SCS Honors Undergraduate Research Thesis

Algebraic Data Type Representation in Virgil

Bradley Teo Carnegie Mellon University
advised by Ben Titzer Carnegie Mellon University

Abstract. Algebraic Data Types (ADTs) are an increasingly common feature in modern programming languages. We explore various optimizations to ADT representations in Virgil, a systems-level programming language that compiles to x86, x86-64, Wasm and the Java Virtual Machine. In Virgil, programmers can now annotate ADTs as unboxed to eliminate the overhead of heap allocation, and we have extended the language to enable programmer-expressed bit-layouts for varying levels of control on memory layout. The performance impact of these representation changes was evaluated on a variety of workloads in terms of execution time and memory usage.

1 *Introduction*

1.1 Algebraic Data Types

Algebraic Data Types (ADTs), also known as *variants* or *sum types*, are composite types that allow values of that type to take on one of several component types. Typically, ADTs are used together with pattern matching, a succinct method of consuming values of that type that ensures matches are exhaustive and type safe.

Once a feature found mainly in functional languages such as Standard ML, OCaml and Haskell, ADTs and pattern matching are an increasingly common feature in modern multi-paradigm languages, such as Rust, Scala and Swift.

1.2 The Virgil Language

Virgil[1] is a fully self-hosted systems-level programming language that compiles to multiple targets: WebAssembly (Wasm), x86, x86-64 (Linux and Darwin), and the Java Virtual Machine (JVM).

[1] Virgil is fully open source; the source code, including changes implemented for ADT unboxing can be found at `https://github.com/titzer/virgil/`.

Virgil comes equipped with an interpreter that runs on an internal SSA IR. Values in components are computed at compile time using this interpreter, and their values are encoded into a heap image so that they do not incur a cost at run-time. Virgil also features automatic memory management using a precise tracing, Cheney-style copy collector for garbage collection (GC). The runtime distinguishes between references and non-references during stack-walking by consulting a stackmap, and uses bitmaps when tracing objects.

Virgil provides higher-level features such as classes, first-class functions and closures [15]. The two largest programs written in Virgil at present are the Virgil compiler itself, and the Wizard Engine, a Wasm engine built for research.

1.3 ADTs in Virgil

Virgil provides support for ADTs and pattern matching. As in most functional languages, ADTs in Virgil are immutable and equality over them is structural, not referential. One difference is that each ADT definition is not only associated with *one* type; each of the child types can be used as their respective product types, and may have their own associated methods.

```
type Option<T> {
    case None;
    case Some(val: T);

    def val() -> T { return Some<T>.!(this).val; }
    def isNone() -> bool { return None.?(this); }
    def isSome() -> bool { return tag == 1; }
}
```

Figure 1: An generic option type written in Virgil, showcasing features of Virgil ADTs.

An illustration of the syntax of ADTs in Virgil can be found in Figure 1. They nay be generic over type parameters. They can be casted (e.g. `Some<T>.!()`, which may throw a `TypeCheckException`) and queried (e.g., `None.?()`). For convenience, they also have an integer *tag* (and a string *name*) which can be accessed using `.tag` and `.name`. ADTs in Virgil also have a default value[2]: it is the first case with all fields set to their respective default values.

[2] A variable takes on its type's default value when it is not explicitly initialized.

1.4 Problem Statement

ADT values in Virgil are represented in different ways, depending on their number of cases and fields. At present, the representational options are:

- A single unsigned integer, if all cases have no fields. This degenerates ADTs into enums when possible.
- Individual scalars[3], for *explicitly unboxed*[4] single-case ADTs. Such ADTs are represented as multiple separate values without a tag.
- Allocated classes. This is the default for boxed ADTs, which are desugared into classes at the AST level, and use the same record infrastructure to represent ordinary classes.

[3] We use *scalar* to refer to values that will not be split further at lower levels of the compiler, with the exception of numeric lowering on 32-bit targets. A rough approximation of a scalar would be a register value on x86 or a variable in the JVM.

[4] There is a compiler flag to unbox all non-recursive single-case ADTs that normalize to fewer than a given number of fields. Otherwise, unboxing is done for single-case ADTs with the `#unboxed` annotation.

The desugaring of variants into classes reduces the amount of variant-specific logic during compilation (Figure 2). However, this limits the efficiency of ADT representations, as it means that all non-empty, multiple-case ADTs must incur a heap allocation on initialization and an indirection on field access. This can hurt performance, not only due to the overhead of allocation, but also due to the increase in garbage collection pressure. Hence, our focus is to be able to

represent ADTs as scalars efficiently, using a generalizable but target-sensitive algorithm.

2 *Prior Work*

Work has been done on unboxing ADTs through automated monomorphization and specialization in Haskell [7]. MLTon implements ADT unboxing for Standard ML and can also specialize as it is a whole-program compiler [16]. However, it does not provide a mechanism for programmer annotations.

Rust, which compiles to LLVM IR, represents ADTs as tagged values. For unboxed ADTs, they become an LLVM structure type with a tag and an array of associated bytes (the size of the largest `enum` case). This is feasible because Rust is a non-garbage-collected language and does not need to differentiate between reference and non-reference types at runtime. Virgil's JVM target means that it is also not possible to efficiently represent ADTs as raw bytes of data at this phase of the compiler.[5]

There is an open RFC on unboxing ADTs in OCaml [6] with annotations. There are certain limitations that limit the power of this optimization. OCaml requires a uniform representation of values because it must compile generic code without concrete types. Virgil is a whole-program compiler and can perform a monomorphization pass, so concrete types of all values are known during normalization.

Rust performs bit-level tag packing using *niche optimization*. Previously, Rust would only perform niche optimizations on ADTs with multiple nullary cases and a single non-null variant containing a 'niche' [4]. Recent improvements in Rust have allowed for non-nullary cases that start or end with data, as long as they do not interfere with the niche [14]. Swift supports a similar optimization, also enabling multiple non-nullary cases [3].

Zig supports packed structs and packed unions, which enable the expression of bit-level layouts but without flexibility on tag location [1]. A proposed feature, though with no current plans for implementation, would enable user-specified tagging for packed unions [8].

Ribbit is a domain-specific language for memory layouts that targets LLVM IR [5]. It is flexible enough to express the default compiler representations for OCaml and Rust ADTs. Dargent is a description and refinement language for memory layouts in Cogent [12]. It allows for programmer-specified memory layouts of boxed values living on the heap.

Virgil supports byte-level layouts that act as views over byte arrays [2]. C# supports a similar feature for control over a struct's memory layout, using the `FieldOffset` annotation [10]. A recent update to Odin adds support for bit-level layouts, which it calls *bit fields*, backed by a user-specified type [11]. Odin's bit fields also supports specifying the endianness of fields.

```
type T {
    case A(x: u32);
    case B(y: float);
}
```

↓ Variant Desugaring

```
class T.A {
    def x: u32;
    new(x) {}
}

class T.B {
    def y: float;
    new(y) {}
}
```

Figure 2: An illustration of desugaring variants into classes.

[5] Representing ADTs as raw, packed bytes without consideration for scalars would mean fields may be inadvertently split across variables, making their access costly.

3 *The Virgil Compiler*

The Virgil compiler undergoes several phases to lower programs into machine code (Figure 3).

1. *Parsing and Verification*: Source to AST;
2. *SSA Generation*: AST to Polymorphic SSA;
3. *Reachability and Normalization*: Polymorphic SSA to Normalized SSA;
4. *Machine Lowering*: Normalized SSA to Machine SSA;
5. *Code Generation*: Machine SSA to Machine Code.

The phases differ somewhat between different targets; for instance, the JVM target uses its own phase to generate JVM bytecode directly from normalized SSA. Numeric lowering occurs at the machine lowering phase: integer or float values wider than 32 bits, and their arithmetic operations, are decomposed into two 32-bit values, done only for 32-bit targets.

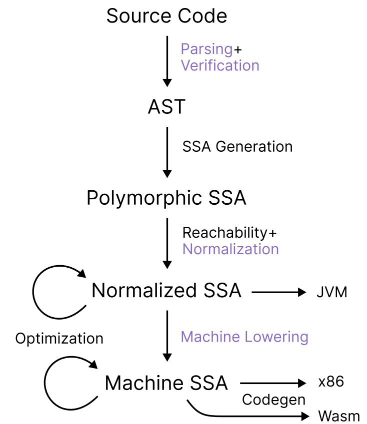


Figure 3: A diagram of the Virgil compiler's phases. Biggest changes from this thesis are in purple.

3.1 Reachability and Normalization

During reachability, the live variables, fields and instructions are determined by a recursive search over the program. Here, we mark if fields always take on constant values, or are never used; this information is used during normalization to simplify the code.

The Static Single-Assignment (SSA) IR of Virgil contains type information. Normalization of this SSA is directed by this type information. At this phase, complex types such as tuples and closures are translated away.

3.2 What needs to change?

The introduction of unboxed ADTs involves a wide range of changes throughout the compiler's phases.

- **Parsing.** The parser has to be modified to accommodate the bit-level layout syntax and the new annotations.
- **Verification.** The verifier must now verify that the packing declarations are semantically possible. This is described in Section 4.2.
- **Normalization.** This is where the biggest changes have to take place; code has to be rewritten to account for the new representations that we are supporting. Variant solving takes place here, because monomorphization has taken place and all types are concretely known.
- **Machine Lowering.** This phase has to be aware of the new packed representation of types in order to generate precise stackmaps.

- **Runtime.** With the addition of tagged pointers, the garbage collector must be changed in tandem with the compiler. The target specifies the bit patterns for reference and non-reference tagged pointers, and the GC must be modified to handle tagged pointers, distinguishing between reference and non-reference values, and masking out the correct bits to extract the underlying reference.

4 *A Language for Bit-Level Layouts*

In the process of enabling programmers control over how they want their ADTs to be laid out in memory, we have designed a simple language embedded within Virgil to specify bit-level layouts.

4.1 Syntax

In the specification of a bit pattern, the first character of a field name is used to identify that field.[6] Packing expressions can include the *concatenation* of other packing expressions using `#concat` and the application of packing declarations.

[6] If this is ambiguous, such as when two fields share the first character, an error is reported during verification.

```
packing Float16(sign: 1, exp: 5, frac: 10): 16 = 0b_seeeeeff_ffffffff;
packing Float32(sign: 1, exp: 8, frac: 23): 32 = 0b_seeeeeee_efffffff_ffffffff_ffffffff;

packing TwoFloat16s(s1: 1, e1: 5, f1: 10, s2: 1, e2: 5, f2: 10): 32
    = #concat(Float16(s1, e1, f1), Float16(s2, e2, f2));
```

Figure 4: A representation of IEEE 754 floating-point numbers using our packing declaration syntax, followed by an example of packing application and concatenation.

```
⟨packing-expr⟩      ::= ϵ | ⟨bit-layout⟩ | ⟨const⟩ | ⟨ident⟩ ( ⟨packing-expr-list⟩ )
                        #concat ( ⟨packing-expr-list⟩ ) | #solve( ⟨packing-expr-list⟩ )
⟨packing-expr-list⟩ ::= ϵ | ⟨packing-expr⟩ | ⟨packing-expr⟩ , ⟨packing-expr-list⟩
⟨packing-decl⟩      ::= packing ⟨ident⟩ ( ⟨param-list⟩ ) : ⟨int⟩ = ⟨packing-expr⟩ ;
⟨param-list⟩        ::= ϵ | ⟨ident⟩ : ⟨int⟩ | ⟨ident⟩ : ⟨int⟩ , ⟨param-list⟩
⟨packing-annot⟩     ::= #packing ( ⟨packing-expr-list⟩ ) | #packing ⟨packing-expr⟩
⟨bit-layout⟩        ::= 0b ⟨packing-bits⟩
⟨packing-bits⟩      ::= ϵ | ⟨packing-bit⟩ ⟨packing-bits⟩
⟨packing-bit⟩       ::= 0 | 1 | ⟨char⟩ | ?
```

Figure 5: Syntax for packing expressions and declarations, in Backus-Naur form.

Lastly, packing expressions can appear in ADT declarations as annotations. In this context, `#solve` expressions can also appear that tell the compiler to

figure out a way to pack the contained fields. `#solve` expressions cannot appear in packing declarations, which must be fully specified. A full description of the syntax of packing expressions can be found in Figure 5.

4.2 Static Verification

During semantic analysis, the compiler verifies that packing declarations are well-formed by checking the sizes of packing expressions. Define the judgment $\Delta, \Gamma \vdash e : n$ to be that packing expression e has size at most $n \in \mathbb{N}$ in the contexts Γ and Δ. Δ is the context containing the packing declarations defined in the program. Expressions that are too short will be padded with zeros. We restrict n to be the size of the largest possible scalar (64 bits on 64-bit targets).

$$\frac{\Delta, \Gamma \vdash e : n \qquad n \leq n'}{\Delta, \Gamma \vdash e : n'} \qquad \frac{|b| = n}{\Delta, \Gamma \vdash b : n} \qquad \frac{\Gamma(f) = n}{\Delta, \Gamma \vdash f : n} \qquad \frac{\overline{\Delta, \Gamma \vdash e_i : n_i}}{\Delta, \Gamma \vdash \mathsf{concat}(\overline{e_i}) : \sum n_i}$$

$$\frac{\Delta, \{\overline{x_i \mapsto n_i}\} \vdash e : n}{\mathsf{packing}\ p(\overline{x_i : n_i}) : n = e\ \mathsf{ok}} \qquad \frac{\mathsf{packing}\ p(\overline{x_i : n_i}) : n = e \in \Delta \qquad \overline{\Delta, \Gamma \vdash e_i : n_i}}{\Delta, \Gamma \vdash p(\overline{e_i}) : n}$$

Figure 6: A subset of the static rules for packing expressions.

Packing annotations on variants are not verified until normalization, since the concrete types of fields are not yet known at this stage before monomorphization.

5 *Solving the Unboxing Problem*

5.1 Conditions for Unboxing

We provide the `#unboxed` annotation for the programmer to annotate a variant as unboxed. Single-case variants that normalize to a small number of scalars are also automatically unboxed.

However, there are two situations in which variants are never unboxed:

- If a variant is recursive (or mutually recursive), we do not unbox it.[7]
- If a variant is closed over (for example, a method `T.f` is referred to without being called), we force the compiler to box `T`, as Virgil represents closures as fat pointers (a code pointer and environment pointer pair).

[7] This is more restrictive than necessary; it should be possible to unbox recursive variants if all the recursive mentions of that type are boxed.

5.2 Scalar and Interval Assignment

We can think of the unboxing problem as two different problems: an assignment of normalized fields to scalars (*scalar assignment*), and an assignment of fields to *bits* in those scalars (*interval assignment*). However, these two problems are deeply intertwined – it is not possible to determine if a scalar assignment is valid

without trying to find an interval assignment on those scalar assignments – and so they must be solved in tandem.

One benefit of performing an explicit scalar and interval assignment, as opposed to just performing interval assignment on one long bit layout, is that fields are now layed out in a way that is sensitive to scalar boundaries. Thus, it will never be possible for a single field to straddle multiple scalars (i.e., multiple registers after code generation), and that field will not need to be reassembled from multiple scalars.

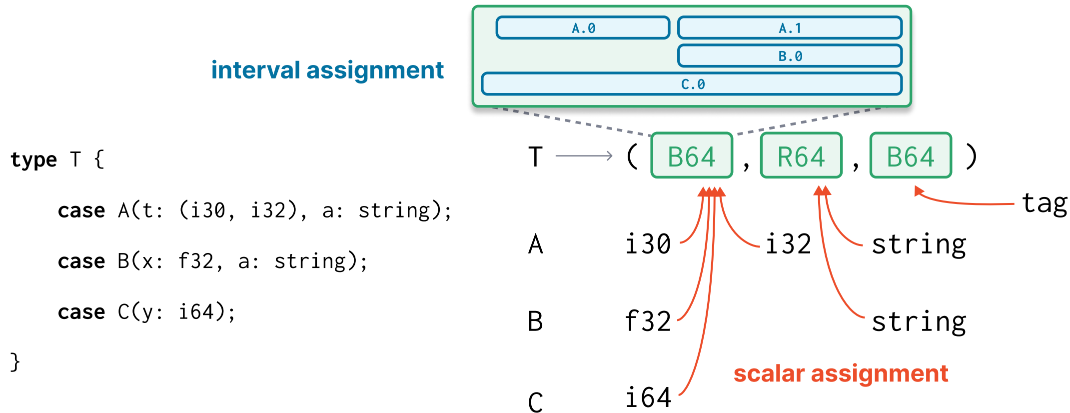


Figure 7: An illustration of scalar and interval assignment.

Our approach to solving the unboxing problem is to use recursive backtracking, sped up with various heuristics, to minimize the number of scalars and cost of field access.

5.3 Scalar Classes

We introduce the idea of a scalar class. At present, the valid scalar classes are `B32`, `B64`, `R32`, `R64`, `Ref`, `F32` and `F64`. These are the union of scalar classes over all supported targets.[8]

During normalization, the target specifies a mapping `GetScalar` between normalized types and *sets* of scalar classes. This set represents the types of scalars that a value of that type can inhabit. For instance, on the x86-64 target, a value of type `u2` could inhabit a `B64`, `F64`, or `R64` register.[9] For the JVM target, a `string` can only occupy a `Ref` scalar, since references in Java are opaque.

These scalar classes help the target express the unifications that are possible between different fields. They also allow us to express which scalar classes are preferential: for instance, mapping `int`'s to {`B32`} and `float`'s to {`B32`, `F32`} will have their merged value live in a integer register, since the intersection is {`B32`}.

The distinction between the reference and non-reference scalar classes also guide the compiler in later phases, when it builds the stackmap for garbage collection.

[8] In the future, we may include `B128` as a scalar class to support XMM registers on x86-64 with SIMD.

[9] If we enable packed references and tagged pointers, integers can appear in bits 2 and 3 of reference values on 64-bit targets due to 8-byte alignment, or after the second bit if the value is not a reference.

5.4 Recursive Backtracking

In order to create an assignment of fields to scalars, we perform recursive backtracking on the assignment of each scalar. As we iterate over the fields present in the variant, we build up a representation. When all fields from all cases have been assigned, we check if the assignments have a valid packing that is also distinguishable. In order to limit the time taken to find a good solution, the number of solving steps is bounded.

5.5 Distinguishability: Explicit and Implicit Tagging

It is insufficient merely to pack the fields for each case; for multi-case ADTs, we need enough information in the packed representation to distinguish between cases. After interval assignment, we have the unassigned bits in the scalar that we can use for this purpose. We have two options here.

- **Explicit Tagging.** If there are sufficiently many aligned, unassigned and contiguous bits after interval assignment, we can encode the case's tag as a field in that interval. If this isn't possible, we can append a new scalar that acts as the case's tag.
- **Implicit Tagging.** There may be sufficiently many unassigned bits for us to use to distinguish between cases, but they are not all correctly aligned. If this is true, we can distinguish between the various cases by constructing a decision tree; each node in the decision tree represents sets of cases that have yet to be distinguished; each node splits on a specific bit position. [10]

[10] Implicit tagging is not yet implemented, only verifying the distinguishability of scalars.

5.6 Heuristics

The aforementioned recursive backtracking algorithm is extremely inefficient on its own; we need the use of heuristics in order to make the packing problem tractable. Additionally, we need some way to *score* solutions, so that the solver is able to pick between multiple valid solutions. Our score is a function of several parameters:

- **Number of scalars.** We penalize solutions that use more parameters, as we would like to encourage more aggressive packing.
- **Access cost.** When possible, we would also like fields to be unpacked, as access cost is reduced. This factor helps distribute fields across scalars, especially for variants with one large case.[11]
- **Presence of explicit tag.** This is essentially the access cost of a variant's tag.

[11] One additional contributor to access cost is casting cost on the JVM: if references are represented as `java.lang.Object`s, then their casting incurs a bytecode instruction. We don't yet consider this as a cost in the current implementation.

5.7 Flattening Packing Annotations

Given a `#packing` annotation, we must convert it into a series of scalars and interval assignments that is recognized by the solver. Each *flattened* packing

expression is a pair containing a bit pattern[12] and a list of interval assignments. This flattening takes place according to the rules in Figure 8. The context Γ stores a mapping of parameters or fields to bit widths; we assume an ambient context Δ containing all packing declarations in the program.

[12] Each packing bit is either 0, 1, assigned (•) or unassigned (?).

- The rule for flattening bit layouts is omitted, but is as expected: for example, flattening the pattern `0b_00aa_bb11` becomes $(\{a \mapsto 4, b \mapsto 2\}, 00\bullet\bullet\bullet\bullet 11)$.
- The rule for application states that we flatten the packing declaration's expression with placeholder fields representing parameters. We flatten all the arguments, and insert the arguments' patterns into the appropriate spots, and shift the assignments by their positions in the full expression.

FIELD
$$\frac{}{\Delta, \Gamma \vdash \mathsf{flatten}(f) = (\{f \mapsto 0\}, \underbrace{\bullet \cdots \bullet}_{\Gamma(f)})}$$

LITERAL
$$\frac{}{\Delta, \Gamma \vdash \mathsf{flatten}(c) = (\varnothing, \mathsf{bits}(c))}$$

CONCAT
$$\frac{\overline{\Delta, \Gamma \vdash \mathsf{flatten}(e_i) = (A_i, B_i)}}{\Delta, \Gamma \vdash \mathsf{flatten}(\mathsf{concat}(\overline{e_i})) = (\bigcup_i A_i, \overline{B_i})}$$

APPLICATION
$$\frac{\mathsf{packing}\ p(\overline{x_i : n_i}) : n \ = \ e \in \Delta \qquad \overline{\Delta, \Gamma \vdash \mathsf{flatten}(e_i) = (A_i, B_i)} \qquad \Delta, \{\overline{x_i \mapsto n_i}\} \vdash \mathsf{flatten}(e) = (A, B)}{\Delta, \Gamma \vdash \mathsf{flatten}(p(\overline{e_i})) = (\bigcup_i \{f \mapsto s + A(x_i) \mid f \mapsto s \in A_i\}, B[\overline{A_i \cdots A_i + n_i \mapsto B_i}])}$$

Figure 8: Rules for flattening packing expressions into patterns and assignments.

6 *Code Normalization*

After solving for a good unboxing and packing solution, the SSA IR must be rewritten in order to make use of this new representation (Figure 9). This takes place during the *normalization* phase.

In the front-end of the Virgil compiler, ADTs are desugared to classes, so they share some of the same SSA operations with classes. There are several SSA operations that can operate or create these ADT values:

- `ClassAlloc`, which creates a value of the specified ADT case from its constituent fields;
- `VariantGetField`, which extracts a specific field from an ADT value;
- `VariantGetTag`, which extracts a numeric tag from an ADT value;
- `VariantReplaceNull`, which replaces a possibly null reference with the default value of that variant. This is necessary as the default value of a class is `null`, but ADTs are non-nullable.

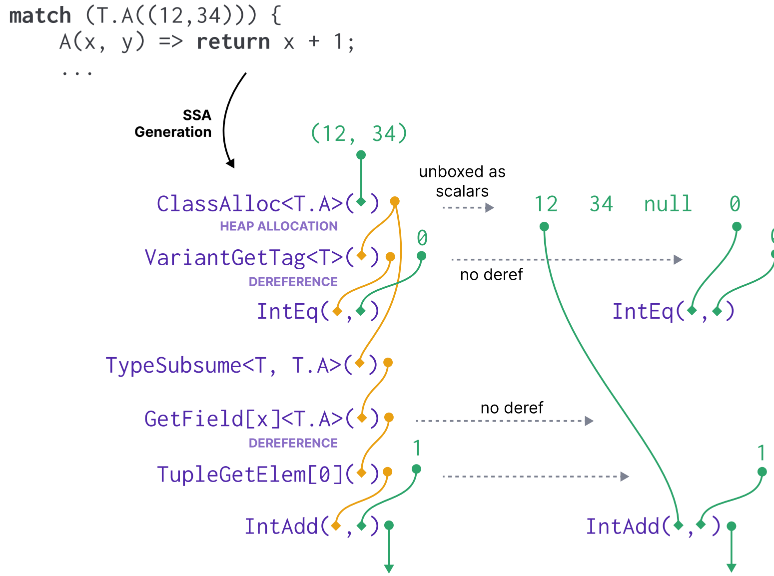


Figure 9: An illustration of SSA rebuilding.

6.1 ADT Operations

We present a simplified but illustrative example of the normalization process for ADTs. Suppose our pre-normalization typed SSA supports the following types:

$$\tau \; ::= \; \mathsf{int} \;\mid\; (\overline{\tau_i}) \;\mid\; \mathsf{class}(\tau)$$

while the post-normalization typed SSA expects the types

$$\rho \; ::= \; \mathsf{int} \;\mid\; \mathsf{class}(\rho) \;\mid\; (\overline{\rho_i}).$$

Consider an ADT of the form $\tau = \mathsf{adt}(\overline{c_i \hookrightarrow \tau_i})$, in which the desugaring process has synthesized the associated types $\tau_{c_i} = \mathsf{class}(\tau_i)$ and $\tau_c = \mathsf{class}(\cdot)$. Operations on the variant τ have been generated as typed SSA operations on class types.

As described in the previous section, we create a *variant norm* for τ that contains information about how the fields and tag are represented (ρ_{field} and ρ_{tag}). We normalize types as in Figure 10.

$$\frac{}{\mathsf{int} \rightsquigarrow \mathsf{int}} \qquad \frac{\overline{\tau_i \rightsquigarrow \rho_i}}{(\overline{\tau_i}) \rightsquigarrow (\overline{\rho_i})} \qquad \frac{\mathsf{class}(\tau)\ \text{boxed} \qquad \tau \rightsquigarrow \rho}{\mathsf{class}(\tau) \rightsquigarrow \mathsf{class}(\rho)} \qquad \frac{\mathsf{class}(\tau)\ \text{unboxed variant}}{\mathsf{class}(\tau) \rightsquigarrow (\rho_{\text{field}}, \rho_{\text{tag}})}$$

Figure 10: Normalization of types.

Given a pre-normalization typed SSA with instructions

$$
\begin{aligned}
p \;\; ::= \;\; \cdots \mid\;& x = \texttt{Const}(v) \\
\mid\;& x = \texttt{ClassAlloc}\langle\tau\rangle(y) \\
\mid\;& x = \texttt{GetContents}\langle\tau\rangle(y) \\
\mid\;& x = \texttt{GetTag}\langle\tau\rangle(y)
\end{aligned}
$$

and a post-normalization typed SSA with instructions

$$
\begin{aligned}
q \;\; ::= \;\; \cdots \mid\;& x = \texttt{Const}(w) \\
\mid\;& x = \texttt{ClassAlloc}\langle\rho\rangle(y) \\
\mid\;& x = \texttt{GetContents}\langle\rho\rangle(y) \\
\mid\;& x = (\overline{y_i}) \\
\mid\;& x = y[i]
\end{aligned}
$$

we can express instruction normalization rules as in Figure 11.

$$
\frac{y \rightsquigarrow y'}{x = \texttt{ClassAlloc}\langle\tau_{c_i}\rangle(y) \rightsquigarrow x = (z, \texttt{Const}(i))}
\qquad
\frac{y \rightsquigarrow y'}{x = \texttt{ClassAlloc}\langle\tau\rangle(y) \rightsquigarrow x = \texttt{ClassAlloc}\langle\tau\rangle(y')}
$$

$$
\frac{y \rightsquigarrow (y'_0, y'_1)}{x = \texttt{GetContents}\langle\tau\rangle(y) \rightsquigarrow x = y'_0}
\qquad
\frac{y \rightsquigarrow (y'_0, y'_1)}{x = \texttt{GetTag}\langle\tau\rangle(y) \rightsquigarrow x = y'_1}
$$

Figure 11: Normalization of SSA Instructions.

6.2 Variant Equality

Variants in Virgil exhibit structural equality. For unboxed variants, equality is normalized as a switch over the variant's tag. For each case, we perform a field-by-field comparison for equality.

6.3 Packed Values

For variant allocation, we assemble packed scalars by performing bitwise left shifts and ors based on the calculated intervals of each field. Field access is normalized as bitwise right shifts and a masking operation.

We have to be more careful when assembling scalars containing packed references. In order to represent packed references, we introduce a new value type `IntRepType` in the IR for bits backed by an integer, representing a specific scalar. For instance, on a 64-bit target, a packed reference would be an `IntRepType` backed by `u64`, representing an `R64` scalar. Since the scalar is known, the machine lowering phase can tell if a value of `IntRepType` represents a reference, in order to generate accurate bitmaps or stackmaps.

6.4 Live Records

Due to the compile-time execution of code in components that are encoded into the heap image, the interpreter generates live records at compile time before normalization. Their representation still rely on their pre-normalization type (with fields that could include complex types).

Instead of having a separate method that normalizes and packs scalars for live records, we use the SSA interpreter to normalize these records, by running them through `ClassAlloc`. Conveniently, this reduces code duplication and guarantees that ADT value representation is identical at compile-time (on the heap image) and run-time.

Although this approach is convenient, it is not the fastest as it involves instantiating an interpreter to execute value normalization. The most performant option would be to create the requisite values directly from the variant normalization at the host level.

7 *Experiments*

There are several dimensions on which we can measure the impact of these changes.

- **Execution Time.** There are several contrasting factors at play here that influence program execution time. Packed scalars can increase field access cost, as bitwise operations have to be performed to extract values. Unboxing also increases register pressure, which may result in poorer register allocation. Simultaneously, forgoing the need for heap allocation could save the cost of dereference and ADT value creation.

- **Memory Usage.** With the elimination of heap allocation for unboxed variants, we would expect less GC pressure and lower memory usage. We can also measure the space savings from packing multiple fields into scalars on real-world type declarations.

7.1 Microbenchmarks

We have written a series of microbenchmarks to test the performance impact of these changes. These microbenchmarks have different characteristics:

- **Allocations and Field Access.** Execution time for ADT-heavy programs is sensitive to the number of allocations if their ADT representations are boxed. Each allocation incurs a runtime cost and could trigger additional GC cycles. Unboxed ADTs do not require any allocations. Field access is also slower for boxed representations due to the cost of an indirection.

- **Movement.** Several of the benchmarks move ADT values around (e.g. by shuffling their position in an array or passing them to functions). Boxed

ADTs should perform better here, as they only involve the movement of one reference value rather than the movement of multiple scalars.

The execution time of these microbenchmarks was assessed on the `x86-64-darwin` target by averaging over 10 runs.

7.2 Big Programs: Wizard Engine and Virgil Compiler

We compile a version of the Wizard Wasm engine with unboxed and boxed versions of `Value`. The two versions of Wizard are run against several benchmarks (Polybench [13] and Ostrich [9]) in Wizard's interpreter mode, and their execution time is recorded. Since the `Value` datatype is used heavily by the interpreter, we would expect some performance impact here. As only Linux is supported for Wizard's native targets, these changes were assessed on the `x86-64-linux` target.

Lastly, we measure the time taken for the Virgil compiler to compile itself (bootstrapping) with unboxed and boxed versions of `Operand`, which is used during machine code generation.

8 *Results*

8.1 Microbenchmarks

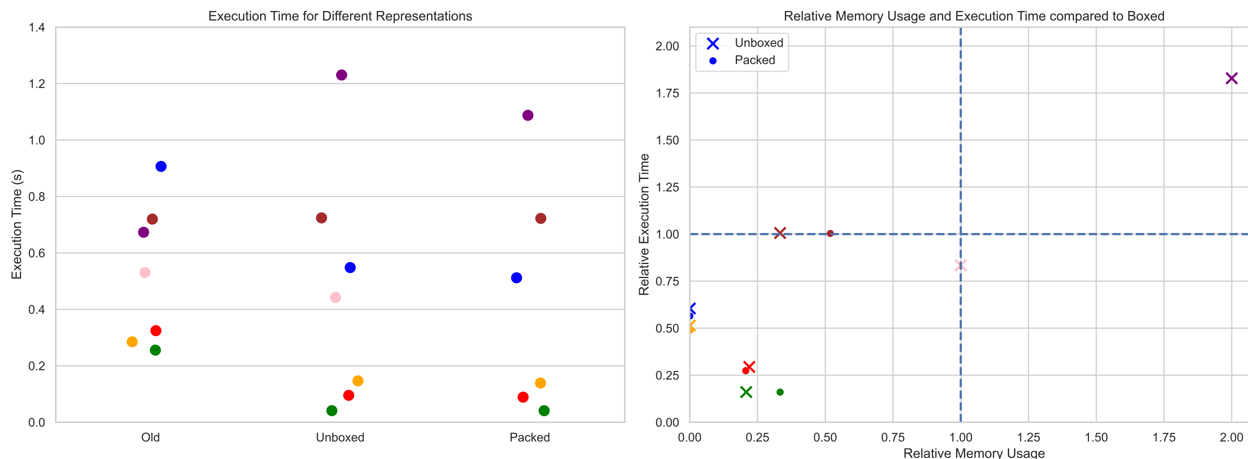


Figure 12: A comparison of execution times for boxed, unboxed and packed ADTs on several microbenchmarks.

In Figure 12, we see that for most of the microbenchmarks, execution time is reduced, sometimes substantially. Memory usage is also reduced for most benchmarks.

For one of the microbenchmarks (Figure 13), execution time is significantly degraded. This microbenchmark tests the passing of an unboxed ADT with many scalars as a function parameter. This confirms our hypothesis that aggres-

sive unboxing can degrade performance when ADT values are moved around a lot during execution.

Several benchmarks have increased memory usage for packed representations over unpacked representations. This is due to inefficiency in the backing types of packed scalars (a packed scalar containing a `u2` becomes a `u64` by default). This problem can be solved by always picking the shortest integer representation possible.

```
type T #unboxed {
  case A(x: int, y: int, z: int,...
  case B(x: int, y: int, z: int,...
  case C(x: int, y: int, z: int,...
  case D(x: int, y: int, z: int,...
}

def shuffle(m: T, n: T, o: T, p: T,
      q: T) -> Array<T> {
    return [m, n, o, p, q];
}
```

Figure 13: A benchmark with degraded performance on unboxed representations.

8.2 Big Programs: Wizard Engine and Virgil Compiler

For most Wasm benchmarks, unboxing yielded modest execution performance benefits of 0.5-5% (Figure 14).

One notable exception was the `heat-3d` benchmark, which saw a performance degradation of 5%. We suspect that this is due to its heavy use of floats, which are encoded as `B64`s in the representation of `Value`. Thus, access to the fields of `Value.F32` or `Value.F64` require `movq` instructions to floating point registers and back, negating the benefits of unboxing.

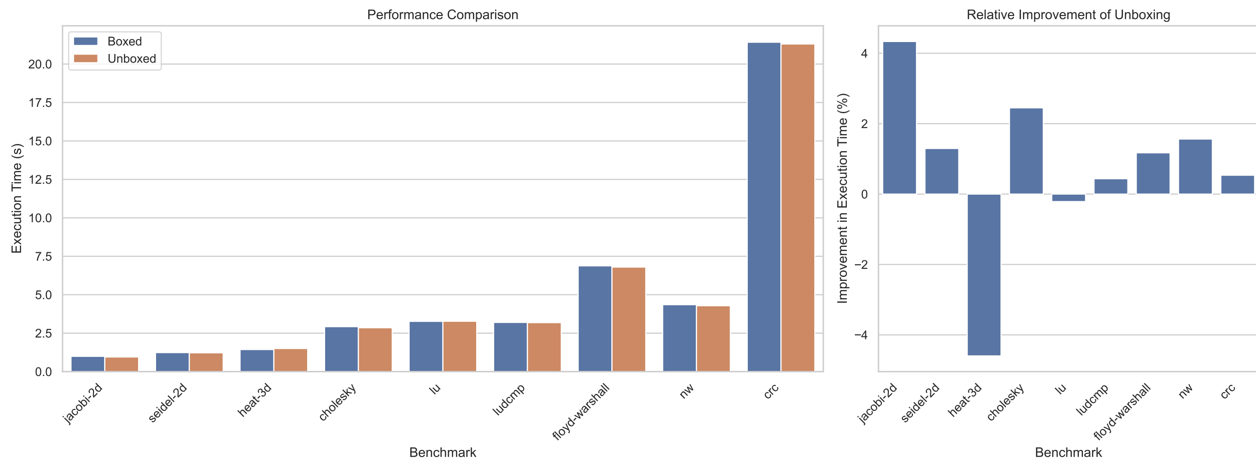


Figure 14: A comparison of execution times in the boxed and unboxed versions of Wizard.

The time for bootstrapping the compiler was improved with unboxing by about 1%. We do not expect the performance improvement to be too large here, since Virgil compiles itself without triggering a GC cycle, and code generation is only one part of multiple phases.

9 *Conclusion and Future Work*

The work that we have done in improving Virgil's representation of ADTs provides an unparalleled level of control and ergonomics, with a combination of features not found in any existing language. Nonetheless, there are a number of improvements to the variant packing algorithm and other future directions we would like to explore.

9.1 Better Bit-Level Layouts

Although the creation of the language of packing expressions and declarations were intended for programmer specification of ADT layouts, this system is more general. A few potential language features include:

- *Pattern matching on bit layouts.* Figure 15 illustrates an example of extending the `match` syntax to support matching over bit layouts. Here, type annotations must be provided in the parameters of the case arm.
- *Bit extraction.* It is frequently necessary to manipulate raw bits when working in certain domains, e.g. when writing an assembler. The necessary machinery to support this is already present, used to flatten packing expressions into patterns and intervals.

```
match(z) {
    MyPacking(x: u2) => return x;
    OtherPacking(x: u2, y: u2) => return x + y;
}
```

Figure 15: An example of a possible syntax for bit pattern matching.

9.2 Non-Contiguous Intervals and Mixed Endianness

At present, the current implementation only allows for fields to be represented by contiguous bits in the scalar. Certain encodings make use of non-contiguous intervals. Non-contiguous intervals will also enable a larger set of packing problems to be solved.

Additionally, we do not support splitting fields across multiple scalars. Access cost for these fields would be larger, but the space savings could be worthwhile under certain circumstances. Thus, we'd like for this to be something programmer-annotated to inform the compiler that the splitting is necessary.

Our current syntax for packing expressions can be readily extended to express these new concepts (Figure 16).

```
packing P(x: u2) = #reverse(x);
type T {
    case A(x: u64, y: u64) #packing (#concat(x[0:32], y[0:32]), #concat(x[32:64], y[32:64]));
```

Figure 16: An example of a possible syntax for field reversal and field splitting.

9.3 SAT-Based Solving and Profile-Guided Unpacking

We plan on creating a separate SAT encoding of the packing problem. An external utility would take Virgil ADT declarations, convert them into the equivalent

SAT problem, find a solution (or declares that no solution is possible), and re-encode the solution into a Virgil packing annotation. This has the benefit of no compile-time overhead in solving, and enables us to leverage the performance strides that have been achieved in the field of SAT solving.

With a SAT-based encoding, we can also assign a cost function to the various configurations in order to find a minimum-cost configuration. Additionally, a profile of the program running on a representative input can be taken, counting the number of variant allocations and field accesses. This profile can be fed into the cost function to derive a representation that best suits the characteristics of the program.

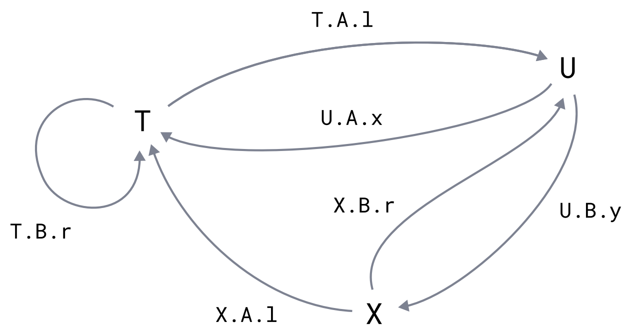
9.4 Cycle-Breaking Recursive ADTs

We do not unbox any ADTs that are recursive (or mutually recursive). Ideally, it should be possible for the compiler to selectively box an ADT, provided that all recursive instances are boxed.

More generally, ADT declarations induce a directed graph where each node represents a type, and each edge represents a field. This problem reduces to removing edges from this graph to break all cycles.

Figure 17: An example of a possible syntax for field reversal and field splitting.

```
type T {             type U {             type X {
    case A(l: U);        case A(x: T);        case A(l: T);
    case B(r: T);        case B(y: X);        case B(r: U);
}                    }                    }
```



References

[1] Documentation - the Zig programming language.

[2] Layouts. `https://github.com/titzer/virgil/blob/master/doc/tutorial/Layouts.md`, 2023.

[3] (Apple). Type layout.

[4] Noah Lev Bartell-Mangel. Filling a niche: Using spare bits to optimize data representations. 2022.

[5] Thaïs Baudon, Gabriel Radanne, and Laure Gonnord. Bit-stealing made legal: Compilation for custom memory representations of algebraic data types. *Proc. ACM Program. Lang.*, 7(ICFP), aug 2023.

[6] Richard Eisenberg. Unboxed types. `https://github.com/goldfirere/ocaml-rfcs/blob/unboxed-types/rfcs/unboxed-types.md`, 2022.

[7] Cordelia Hall, Simon L. Peyton Jones, and Patrick M. Sansom. Unboxing using specialisation. In Kevin Hammond, David N. Turner, and Patrick M. Sansom, editors, *Functional Programming, Glasgow 1994*, pages 96–110, London, 1995. Springer London.

[8] Hejsil. More control over the data layout of tagged unions. `https://github.com/ziglang/zig/issues/1922`.

[9] David Herrera, Hanfeng Chen, Erick Lavoie, and Laurie Hendren. Numerical computing on the web: Benchmarking for the future. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2018, page 88–100, New York, NY, USA, 2018. Association for Computing Machinery.

[10] (Microsoft). StructLayoutAttribute. `https://learn.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.structlayoutattribute?view=net-8.0`.

[11] (Odin). Bit fields. `https://odin-lang.org/docs/overview/#bit-fields`.

[12] Liam O'Connor, Zilin Chen, Partha Susarla, Christine Rizkallah, Gerwin Klein, and Gabriele Keller. Bringing effortless refinement of data layouts to Cogent. In *Leveraging Applications of Formal Methods, Verification and Validation. Modeling: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part I*, page 134–149, Berlin, Heidelberg, 2018. Springer-Verlag.

[13] Louis-Noël Pouchet. PolyBench, May 2016.

[14] the8472. Improve niche placement by trying two strategies and picking the better result. `https://github.com/rust-lang/rust/pull/108106`.

[15] Ben L. Titzer. Harmonizing classes, functions, tuples, and type parameters in Virgil III. *SIGPLAN Not.*, 48(6):85–94, Jun 2013.

[16] Stephen Weeks. Whole-program compilation in MLton. In *Proceedings of the 2006 Workshop on ML*, ML '06, page 1, New York, NY, USA, 2006. Association for Computing Machinery.