needed depends on the mechanism of the traversal. Here, we discuss two common patterns.

Pointer-style indirection The most common type of indirection is a "pointer style" indirection, where the indirection serves to provide easy access to children beyond the first child: a node contains a field that contains the size of the left subtree. Adding that value to the current cursor allows the cursor to be moved past the left subtree and on to the right subtree. These types of indirections are useful to quickly access, say, the right child of a node without traversing the left child's subtree. The `right` code example from Section 2.1 can benefit from a pointer-style indirection.

Rope-style indirection This is a more subtle style of indirection. In some types of tree traversals (such as those that arise in n-body codes [10]), an interior node is visited and, based on some *data-dependent* condition, either both children of the interior node are visited or *neither* is visited, effectively truncating the traversal of both the left and the right subtrees. This truncation effectively serves as a data-dependent base case

for a recursive traversal. We call these *rope-style* indirections because these types of indirection pointers are frequently called "ropes" when used in GPU implementations of tree traversals [9, 21, 12]. An indirection pointer captures the size of both the left and right subtrees (generalizing, all child subtrees) of a node, allowing the cursor to be bumped to the necessary location upon truncation.
Interestingly, Gibbon's packed representation makes finding the next node easy—a simple calculation of the size of subtrees. In pointer-based representations, finding the next node of the tree requires more work: it could be the right sibling of the current node, it could be the node's parent's right sibling, etc.

Note that in both cases, the indirection pointer's main job is to capture the size of a subtree or subtrees rooted at a particular node. In general, if a given interior node knows the sizes of all of its child subtrees, it can use these indirection pointers to provide *random access* to a tree, even if that tree is in a packed representation. Hence, we call this indirection information *layout information*.

Not every traversal requires full random access to the tree, and hence not every piece of layout information is necessary to synthesize traversals over packed trees. Automatically inferring what layout information is necessary, inserting them into packed representations, and synthesizing cursor updates based on that layout information is a topic for future work.

7.2 Evaluation

Because Gibbon does not currently support a packed representation extended with layout information, our evaluation uses hand-written packed implementations (in C) that include that layout information, mimicking what would be produced by a backend that understands indirections. We evaluate two benchmarks: a *parallelization* microbenchmark (Section 7.2.1) that uses pointer-style indirections to distribute the traversal of a tree, and an implementation of *two-point correlation* (Section 7.2.2) that uses rope-style indirections.

7.2.1 Parallelism opportunity study

We evaluate a *parallel* version of our `add1` benchmark from Section 6.1. In the pointer-based version of this code, adding parallelism using Cilk [2] is straightforward: because the `add1` operation treats the left and right subtrees independently, we can simply add Cilk `spawn` commands to recursive calls to introduce parallelism, cutting off parallelism (after depth 5) to avoid runtime overhead.

For the packed representation, however, we cannot simply adopt this approach: being able to `spawn` a task that processes the right subtree of a node requires being able to reach that right subtree without traversing the left subtree. We thus manually introduce *pointer-style* indirections that allow programs written over the packed representation to directly access the right subtree, facilitating parallel execution. In any scenario where there is a Cilk `spawn`, we use the indirection pointer to launch the right-subtree task, allowing that work to be stolen. Once we cease using `spawn`s, producing coarse-grained leaf tasks, we revert to the full tree-walks supported by Gibbon.

Figure 9 shows the result of our parallel packed implementation (left), compared to the performance of a mature parallel functional compiler, GHC, running the same benchmark (right). While for small trees we see that our parallel implementation does not yield much scaling, for large trees we can achieve a speedup of about 11× on 16 cores (relative to one-core execution). In contrast, the GHC implementation cannot scale beyond eight cores. At these

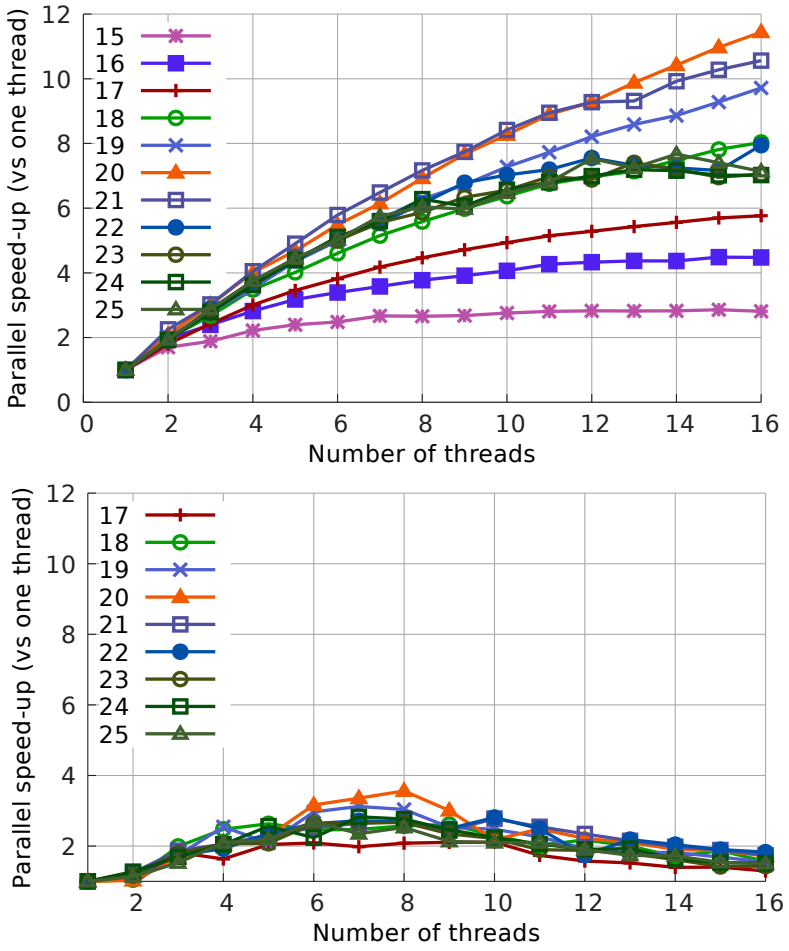


Figure 9 Parallel speedup: mapping a function over a packed tree. Each line is labeled with the tree depth that it represents, including trees of 2^{15} to 2^{25} leaves. This compares a Cilk (C) implementation using the packed trees with layout information that allow random access to subtrees (top). For comparison, we also show the parallel speedup from a mature parallel functional compiler (GHC, bottom). All lines are normalized to their own 1-core speeds. In absolute terms, GHC starts off 34× slower than our approach at one core, and grows to 223× slower at 16 cores.

allocation rates, GHC spends much of its time in garbage collection, and the runtime system presents a bottleneck. When comparing the packed implementation directly to GHC, the packed version is 34× faster on a single core and 223× faster on 16 cores!

While automatically exploiting parallelism in Gibbon is future work, these results demonstrate the potential for large performance gains.

7.2.2 Point correlation

Point correlation is a well-known algorithm used in data mining [10]: given a set of points in a k-dimensional space, point correlation computes the number of points in that space that lie within a distance r from a given point p.

In a naive implementation of point correlation, each point in the space needs to be checked against the query point. A more efficient approach is to use kd-trees [1] to store the points. KD-trees are space-partitioning trees where the root of the kd tree represents the entire

space, and each node's children represents a partition of that node's space into two subspaces. KD-trees allow the search process to skip some regions in the space. By storing at each internal node the boundaries within which all descendent points lies, the search process can skip a subtree is a given point is far enough from the boundaries. As a result, querying a kd-tree to perform point-correlation is $O(log\ n)$ instead of $O(n)$. Note that it is exactly the process of "skipping" subtrees that gives kd-tree-based point correlation its efficiency, but also that prevents a normal packed representation from sufficing to implement the algorithm: there is no way to skip past a subtree without performing a dummy traversal, obviating the asymptotic performance gains.

We implemented both a standard pointer-based version of 2-point correlation in C, as well as a version that operates over a packed representation augmented with indirection pointers. Each interior node stores a rope-style indirection pointer that maintains the size of its child subtrees. If a traversal is truncated at that node, the cursor is incremented by the value in that indirection pointer, skipping the subtrees and resuming traversal on the rest of the tree.

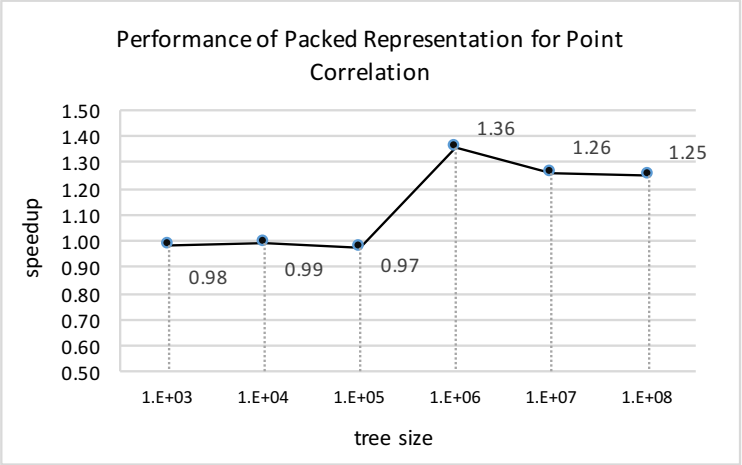


Figure 10 Speedup of packed implementation of point correlation over pointer-based implementation. X axis shows varying tree sizes (represented in number of nodes).

Figure 10 shows the speedup of the packed version with respect to the standard pointer-based implementation for different tree sizes. For each tree size, we ran 10 query points through the tree. For small trees, the queries were performed 10000 times to produce sufficient runtime for accurate measurements. Each experiment was performed 10 times, and the mean is reported.

We note first that for every tree size, the packed representation uses 56% less memory than the pointer-based trees. This reduction in memory usage has two sources: nodes do not need to store left-child pointers; and more efficient packing of data in the packed representation. For small trees, the runtime performance of the packed and pointer versions are comparable. For large trees, the packed version is up to 35% faster than the pointer-based version.

We note that the relatively smaller performance improvement for this benchmark versus the AST benchmarks is unsurprising. First, taking an indirection means that any spatial locality gains from the packed representation are lost, resulting in similar behavior to the pointer-based version. Second, there is relatively more work to be done per node in this benchmark, so the time spent in traversal of the tree is relatively less, reducing the opportunity for improvement.

8 Future Work and Conclusions

Future work

While our initial results show that packed tree-based data representation have significant promise for accelerating tree transformations, much more work remains to be done. First, our Gibbon compiler remains an initial prototype—a more realistic implementation supporting arrays, lists, and more base values would allow the construction of more interesting programs, further validating our hypotheses. We also plan to support optional automatic inclusion of layout information to enable applications such as kd-trees directly in Gibbon. This should support studies in auto-parallelization, which can use packed data regions to coarsen tasks and help with parallel communication and memory management.

The area of buffer management also deserves attention. For example, using indirections, it is possible to write a insert or rebalancing operation on an immutable packed tree, by writing the new nodes into a fresh buffer (like a transaction log). But this quickly introduces fragmentation and memory reclamation problems that must be managed.

Another extension is *data type factoring*, storing leaves in a separate, dense, aligned vector. This enables (1) vectorization of numeric operations, and (2) separating out pointers that the GC must traverse. This may prove essential for an open-world implementation of the Gibbon approach in a managed language such as Java, Haskell, or Racket, where GC support is necessary and interoperation with arbitrary pointer-based values is desirable.

Conclusions

This paper investigates the use of *packed* representations to represent tree structures, which serialize a tree and eliminate the pointers connecting the various nodes. While this representation saves space and, with carefully-written code, can result in performance improvements (due to prefetching and spatial locality), writing programs that operate directly on the packed representation is challenging and error prone. To address this problem, this paper introduces Gibbon, a simple functional language and compiler that allows programmers to write tree traversal algorithms in standard, idiomatic ways (recursion over algebraic data types), and a compiler that automatically generates the packed representation for an application and transforms Gibbon programs to operate directly on that representation.

We show through a series of microbenchmarks and case studies that our packed representation is highly efficient compared to pointer-based representations, both in terms of space usage and time, and that we can process complex data, such as the full Racket language's ASTs in Gibbon, and automatically translate them to packed implementations. We also discuss extensions to Gibbon's representation that introduces selective random access to packed tree nodes, enabling more complex applications.

References

1 J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18:509–517, September 1975.

2 R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Notices*, 30:207–216, August 1995.

3 K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 89–100. IEEE, 2011.

4 T. Chilimbi, M. Hill, and J. Larus. Cache-conscious structure layout. *ACM SIGPLAN Notices*, 1999.

5 T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 13–24, New York, NY, USA, 1999. ACM.

6 T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement, 1999.

7 A. Chlipala. An optimizing compiler for a purely functional web-application language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 10–21, New York, NY, USA, 2015. ACM.

8 M. Flatt and PLT. Reference: Racket. Technical report, PLT Design, Inc., 2010. `http://racket-lang.org/tr1/`.

9 M. Goldfarb, Y. Jo, and M. Kulkarni. General transformations for gpu execution of tree traversals. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (Supercomputing)*, SC '13, 2013.

10 A. G. Gray and A. W. Moore. N-body'problems in statistical learning. In *NIPS*, volume 4, pages 521–527. Citeseer, 2000.

11 D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *PLDI*, 2002.

12 M. Hapala, T. Davidovic, I. Wald, V. Havran, and P. Slusallek. Efficient Stack-less BVH Traversal for Ray Tracing. In *Proceedings 27th Spring Conference of Computer Graphics (SCCG) 2011*, pages 29–34, 2011.

13 K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *Proceedings of the 7th European Symposium on Programming: Programming Languages and Systems*, ESOP '98, pages 122–138, London, UK, UK, 1998. Springer-Verlag.

14 A. W. Hsu. The Key to a Data Parallel Compiler. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2016, pages 32–40, New York, NY, USA, 2016. ACM.

15 J. Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. PhD thesis, University of California at Berkeley, 1989.

16 C. Lattner and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. *ACM SIGPLAN Notices*, 40:129–142, 2005.

17 C. Lattner and V. S. Adve. Transparent pointer compression for linked data structures. In *Proceedings of the 2005 Workshop on Memory System Performance*, MSP '05, pages 24–35, New York, NY, USA, 2005. ACM.

18 J. Makino. Vectorization of a treecode. *J. Comput. Phys.*, 87:148–160, March 1990.

19 T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier. Optimising purely functional GPU programs. In *ICFP: International Conference on Functional Programming*, pages 49–60. ACM, 2013.

20 L. A. Meyerovich, T. Mytkowicz, and W. Schulte. Data parallel programming for irregular tree computations. In *HotPAR*. USENIX, May 2011.

21 S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26(3):415–424, Sept. 2007. (Proceedings of Eurographics).

22 B. Ren, G. Agrawal, J. R. Larus, T. Mytkowicz, T. Poutanen, and W. Schulte. SIMD parallelization of applications that traverse irregular data structures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*, pages 20:1–20:10. IEEE Computer Society, 2013.

23 B. Ren, T. Mytkowicz, and G. Agrawal. A portable optimization engine for accelerating irregular data-traversal applications on SIMD architectures. *TACO*, 11(2):16:1–16:31, 2014.

24 B. J. Svensson, M. Sheeran, and R. R. Newton. Design exploration through code-generating dsls. *Commun. ACM*, 57(6):56–63, June 2014.

25 S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. pages 395–406, 2008.

26 M. Tofte and J. Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, 1997.

27 D. N. Truong, F. Bodin, and A. Seznec. Improving cache behavior of dynamically allocated data structures. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, pages 322–, Washington, DC, USA, 1998. IEEE Computer Society.

28 K. Varda. Cap'n Proto, 2015.

29 E. Z. Yang, G. Campagna, O. S. Ağacan, A. El-Hassany, A. Kulkarni, and R. R. Newton. Efficient communication and collection with compact normal forms. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 362–374, New York, NY, USA, 2015. ACM.