Bit-Stealing Made Legal

Compilation for Custom Memory Representations of Algebraic Data Types

THAÏS BAUDON, Univ Lyon, France, EnsL, France, UCBL, France, CNRS, France, Inria, France, and LIP, France
GABRIEL RADANNE, Inria, France, EnsL, France, UCBL, France, CNRS, France, and LIP, France
LAURE GONNORD, UGA, France, Grenoble INP, France, LCIS, France, and LIP, France

Initially present only in functional languages such as OCaml and Haskell, Algebraic Data Types (ADTs) have now become pervasive in mainstream languages, providing nice data abstractions and an elegant way to express functions through *pattern matching*. Unfortunately, ADTs remain seldom used in low-level programming. One reason is that their increased convenience comes at the cost of abstracting away the exact memory layout of values. Even Rust, which tries to optimize data layout, severely limits control over memory representation.

In this article, we present a new approach to specify the data layout of rich data types based on a dual view: a source type, providing a high-level description available in the rest of the code, along with a memory type, providing full control over the memory layout. This dual view allows for better reasoning about memory layout, both for correctness, with dedicated validity criteria linking the two views, and for optimizations that manipulate the memory view. We then provide algorithms to compile constructors and destructors, including pattern matching, to their low-level memory representation. We prove our compilation algorithms correct, implement them in a tool called RIBBIT that compiles to LLVM IR, and show some early experimental results.

CCS Concepts: • **Software and its engineering** → *Compilers*; *Language features*; *Data types and structures.*

Additional Key Words and Phrases: Algebraic Data Types, Pattern Matching, Compilation, Data Layouts

ACM Reference Format:
Thaïs Baudon, Gabriel Radanne, and Laure Gonnord. 2023. Bit-Stealing Made Legal: Compilation for Custom Memory Representations of Algebraic Data Types. *Proc. ACM Program. Lang.* 7, ICFP, Article 216 (August 2023), 34 pages. https://doi.org/10.1145/3607858

1 INTRODUCTION

Algebraic Data Types (ADTs) are an essential tool to model data. They allow grouping together information in a consistent way through the use of records, also called product types, and organizing options through the use of variants, also called sum types. Proper ADT support enables:

- Modeling data in a way that is close to the programmer's intuition, abstracting away the details of the memory representation of said data.
- Safely handling data by ensuring via pattern-matching that its manipulation is well-typed, exhaustive and non-redundant.
- Optimizing data manipulation thanks to rich constructs understood by the compiler.

Authors' addresses: Thaïs Baudon, Univ Lyon, Lyon, France and EnsL, Lyon, France and UCBL, Lyon, France and CNRS, France and Inria, France and LIP, Lyon, France, thais.baudon@ens-lyon.fr; Gabriel Radanne, Inria, France and EnsL, Lyon, France and UCBL, Lyon, France and CNRS, France and LIP, Lyon, France, gabriel.radanne@inria.fr; Laure Gonnord, UGA, Grenoble, France and Grenoble INP, Grenoble, France and LCIS, Valence, France and LIP, Lyon, France, laure.gonnord@grenoble-inp.fr.

2475-1421/2023/8-ART216
https://doi.org/10.1145/3607858

Despite these promises, ADTs were initially only present in functional programming languages such as NPL [Burstall 1977], HOPE [Burstall et al. 1980], and later OCaml and Haskell. Recently, they have gained a foothold in more mainstream languages such as TypeScript, Scala, Rust and soon Java. They are, however, still lacking in high-performance lower-level languages. Indeed, low-level programmers find it essential to have fine-grained control over data layout: how structs are packed, where to insert indirections, or whether to shatter an array of records into several arrays [AoS and SoA 2023]. All these choices and optimizations are essential for performance.

This is particularly unfortunate as the descriptive nature of ADTs enables safe yet efficient representations. Classically, the `Option` type, whose values are either `Some(value)` or `None`, can be simply represented by boxing the value in the `Some` constructor below a pointer. However, if that value is an integer ranging from 0 to 10, we can represent it unboxed and use 11 for `None`. This optimization is regularly done manually by C or C++ programmers, at the cost of error-prone manipulations. Complex optimizations on nested and rich types are even more error-prone.

But how does one actually pick a representation for ADTs? The Rust Reference [2023] famously says of the representation of values: "The only data layout guarantees made by [the Rust] representation are those required for soundness". OCaml, Java and Scala make similar non-guarantees about the art and craft of representing structured types. Even the C standard gives very little information about concrete choices made on a given platform. Most languages similarly keep a prudish veil on memory representation to allow for optimization and language evolution, thus denying programmers the ability to fine-tune data layout. Alternatively, GHC Haskell provides a rich algebra of types with various unboxing annotations. This, however, exposes internal representation choices to the full program, polluting every value manipulation such as construction and pattern matching. All these alternatives come up with specific pattern matching compilation algorithms [Augustsson 1985; Maranget 2008; Sestoft 1996; Wadler 1987] that, despite their efficiency, do not yield themselves to the tortuous data representations found in low-level programs.

In this article, we explore a different approach: a "source" specification relying on run-of-the-mill ADTs and used by the rest of the code, accompanied by a "memory" specification describing the nitty-gritty of data layout in full detail. This dual description enables reasoning about data layout optimizations and provides generic, yet efficient pattern matching compilation. We then leverage the link between source and memory specifications to emit efficient low-level code for value construction and pattern matching, based on high-level source code. We develop our approach in the context of immutable, monomorphic data types, for sequential code emission.

Our contributions are as follows:

- A DSL, dubbed RIBBIT and presented in Section 2, to describe the memory layout of monomorphic, immutable ADTs, with support for well-known memory tricks: struct packing, bit-stealing, ...
- The formalization of a *simplified* language (Section 3) with validity criteria for types and their representation (Section 4), novel compilation algorithms for ADT value constructors and pattern matching (Sections 5 and 6), along with their soundness proofs.
- A sketch of extensions of our formalism, including irregular data layouts and arrays (Section 7).
- A full implementation of our DSL RIBBIT, with code generation targeting LLVM IR (Section 8).
- Some initial performance evaluation and comparison (Section 9).

2 MEMORY REPRESENTATION OF ADTS

In the following section, we peel back the veil hiding the details of data layouts of rich data types through case studies of increasing complexity. In these examples, we use RIBBIT[1] to specify both

[1] https://gitlab.inria.fr/ribbit/ribbit

```
1  // Colors for Red-Black Trees
2  enum Color { Red, Black }
3
4  // Nodes for Red-Black Trees
5  struct Node
6  { c: Color, v: i64, // color and value
7    l: RBT, r: RBT } // children
8
9  // Red-Black Trees
10 enum RBT
11 { Node(Node),
12   Empty }
```

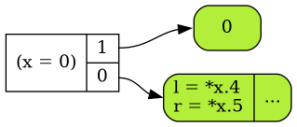
(a) Definition of Red-Black Trees

```
1  (c: Color) represented as
2  i1 with . : split . { 0 from Red | 1 from Black }
3
4  (n: Node) represented as
5  {{ (n.v as i64), (n.c as Color), i63,
6     (n.l as RBT), (n.r as RBT) }}
7
8  (t: RBT) represented as
9  split . { // empty path: inspect the whole value
10   | 0 from Empty => i64
11   | _ from Node => &<64>(t.Node as Node)
12 }
```

(b) A simple memory representation

```
1  pub fn cardinal(x: RBT) -> u32 {
2    match x {
3      Node{c:_, v:_, l, r} => 1+cardinal(l)+cardinal(r),
4      Empty => 0
5    }}
```

(c) Code for the cardinal operation



(d) A decision tree for `cardinal`

Fig. 1. Red-Black Trees with a Rust-like layout

the source type, some client code that uses it, and some non-trivial memory representation [2] showcasing the expressivity of our approach, along with some compilation results.

2.1 Gentle Representations for Record and Sum Types

A Rust-like version of *red-black trees* (RBTs) is depicted in Fig. 1. We first define `Color`, which is either the constant `Red` or `Black`. RBTs are defined through the mutually recursive types `Node` for a node of the tree, and `RBT` for the tree itself. A `Node` is a product type containing a color, a value, and its left and right children. A tree of type `RBT` is a sum type with two variants: `Empty` and `Node`. For instance, `Node{c:Red, v:1515, l:Node{c:Black, v:0, l:Empty, r:Empty}, r:Empty}` is a tree containing two integers. The `cardinal` function takes a tree and returns its cardinal using *pattern matching*. If its argument is of the "shape" of the left-hand side of a rule then the value of its right-hand side (body) is evaluated. Pattern matching is introduced by the keyword `match` and alternatives are enumerated in a list of branches the form $p \Rightarrow b$ where p is a *pattern* and b the body. Moreover, patterns can be nested, and the body can use named subterms. In our example, `Empty` yields a cardinal of 0 and `Node{c:_, v:_, l, r}` yields a cardinal of `1+cardinal(l)+cardinal(r)`.

Like most self-balancing trees, RBTs are a performance-intensive data structure. Indirections in the memory representation limit locality, result in slow memory loads, cache misses, and slowdowns of several orders of magnitude. To achieve best possible performance, it is critical to pay attention to the memory representation of our types. We would nevertheless prefer to tweak the representation without mangling its *source* representation, which provides nice data constructors and access functions that are close to intended type *semantics*. We propose to use detailed annotations to precisely describe the memory layout of data types. Unlike previous efforts [Chen et al. 2023], our annotations are fully explicit, handle sum types gracefully, and scale well to a wide variety of popular representation techniques such as bit-stealing, unboxing, packing, etc.

As our first foray into representation tweaking, we chose an almost naive representation of RBTs where trees are either an aligned pointer to a `Node`, or `Empty` represented as a 64-bit word encoding the value `1`. This representation is described using our DSL RIBBIT in Fig. 1b. In lines 4–8, we define

[2] In many cases, the memory layout of a particular type is irrelevant; RIBBIT provides *generic* representations (e.g., OCaml layout) for such situations.

```
1  enum Zarith::Integer {
2    Small(i63),
3    Large(GMP::BigInt),
4  }

5  (i : Zarith::Integer) represented as
6  split .[0, 1] { // inspect the lowest bit
7    | 1 from Small => i64 with .[1, 64] : (i.Small as i63)
8    | 0 from Large => &<64, 8>(i.Large as GMP::BigInt)
9  }
```

Fig. 2. Implementation of the Zarith integer type in RIBBIT

the representation of a single `Node` as a block, denoted by `{{...}}`. Each subterm of the original type is represented, along with the layout it should use. For instance, we decide to encode the value (field `v` of the source record) as an `i64` placed in the first field of the memory block (as we assume it is the most accessed field) with `(n.v as i64)`. We also decide to explicitly pad the structure after the color to keep the left and right children aligned (hence `i63`). The `RBT` type is an alternative: either `Empty` or `Node`. For this purpose, we use the notion of *splits*. `split p {...}` indicates a choice depending on the integer value stored at position `p`. In this case, the empty position `.` inspects the bitword (word or pointer) at the root of the tree being described. The split then contains a list of branches each containing an integer value in its left-hand side, and a layout specification in its right-hand side. Concretely, line 12 indicates that if the `RBT` value `t` is `0`, we are considering the `Empty` case and the tree is represented as an `i64`. Otherwise, on line 13, we are in the `Node` case, which is represented as a 64-bit pointer (denoted `&<64>(...)`) to a `Node` block.

This memory specification allows us to control the representation precisely: for instance, here, we do not add any indirection to left and right children, so that empty children are represented directly. For pedagogical purposes, we picked a common representation for trees in C-like languages: a pointer that is null for empty trees. Unlike C, however, the language forbids dereferencing a null pointer and users can use nice constructs such as pattern matching and field accesses. As a first example, we provide in Fig. 1d the output of our compilation procedure for the cardinal function of Fig. 1c. In the proposed low-level representation, x (the input representation of a tree in memory) is only dereferenced when the tree is not empty.

2.2 Pointer Tagging and Unboxing

Potentially-null pointers are a common trick that is easily attainable for a dedicated optimizing compiler. In fact, Rust automatically finds a similar representation. Let us explore a more delicate example taken from Zarith [Leroy and Miné 2010], an OCaml library for arbitrary-precision integers. To speed up computations, integers in zarith are either "small", represented as 63-bit integers and using usual instructions, or "big", using GMP's BigInt. The choice of representation made in Zarith is optimized and not naturally expressible via OCaml datatypes. Instead, it uses unsafe operations and the C foreign function interface. We can readily implement this type with a bespoke representation in RIBBIT, shown in Fig. 2, which leverages two new notions: bitword specifications and pointer alignment. A Zarith integer is represented in the `Small` variant as a 64-bit word with its lowest bit set to `1` and its 63 higher bits encoding the actual integer. This is specified via the `with` construct, which takes a position inside of the current bitword (here, a range of bits), and a memory specification describing its contents. The `Large` variant is represented as a 64-bit word-aligned pointer (denoted `&<64, 8>(...)`, where 8 is the number of unused bits). Using `split`, we distinguish between these two variants using the lowest bit (`.[0, 1]`, `0` is in the range, `1` is excluded), which is `1` in the `Small` variant (by specification) and `0` in the `Large` variant (thanks to pointer alignment). As before, we take care of mentioning all subterms in the type, here `i.Small` and `i.Large`, along with their representations.

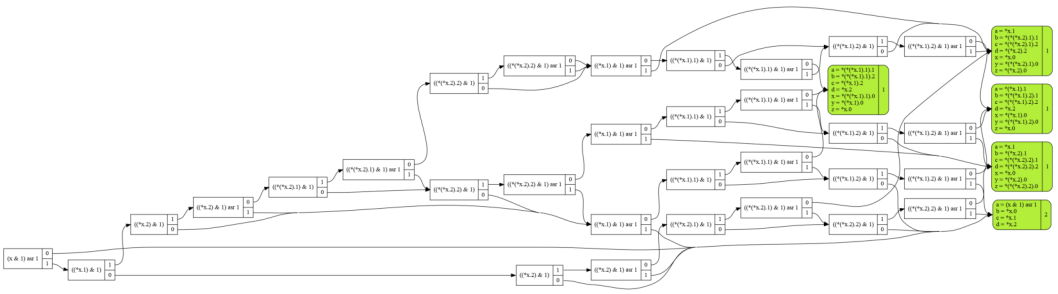
In this example, we exploit pointer alignment to embed the tag identifying the variant in the lowest bit. This optimization is generally known as *bit-stealing* and is commonly done manually in C programs. This involves implementing all case analysis by hand using masks and manual dereferencing for every manipulation of the corresponding data. This is not only error-prone, but

```
1  enum RBT {
2    Empty, Node({c:Color, v:i64, l:RBT, r:RBT})
3  }
4  represented as
5  split .[0, 1] {
6  | 0 from Node => &<64, 2>(
7    {{(_.v as i64),(_.l as RBT),(_.r as RBT)}}
8    ) with .[0,1]:(=0) with .[1,2]:(_.c as Color)
9  | 0 from Empty => i64 with . : (= 0)
10 }
```

(a) The Linux representation

```
1  match (c, v, t1, t2) {
2    Black,z,Node(Red,y,Node(Red,x,a,b),c),d
3    | Black,z,Node(Red,x,a,Node(Red,y,b,c)),d
4    | Black,x,a,Node(Red,z,Node(Red,y,b,c),d)
5    | Black,x,a,Node(Red,y,b,Node(Red,z,c,d))
6      => Node{c:Red,v:y,
7              l:Node{c:Black,v:x,l:a,r:b},
8              r:Node{c:Black,v:z,l:c,r:d)},
9    c, v, l, r => Node {c,v,l,r}
10 }
```

(b) Balancing Red-Black Trees



(c) Output of our RIBBIT compiler for the balancing code for RBTs in the Linux memory representation

This output is a decision tree whose root is on the left. Decision (square) nodes compute values from their input (low-level representation), then branch depending on this value. Round nodes denote the final returned value of the procedure (the numbering of the matched branch).

Fig. 3. Red-Black Trees in the Linux kernel

also obscures program semantics compared to pattern matching. Conversely, high-level languages have difficulty expressing such types cleanly (this example is taken from an OCaml library) and must drop down to lower-level manipulation, thus forgoing their nice high-level guarantees. Thanks to our specification, the programmer retains the full control over memory layout available in C, while enjoying the descriptive aspect of high-level constructs associated with ADTs.

We already showcased the safety result of non-null pointer dereferences "by construction" in the previous section. Our approach also provides additional constraints for the memory layouts to be correct. Here, our "bit-stolen" type is *valid* and *agrees* with the user source type because:

- All source subtypes (Small and Large) are properly accounted for in the memory type: `i.Small as ...` and `i.Large as ...`.
- Split branches in Fig. 2 are distinct: their left-hand sides all contain different values (`0` and `1`) that correspond to all possible variants (Small and Large).
- The *bit-stealing* process is "legal": the first bit is always unused in the Small case and fits within address alignment bits in the Large case.

The formalization of such *validity criteria* is one of our contributions, described in Section 4.

2.3 Real World: Red-Black Trees in Linux

As a final case study, let us look again at RBTs, but this time with a highly optimized representation originally found in the Linux kernel to model device trees [Torvalds 2023]. The representation is hand-tweaked to take as little space as possible by embedding both the tag *and* the color in pointer alignment bits. The original version keeps a pointer to the parent node, which we eschew for pedagogical purposes. The RIBBIT code is shown in Fig. 3a. The source type is the same as in our first representation in Fig. 1a. The main novelty is that we combine `with` and subterm types in line 8 to store the `Color` subterm inside a second alignment bit. Since the source type does not change, the rest of the program, such as the `cardinal` function we showcased, will work unmodified.

$$\tau ::= t \mid \mathrm{Int}_\ell \mid \&\tau \mid \langle \tau_0, \ldots, \tau_{n-1} \rangle \mid K_0(\tau_0) + \ldots + K_{n-1}(\tau_{n-1}) \quad \text{(Type expressions)}$$

$$\Gamma ::= \{t_1 \mapsto \tau_1; \ldots; t_n \mapsto \tau_n\} \quad \text{(Type environment)} \qquad \pi ::= \epsilon \mid \pi.* \mid \pi.i \mid \pi.K \quad \text{(Paths)}$$

Fig. 4. Grammar for source types and paths

A similar change of representation in another context would often require rewriting a significant amount of code.

RBTs famously rely on a fairly complex balancing step, which redistributes colors depending on the internal invariant of the data structure. Thanks to nested patterns and "or" patterns, this step can be expressed very compactly using pattern matching, as shown in Fig. 3b. This pattern matching simultaneously inspects four arguments: the current color `c`, the current value `v` and the subtrees `t1` and `t2`. Writing the corresponding C code for an optimized memory representation is rather painful, requiring bitmasks before dereferencing and careful bit manipulations.

Since this pattern matching is at the core of a performance-sensitive data structure, we naturally want it to be as efficient as possible. Optimized decision trees result in highly non-trivial code, as can be seen in Fig. 3c. Many pattern matching implementations come with clever techniques to output optimized decision trees [Kosarev et al. 2020; Maranget 2008; Sestoft 1996]. Unfortunately, these approaches are designed for values that directly reflect the structure of the underlying ADT. As we have seen, this is not necessarily the case for optimized memory representations: some part of the value could be used to discover the shape of some other part, such as whether a dereferencing is legal or not. One of our contributions is a novel algorithm that takes such fine-grained dependencies into account in order to compile source-level pattern matching to low-level code. This algorithm is parametrized by a *memory specification*, which is part of our input language which we define now. The next section focuses on core features of our input language. In Section 7 and Appendix B, we describe extensions that allow our formalism to capture more intricate layouts, such as WebKit-like *NaN-boxing*, which we present in Appendix A.

3 SPECIFICATIONS FOR ALGEBRAIC DATA TYPES

We now formalize our input language. As before, we present a two-tiered view: *source* types used for programming and following a common presentation of ADTs and *memory* specifications detailing how to represent them in memory. We also detail the shape of programs we consider for our compilation algorithms. For simplicity, we initially consider only "regular" memory specifications, which limit inlining of sum types. We show how to relax this restriction later on.

3.1 Source Types

Our source language is composed of simple (monomorphic, immutable) algebraic data types whose grammar is presented in Fig. 4. We denote types using τ and type variables with t. We denote all tuples with angle brackets, for instance $\langle \mathrm{Int}_{64}, \mathrm{Int}_{64} \rangle$ for the type of 64-bit integer pairs. Constructors of sums are marked with a capital letter, for instance $\mathrm{Some}(t) + \mathrm{None}$ is an option type. In examples, we use K as shortcut for $K(\langle\rangle)$. All references are marked with an ampersand, for instance $\&\mathrm{Int}_8$ is the type of references to 8-bit integers. In addition, we use Γ to denote type environments, i.e., maps from type variables t to types τ, which we use to model recursive types.

Paths and Focusing. Paths, denoted π in Fig. 4, indicate a position into a value, a type, or a pattern. For instance, in the pattern $B(\&x)$, the variable x is at the position $\pi = .B.*$, where $*$ denotes dereferencing. We also generally denote $\mathbf{focus}\,(\pi, \ldots)$ the term at this position. For instance, $\mathbf{focus}\,(.B, B(\&x)) = \&x$. The full definition of focus can be found in Appendix C.

$$
\begin{array}{lll}
\widehat{\tau} ::= \widehat{t} & \text{(variable)} & \widehat{\pi} ::= \epsilon \quad \text{(Empty path)} \\
\quad \mid (\pi \ \mathit{as}\ \widehat{\tau}) & \text{(subterm at (source) position } \pi) & \quad \mid \widehat{\pi}.* \quad \text{(Dereference)} \\
\quad \mid (= w) & \text{(singleton } w \in \mathbb{W}) & \quad \mid \widehat{\pi}.i \quad (i\text{-th field access}) \\
\quad \mid \mathrm{Word}_\ell \bowtie_{0 \le i < n} \widehat{\pi}_i : \widehat{\tau}_i & (\ell\text{-bit-wide word}) & \quad \mid \widehat{\pi}.[i, j] \quad \text{(Bit selection)} \\
\quad \mid \mathrm{Ptr}_{\ell,a}(\widehat{\tau}_*) \bowtie_{0 \le i < n} \widehat{\pi}_i : \widehat{\tau}_i & \text{(pointer to } \widehat{\tau}_* \text{ with alignment } a) & \quad \mid \widehat{\pi}.f \quad f \in \{\neg; \wedge w; \ldots\} \quad \text{(Word operations)} \\
\quad \mid \{\!\{\widehat{\tau}_0, \ldots, \widehat{\tau}_{n-1}\}\!\} & \text{(block/struct with } n \text{ fields)} & \\
\quad \mid \mathrm{Split}\ (\widehat{\pi})\ \{w_i : \mathcal{K}_i \Rightarrow \widehat{\tau}_i \mid 0 \le i < n\} & \text{(split w. } n \text{ branches)} &
\end{array}
$$

Fig. 5. Grammar of memory types $\widehat{\tau}$ and memory paths $\widehat{\pi}$.

3.2 Memory Specifications

The second ingredient of our language is a memory description for every source type τ. This *memory type* describes how the given type will be represented in memory. We consider an abstraction of memory similar to LLVM IR's types, shown in Fig. 5. At this representation level, we are bit-precise, yet abstract away some architecture-dependent details such as endianness and machine pointer size and alignment. Handling of these details is delegated to the underlying compiler backend (in our case, LLVM: see Section 8). For the purpose of this formalization, each pointer solely consists of an address and unused alignment bits at the lower end.[3]

Let us first ignore the "Split" alternation, and focus on the rest of the grammar. As a convention, all memory elements are given a hat, for instance $\widehat{\tau}$ for memory types.

Simple Memory Types Construction. In Fig. 5, type variables, which we use to handle recursive types, are denoted by $\widehat{t}$. $(\pi \ \mathit{as}\ \widehat{\tau})$ indicates that the subterm at the position π *in the source type* will be represented by the memory type $\widehat{\tau}$. A singleton type, denoted $(= w)$, is a memory type whose only possible value is the immediate w. Structure types are denoted by $\{\!\{\widehat{\tau}_0, \ldots, \widehat{\tau}_{n-1}\}\!\}$. Memory words can be of two natures: Word_ℓ is the type of words of size ℓ and $\mathrm{Ptr}_{\ell,a}(\widehat{\tau})$ is the type of pointers of size ℓ, with a unused bits due to address alignment, and pointing to a value of type $\widehat{\tau}$. In both cases, they can be accompanied by *bitword contents*, denoted $\ltimes\widehat{\pi} : \widehat{\tau}$ which indicates that the value at position $\widehat{\pi}$ follows the memory type $\widehat{\tau}$ (observe the priority of ':' over '$\ltimes$').

Memory path operations include pointer dereferencing and field accesses in structs, as well as (constant) bitwise and arithmetic operations that manipulate memory words. The set of available bitwise and arithmetic operations shown in the grammar is arbitrary; it depends on the targeted instruction set. In the rest of this paper, we will mainly use the extraction operation $[i, j]$, whose semantics are "extract bits from i inclusive to j exclusive, with 0 denoting the least significant bit".

Example 3.1 (Memory Type Description). Let us consider the source types $\tau_{\mathrm{int}} = \mathrm{Int}_{64}$ and $\tau_{\mathrm{tup}} = \langle \mathrm{Int}_{32}, \&\tau, \mathrm{Int}_8 \rangle$. We have a straightforward memory encoding for τ_{int}: $\widehat{\tau}_{\mathrm{int}} = (\epsilon \ \mathit{as}\ \mathrm{Word}_{64})$ which encodes a 64-bit integer as "itself". For τ_{tup}, we choose to represent the tuple with a "struct", and decide to order its components as follows: first, we store the Int_{32} (position .0 in the source type), then the Int_8 (position .2 of the source type), then 24 bits of padding (with zeroes, hence the bitword content $\ltimes\epsilon : (= 0)$), then the τ pointer on 64 bits. We group the Int_8, Int_{32} and 24 padding bits together, thus leading to $\widehat{\tau}_{\mathrm{tup}} = \{\!\{(.0 \ \mathit{as}\ \mathrm{Word}_{32}), (.2 \ \mathit{as}\ \mathrm{Word}_8), \mathrm{Word}_{24} \ltimes \epsilon : (= 0), (.1 \ \mathit{as}\ \mathrm{Ptr}_{64}(\widehat{\tau}))\}\!\}$.

Split Memory Types. Using a memory-level specification to describe type layouts and pattern matching is convenient, but it raises several new difficulties that are not obvious in more direct

[3] Other schemes, such as negative pointers, would require minor extensions to our memory types. Precise adaptations to more or less unusual architectures are orthogonal to our setup.

$$
\begin{array}{lll}
e ::= \texttt{let } x = e \texttt{ in } e' & & \text{(Bindings)} \\
\quad \mid \mathrm{v} & & \text{(Constructor)} \\
\quad \mid \texttt{match } x \ \{p_0 \to e_0, \ldots, p_{n-1} \to e_{n-1}\} & & \text{(Destructor)} \\
\quad \mid \ldots & &
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{v} ::= x \mid c \in \mathbb{C} \mid \&\mathrm{v} & & \\
\quad \mid \langle \mathrm{v}_0, \ldots, \mathrm{v}_{n-1} \rangle \mid K(\mathrm{v}) & & \text{(Values)} \\
p ::= \_ \mid x \mid p\text{"|"}p \mid \&p & & \\
\quad \mid \langle p_0, \ldots, p_{n-1} \rangle \mid K(p) & & \text{(Patterns)}
\end{array}
$$

Fig. 6. Grammar for the source language ($x \in \mathrm{Var}$)

approaches. One notable such difficulty is that, notably in the case of sum types, *memory accesses can have dependencies that do not follow source-level nested subterms*. Let us show this on an example.

Example 3.2 (Non-Nested Dependencies). We consider the Zarith example from Section 2.2, using a large integer type to model `GMP::BigInt`. Let $\tau_z = \mathrm{Small}(\mathrm{Int}_{63}) + \mathrm{Large}(\mathrm{Int}_{128})$. Once again, we choose a compact representation exploiting bit-stealing: values of the form $\mathrm{Small}(n)$ map to 64-bit words whose lowest bit is always 1; values of the form $\mathrm{Large}(n)$ map to pointers aligned on at least one byte. Now, consider the pattern $\mathrm{Large}(1) \mid \mathrm{Large}(2)$. In the source version, it is clear we must establish whether the head constructor is Large before matching its contents. In the memory version, things are less clear: all values are "simply" words, and both patterns are supposed to dereference a pointer to access integers. For safety however, we *must* ensure that the word is indeed a pointer before dereferencing it to observe its contents!

To enforce such constraints, we introduce *splits* whose purpose is to model constraints of the form "if the numeric value at a given position in memory is w, then we follow the memory type associated with w". This lets us model choices in representation, and clarify dependencies in non-nested cases. We denote $\mathrm{Split}\ (\widehat{\pi})\ \{B\}$ a split type with a discriminant position $\widehat{\pi}$ and a set B of branches of the form $w_i : \mathcal{K}_i \Rightarrow \widehat{\tau}_i$. It expresses that if the value at position $\widehat{\pi}$ in memory is w_i, then it corresponds to a source constructor in the set $\mathcal{K}_i$, dubbed the branch *provenance*, and its memory type is $\widehat{\tau}_i$.

Example 3.3 (Splits in Zarith). Our choice of representation in Example 3.2 is expressed by:

$$
\widehat{\tau_z} = \mathrm{Split}\ (.[0,1])\ \left\{ \begin{array}{l} 1 : \{\mathrm{Small}\} \Rightarrow \mathrm{Word}_{64} \ltimes \Big([0,1] : (= 1) \Big) \ltimes \Big([1,64] : (.\mathrm{Small}\ \mathit{as}\ \mathrm{Word}_{63}) \Big) \\ 0 : \{\mathrm{Large}\} \Rightarrow \mathrm{Ptr}_{64,1}((.\mathrm{Large}\ \mathit{as}\ \mathrm{Word}_{128})) \ltimes \Big([0,1] : (= 0) \Big) \end{array} \right\}
$$

Since there are two possible representations depending on the head constructor, we use a "split" type with two branches. In the first branch, the provenance {Small} indicates that it represents values of the $\mathrm{Small}(\mathrm{Int}_{63})$ case in the source type. Its associated memory type is $\mathrm{Word}_{64} \ltimes ([0,1] : (= 1)) \ltimes ([1,64] : (.\mathrm{Small}\ \mathit{as}\ \mathrm{Word}_{63})$: a 64-bit word whose lowest bit is set to 1 and whose 63 highest bits encode the subterm at position .Small (that is, the 63-bit integer value from the source type). In the second branch, the provenance is {Large} and the memory type representing $\mathrm{Large}(\mathrm{Word}_{128})$ is $\mathrm{Ptr}_{64,1}((.\mathrm{Large}\ \mathit{as}\ \mathrm{Word}_{128})) \ltimes [0,1] : (= 0)$: a byte-aligned pointer to a 128-bit word encoding the integer value at position .Large in the source type, with its lowest bit ("alignment bit") set to 0. Finally, the split discriminant $[0,1]$ indicates how to tell these two cases apart: by looking at the lowest bit, which is 1 for the Small variant and 0 for the Large variant.

3.3 Source Programs and Compilation Setup

As setup for our compilation algorithms, we formalize input programs as simplified expressions (shown in Fig. 6) where every expression is let-bound, akin to A-normal form (Sabry and Felleisen [1993]). Our main interest lies in two categories of expressions: *constructors*, i.e., value allocation; and *destructors*, i.e., pattern matching. Constructors of ADTs, denoted v, are simply values with a syntax reminiscent of types, along with constants $c \in \mathbb{C}$ and variables x. They syntactically cannot

contain expressions (which should be let-bound). To showcase proper compilation of complex values, sub-values can be nested directly. For instance, Node($\langle$Red, 42, $x, y\rangle$) is a single constructor. For simplicity, we do not support recursive values (types might be recursive).

Destructors correspond to pattern matching, shown in Section 2: they allow to filter out value shapes at the left-hand side of rules, bind the variables they contain, and return the expression on the right-hand side. Patterns, denoted p, consist of constructor patterns, along with variables, wildcards (_), and or-patterns ("|"). For instance, given the source type $\tau = A(\mathrm{Int}_1) + B(\&\mathrm{Int}_{32})$, the pattern $B(\&\_)$ denotes any value of the given "shape" with an ignored sub-value at the position of _.

For both values and patterns, we consider a *focusing* operation, similar to the one described for types. For instance, **focus** $(.B.*, B(\&x)) = x$. The full definition of focus can be found in Appendix C.

Expressions generally contain numerous other operations, notably function definition and application. We consider the compilation of functions to be orthogonal to the contents of this article, and will not detail it. In the rest of this article, we will focus on compilation of constructors in Section 5 and of destructors in Section 6. However, for these two compilation procedures to be correct, we first need to formalize constraints on user-provided memory layouts.

4 VALIDITY AND AGREEMENT OF TYPES

To ensure the correctness of a given memory specification, we consider two complementary notions: intrinsic *validity* of a memory type, and *agreement* between source and memory types. We first define utility operations on memory types.

4.1 Specialization

In many cases, we need to identify "the part of a memory type corresponding to the variant K". For instance, when compiling the pattern $p = K(p')$, we want to know which parts of the memory type are relevant to the pattern p'. For this purpose, $\widehat{\tau}/K$ filters a type $\widehat{\tau}$ by K. It proceeds as a map over $\widehat{\tau}$ and returns a specialized, split-free memory type. For splits, it returns the branch whose provenance set contains K. The full definition is in Appendix D; we present it here on an example.

Example 4.1 (Specialization of the Zarith memory type). Let us specialize $\widehat{\tau_z}$ from Example 3.3 for the constructor Large. By looking at provenances, we establish that only the second branch of the split definition is relevant. We then explore that type recursively and obtain $\widehat{\tau_z}/\mathrm{Large} = \mathrm{Ptr}_{64,1}((.\mathrm{Large}\ as\ \mathrm{Word}_{128})) \ltimes [0,1] : (= 0)$.

4.2 Focusing

The memory version of focusing, which extracts nested memory terms from a parent term at a position specified by a memory path, is denoted $\widehat{\textbf{focus}}$ and defined in Appendix D. Unlike its source version, $\widehat{\textbf{focus}}$ is not complete: indeed, it is unclear what the focus of a split should be when the discriminant itself is not under focus. We thus define $\widehat{\textbf{focus}}$ to be invalid on splits. For all other constructs, it follows intuitions from the source. For instance, $\widehat{\textbf{focus}}\left(.[0,1], \mathrm{Ptr}_{64,1}(\dots) \ltimes [0,1] : (= 0)\right) = (= 0)$.

4.3 Validity of Memory Specifications

We first define a validity judgment denoted $\widehat{\Gamma} \vDash \widehat{\tau} : \widehat{\kappa}$. The kind $\widehat{\kappa}$ is either **Spec**, representing unsized numerical data and only used to specify bitword contents, **Block** representing contiguous memory values and never used in bitwords, or **Word** representing bitwords and usable anywhere.

The full judgment is defined in Appendix D; we show the rules concerning words and splits.

$$\textsc{WordValidity}\quad \frac{\widehat{\Gamma} \vDash \widehat{\tau}_i : \widehat{\kappa}_i \in \{\mathbf{Spec}; \mathbf{Word}\} \quad \widehat{\pi}_i \text{ valid in a word of size } \ell \quad \forall j \neq i, \widehat{\pi}_i \text{ and } \widehat{\pi}_j \text{ have distinct bit ranges}}{\widehat{\Gamma} \vDash \mathrm{Word}_\ell \bigbowtie_{0 \leq i < n} \widehat{\pi}_i : \widehat{\tau}_i : \mathbf{Word}}$$

$$\textsc{SplitValidity}\quad \frac{\widehat{\Gamma} \vDash \widehat{\tau}_i : \widehat{\kappa} \quad \forall j \neq i, \mathcal{K}_i \cap \mathcal{K}_j = \varnothing \quad \widehat{\mathbf{focus}}\,(\widehat{\pi}, \widehat{\tau}_i) = \{(= w_i)\}}{\widehat{\Gamma} \vDash \mathrm{Split}\,(\widehat{\pi}) \left\{\overline{w_i : \mathcal{K}_i \Rightarrow \widehat{\tau}_i}\right\} : \widehat{\kappa}}$$

In the WordValidity rule, we enforce that bitword contents should be specs or words. We also ensure that paths must behave properly: they should adequately fit in the word size, and not overlap each other. This constraint is essential for $\widehat{\mathbf{focus}}$ to be defined on words. In the SplitValidity rule, we ensure that provenances in distinct branches are disjoint, and that focusing on the discriminant indeed yields the word on the left-hand side. These constraints ensure that specializing a type with a tag always returns a single result.

Example 4.2 (Invalid Memory Type: C Unions). This validity judgment already rejects non-trivial user mistakes. Let us emulate a traditional C union layout for τ_z, which takes up as much space as the largest variant (Large): $\widehat{\tau}_{\text{bad}} = \mathrm{Word}_{128} \ltimes [0, 63] : (.\mathrm{Small}\ \mathit{as}\ \mathrm{Word}_{63}) \ltimes \epsilon : (.\mathrm{Large}\ \mathit{as}\ \mathrm{Word}_{128})$. $\widehat{\tau}_{\text{bad}}$ is not valid: the **Word** kinding rule does not apply because $[0, 63]$ and ϵ overlap. Since it lacks distinguishing data to use as a discriminant, this layout is not expressable as a split.

4.4 Agreement between Source and Memory Types

We now state the relationship between a given source type and its memory specification.

Definition 4.3 (Agreement). Let τ a source type and $\widehat{\tau}$ a memory type. We say that $\widehat{\tau}$ *represents* τ, or *agrees* with τ, and we write $\widehat{\tau} \Vdash \tau$, if the following conditions hold:

All subterms bind source subtypes to their valid representation. (Subterm Coherence)
Either τ and $\widehat{\tau}$ are leaves (i.e., $\tau = \mathrm{Int}_\ell$ and $\widehat{\tau} = \mathrm{Word}_\ell$) or for all $\widehat{\pi}$ such that $\widehat{\mathbf{focus}}\,(\widehat{\pi}, \widehat{\tau}) = (\pi\ \mathit{as}\ \widehat{\tau}_\pi)$, we have $\widehat{\tau}_\pi \Vdash \mathbf{focus}\,(\pi, \tau)$.

Provenances and split branches are coherent with the source type. (Provenance Coherence)
For all $\widehat{\pi}$ s.t. $\widehat{\mathbf{focus}}\,(\widehat{\pi}, \widehat{\tau}) = \mathrm{Split}\,(\widehat{\pi}')\left\{w_i : \mathcal{K}_i \Rightarrow \widehat{\tau}_i\right\}$, for each i and each $K \in \mathcal{K}_i$, K is a head constructor of τ and $\widehat{\tau}/K \Vdash K(\tau')$ where $\tau' = \mathbf{focus}\,(.K, \tau)$.

All source subtypes are represented within the memory type. (Coverage)
For every π that leads to a leaf in τ (i.e., $\mathbf{focus}\,(\pi, \tau) = \mathrm{Int}_\ell$), $\widehat{\tau}$ *covers* π: the memory type $\widehat{\tau}' = \widehat{\tau}/\pi$ contains a subterm type for a source position π_0 prefix of π. More precisely, there exist source and memory paths $\pi_0, \widehat{\pi}_0$ such that $\widehat{\mathbf{focus}}\,(\widehat{\pi}_0, \widehat{\tau}') = (\pi_0\ \mathit{as}\ \widehat{\tau}'')$ and $\pi_0 \preceq \pi$.

Memory types provide a way to tell branches of sum types apart. (Distinguishability)
For all distinct head constructors of τ, K and K', $\widehat{\tau}$ *distinguishes* K and K'. More precisely, there exists a memory path $\widehat{\pi}$ such that $\widehat{\mathbf{focus}}\,(\widehat{\pi}, \widehat{\tau}/K) = (= w)$, $\widehat{\mathbf{focus}}\,(\widehat{\pi}, \widehat{\tau}/K') = (= w')$ and $w \neq w'$.

As a first example, we show that our memory specification for Zarith agrees with its source type.

Example 4.4 ($\widehat{\tau_z}$ Agrees with τ_z). The memory type

$$\widehat{\tau_z} = \mathrm{Split}\,([0, 1]) \left\{\begin{array}{l} 1 : \{\mathrm{Small}\} \Rightarrow \mathrm{Word}_{64} \ltimes [0, 1] : (= 1) \ltimes [1, 64] : (.\mathrm{Small}\ \mathit{as}\ \mathrm{Word}_{63}) \\ 0 : \{\mathrm{Large}\} \Rightarrow \mathrm{Ptr}_{64,1}((.\mathrm{Large}\ \mathit{as}\ \mathrm{Word}_{128})) \ltimes [0, 1] : (= 0) \end{array}\right\}$$

meets all four criteria of Definition 4.3 for $\tau_z = \mathrm{Small}(\mathrm{Int}_{63}) + \mathrm{Large}(\mathrm{Int}_{128})$. We define

$$\widehat{\tau}_{\text{Small}} = \widehat{\tau_z}/\mathrm{Small} = \mathrm{Word}_{64} \ltimes [0, 1] : (= 1) \ltimes [1, 64] : (.\mathrm{Small}\ \mathit{as}\ \mathrm{Word}_{63})$$
$$\widehat{\tau}_{\text{Large}} = \widehat{\tau_z}/\mathrm{Large} = \mathrm{Ptr}_{64,1}((.\mathrm{Large}\ \mathit{as}\ \mathrm{Word}_{128})) \ltimes [0, 1] : (= 0)$$

Subterm Coherence: immediate since there are no subterms outside of the split (i.e., no subterms are reachable through focusing).
Provenance Coherence: Small and Large are head constructors of the type τ_z. We now show that $\widehat{\tau}_{\text{Small}} \Vdash \text{Small}(\text{Int}_{63})$ and $\widehat{\tau}_{\text{Large}} \Vdash \text{Large}(\text{Int}_{128})$. Provenance coherence and distinguishability are immediate, since both source types have only one constructor and neither memory type contains a split. Coverage for each subtype follows from coverage for the root type (see the next item). Finally, subterm coherence holds since $\text{Word}_{63} \Vdash \text{Int}_{63}$ and $\text{Word}_{128} \Vdash \text{Int}_{128}$ ("leaf" coverage case).
Coverage: both subterms are covered; $\widehat{\tau}_{\text{Small}}$ contains $(.\text{Small } \textit{as } \text{Word}_{63})$ at position $.[1, 64]$ and $\widehat{\tau}_{\text{Large}}$ contains $(.\text{Large } \textit{as } \text{Word}_{128})$ at position $.*$.
Distinguishability: we have $\widehat{\textbf{focus}}\,(.[0,1], \widehat{\tau}_{\text{Small}}) = (= 1)$ and $\widehat{\textbf{focus}}\,(.[0,1], \widehat{\tau}_{\text{Large}}) = (= 0)$, therefore $\widehat{\tau_z}$ distinguishes Small and Large.

The rest of this section is dedicated to counter-examples.

Example 4.5 (Non-Coverage: Tags without Payload). A simple tag without payloads, similar to C enums, is not sufficient to encode arbitrary sum types with non-unit variants, as it does not meet the coverage criterion: $\text{Split}\,(\epsilon)\,\{0 : \{\text{Large}\} \Rightarrow \text{Word}_{32} \ltimes \epsilon : (= 0), 1 : \{\text{Small}\} \Rightarrow \text{Word}_{32} \ltimes \epsilon : (= 1)\}$ does not cover τ_z as it does not include the subterms at positions .Small and .Large.

Example 4.6 (Non-Distinguishability). Let $\widehat{\tau} = \{\!\{(.\text{Large } \textit{as } \text{Word}_{128}), (.\text{Small } \textit{as } \text{Word}_{64})\}\!\}$. It includes both variants' subterms in distinct struct fields but provides no way to distinguish the variants. It thus does not meet the distinguishability criterion for τ_z: we have $\widehat{\tau}/\text{Large} = \widehat{\tau}/\text{Small} = \{\widehat{\tau}\}$; it lacks a discriminant that differs between Large and Small.

Note that these criteria allow for simple inlined structs and nested fields, but do not allow for complex mangling of nested values. Indeed, distinguishability mandates that every head constructor is identified in a split that occurs "before" its nested constructors' distinguishing splits. This introduces an unnecessary dependency between distinction of nested constructors and head constructors. Furthermore, provenances consist solely of head constructors, preventing specialization of deeply nested source variants. In other words, it is impossible to rearrange deeply nested values or flatten inductive data structures. These simplifications are done for pedagogical purposes, and we sketch how to lift them in Section 7.1 with richer, nested provenances and more elaborate criteria. The full formalism used in RIBBIT and shown in appendices uses these richer notions.

5 CONSTRUCTORS: BUILDING VALUES IN MEMORY

As an appetizer for things to come, we start with the compilation of values. Our goal is to compile a source value into the corresponding *memory value* according to a given memory type. As a reminder, source values are composed of tuples, variants, constants, and variables.

5.1 Memory Values

Memory values, defined below and denoted $\widehat{v}$, can be seen as pieces of code that build values in memory. Following memory types, bitword values are denoted $\text{Word}_\ell(w)$ and $\text{Ptr}_{\ell,a}(\widehat{v})\ (w)$, where w are concrete integers. For pointers, w fits in alignment bits. Struct values are denoted $\{\!\{\widehat{v}_0, \ldots, \widehat{v}_n\}\!\}$. For example, $\text{Ptr}_{64,1}(\text{Word}_{128}(42))\ (0)$ is a 64-bit wide pointer to a 128-bit wide word encoding the integer value 42, with the pointer's lowest bit (declared as an alignment bit) set to 0.

$$\widehat{v} ::= \text{Word}_\ell(w) \mid \text{Ptr}_{\ell,a}(\widehat{v})\ (w) \mid \{\!\{\widehat{v}_0, \ldots, \widehat{v}_{n-1}\}\!\}$$

5.2 From Source to Memory Values

Memory types provide a rich description. Splits not only have a discriminant and various subtypes, but also a *provenance* indicating which source variants could lead to a branch. Similarly, subterm

types (π *as* $\widehat{\tau}$) indicate the position in the source type to which they correspond. This information is unnecessary to specify the memory representation, since it is about the source type. However, it is exactly what is needed to guide the translation from source values to memory values. The key idea is to do a cross-examination of both source value and memory type: the value contains head constructors which let us select relevant parts of the memory type (using specialization, as defined in Appendix D) and synthesize proper memory values, and so on until a leaf is found. For this purpose, we define two mutually recursive functions: **ty2val**, which synthesizes a memory value from a type, and **val2mem**, which explores a source value. Both take a source value v, a filtered memory type $\widehat{\tau}$, and return a memory value.

Synthesized Memory Values from Types. **ty2val** takes a *filtered* type, i.e., without any splits. It proceeds by induction on the memory type, rewriting it as a memory value: singleton types are turned into constant values, type variables are looked up in the environment. For subterm types (π *as* $\widehat{\tau}$), it first finds the subvalue at position π within v denoted $v_\pi = \mathbf{focus}\ (\pi, v)$. $\widehat{\tau}$ indicates the memory type guiding the representation at this point, so we can recursively call $\mathbf{val2mem}(v_\pi, \widehat{\tau})$. In struct, pointer and word cases, we recursively call **ty2val** on each nested memory type *with the same source value* to populate the struct, pointer or word. For words and pointers, we want to synthesize a concrete bitword. For this, we should merge all bitword contents of the form $\widehat{\pi_i} : \widehat{v_i}$ into a single concrete numeric value w such that for every i, the evaluation of $\widehat{\pi_i}$ on w yields $\widehat{v_i}$. This is done by **fillbitword**, which returns the sum of every $\widehat{v_i}$ interpreted as an integer, truncated and shifted to fit into the bit range of $\widehat{\pi_i}$. This filling method is correct owing to memory type validity, which asserts that the memory paths $\widehat{\pi_i}$ never overlap (i.e., their bit ranges are distinct).

$\mathbf{ty2val}(v)\{$

$$
\begin{array}{ll}
\widehat{t} & \longrightarrow \mathbf{val2mem}\left(v, \widehat{\Gamma}(\widehat{t})\right) \\
(= w) & \longrightarrow w \\
(\pi\ \mathit{as}\ \widehat{\tau}) & \longrightarrow \mathbf{val2mem}\left(\mathbf{focus}\ (\pi, v), \widehat{\tau}\right) \\
\{\!\{\widehat{\tau_0}, \ldots, \widehat{\tau_{n-1}}\}\!\} & \longrightarrow \{\!\{\widehat{v_0}, \ldots, \widehat{v_{n-1}}\}\!\} \text{ where } \widehat{v_i} = \mathbf{ty2val}(v, \widehat{\tau_i}) \\
\mathrm{Word}_\ell \underset{0\le i<n}{\bowtie} \widehat{\pi_i} : \widehat{\tau_i} & \longrightarrow \mathrm{Word}_\ell\left(\mathbf{fillbitword}(\widehat{\pi_0} : \widehat{v_0}, \ldots, \widehat{\pi_{n-1}} : \widehat{v_{n-1}})\right) \text{ where } \widehat{v_i} = \mathbf{ty2val}(v, \widehat{\tau_i}) \\
\mathrm{Ptr}_{\ell,a}(\widehat{\tau}) \underset{0\le i<n}{\bowtie} \widehat{\pi_i} : \widehat{\tau_i} & \longrightarrow \begin{array}{l}\mathrm{Ptr}_{\ell,a}(\mathbf{ty2val}(v, \widehat{\tau}))\ (\mathbf{fillbitword}(\widehat{\pi_0} : \widehat{v_0}, \ldots, \widehat{\pi_{n-1}} : \widehat{v_{n-1}})) \\ \text{where } \widehat{v_i} = \mathbf{ty2val}(v, \widehat{\tau_i})\end{array}
\end{array}
$$

$\}$

Val2Mem. **val2mem** explores the value to direct the usage of **ty2val**. Variables and constant types (such as Int_{64}) are immediately translated. For values headed by a constructor K, we consider $\widehat{\tau'} = \widehat{\tau}/K$, that is, the specialization of $\widehat{\tau}$ by K as defined in Section 4.1. Specialization returns a memory type without any splits which covers all potential memory types that apply to K. For values without a head constructor, we keep the memory type as-is, as it cannot contain any splits thanks to (Provenance Coherence) from Definition 4.3. In both cases, $\mathbf{ty2val}(v, \widehat{\tau'})$ returns the memory value which fits $\widehat{\tau'}$ and contains subvalues of v at the position of each subterm type.

$$
\begin{array}{cc}
\mathbf{val2mem}(w, \mathrm{Word}_\ell) = \mathrm{Word}_\ell(w) & \mathbf{val2mem}(K(v'), \widehat{\tau}) = \mathbf{ty2val}(K(v'), \widehat{\tau}/K) \\
\mathbf{val2mem}(x, \widehat{\tau}) = x & \mathbf{val2mem}(v, \widehat{\tau}) = \mathbf{ty2val}(v, \widehat{\tau})
\end{array}
$$

Example 5.1 (τ_z and $\widehat{\tau_z}$ Values). Recall $\tau_z = \mathrm{Small}(\mathrm{Int}_{63}) + \mathrm{Large}(\mathrm{Int}_{128})$ and

$$
\widehat{\tau_z} = \mathrm{Split}\ (.[0,1]) \begin{Bmatrix} 1 : \{\mathrm{Small}\} \Rightarrow \mathrm{Word}_{64} \ltimes [1,64] :\ (.\mathrm{Small}\ \mathit{as}\ \mathrm{Word}_{63}) \\ 0 : \{\mathrm{Large}\} \Rightarrow \mathrm{Ptr}_{64,1}((.\mathrm{Large}\ \mathit{as}\ \mathrm{Word}_{128})) \end{Bmatrix}
$$

Let us consider the source value $v_s = \mathrm{Small}(42)$ and compute its memory value. The second rule of **val2mem** applies: we first specialize the memory type and get $\widehat{\tau_s} = \widehat{\tau_z}/\mathrm{Small} = \mathrm{Word}_{64} \ltimes [0,1]$:

$(= 1) \ltimes [1, 64] : (.\text{Small } as \text{ Word}_{63})$, then compute $\mathbf{ty2val}(v_s, \widehat{\tau}_s)$. We have $\mathbf{ty2val}(v_s, (= 1)) = 1$ and $\mathbf{ty2val}(v_s, (.\text{Small } as \text{ Word}_{63})) = \mathbf{val2mem}(42, \text{Word}_{63}) = \text{Word}_{63}(42)$, hence $\mathbf{ty2val}(v_s, \widehat{\tau}_s) = \text{Word}_{64}\big(\mathbf{fillbitword}([0, 1] = 1 \ltimes [1, 64] = \text{Word}_{63}(42))\big) = \text{Word}_{64}(1 + 42 \times 2) = \text{Word}_{64}(85)$.

Remark on Regularity. During translation to memory values, the contents of used variables can be used directly, either by reference or by copy. Indeed, the "regularity" limitation from Section 4 asserts direct subterms are all present in the memory type, forbidding mangling of original subvalues. This still allows inlined structs, but disallows some complex schemes. Lifting this limitation requires a more complex procedure to synthesize values, which we sketch in Section 7.1.

6 DESTRUCTORS: FROM PATTERNS TO DECISION TREES

Compilation of patterns is done in three phases: first, we translate *source* patterns into *memory patterns*. We then use a bespoke intermediate representation dubbed "memory trees" to compile memory patterns to decision trees. Finally, we enrich the resulting decision trees to bind variables to appropriate positions in memory. We assume patterns are exhaustive, non-redundant and contain no variable shadowing, which can be achieved with well-known techniques [Maranget 2007].

6.1 Memory Patterns

Memory patterns, defined below and generally denoted $\widehat{p}$, are patterns that work directly on memory values. They contain a mix of elements from regular patterns and of constructions found in memory types. Similar to source patterns, they can contain variables x, wildcards _, and alternatives $\widehat{p} \mid \widehat{p}'$. Following memory types, patterns can also be bitwords such as Word_ℓ and $\text{Ptr}_{\ell,a}(\widehat{p})$, with some associated *specification patterns* written $\ltimes \widehat{\pi} : \widehat{p}$ indicating that the pattern $\widehat{p}$ will be applied at position $\widehat{\pi}$. Finally, struct patterns are denoted $\{\!\{\widehat{p}_0, \ldots, \widehat{p}_n\}\!\}$.

$$\widehat{p} ::= x \mid \_ \mid \widehat{p}\text{"}\mid\text{"}\widehat{p}' \qquad \text{(Pattern constructors)}$$

$$\mid \text{Word}_\ell \bowtie_{0 \le i < n} \widehat{\pi}_i : \widehat{p}_i \mid \text{Ptr}_{\ell,a}(\widehat{p}) \bowtie_{0 \le i < n} \widehat{\pi}_i : \widehat{p}_i \mid \{\!\{\widehat{p}_0, \ldots, \widehat{p}_{n-1}\}\!\} \qquad \text{(Memory constructors)}$$

6.1.1 From Source to Memory Patterns. The translation from source to memory patterns is similar to the translation from source to memory values defined in Section 5.2. It uses two mutually recursive functions **ty2pat** and **pat2mem**, which both take a source pattern and a memory type, and return a memory pattern. We only give a quick rundown of the differences and demonstrate them on an example. The full definition is in Appendix E.

Ty2Pat. Just like its value version, **ty2pat** proceeds by induction on the memory type to produce a memory pattern and calls **pat2mem** on subterm types. Unlike values, we do not need to synthesize immediates using **fillbitword**: since memory patterns closely follow the shape of memory types, with bitword specifications for words and pointers, we simply proceed recursively. For instance, $\mathbf{ty2pat}\left(\langle x, y\rangle, \text{Word}_8 \ltimes .[0, 4] : (.0 \ as \ \widehat{\tau}_x) \ltimes .[4, 8] : (.1 \ as \ \widehat{\tau}_y)\right) = \text{Word}_8 \ltimes .[0, 4] : x \ltimes .[4, 8] : y$.

Pat2Mem. Since the details of memory-specific constructs are handled by **ty2pat**, the main purpose of **pat2mem** itself is to handle pattern-specific constructs: variables, wildcards and disjunctions, which are immediately translated to their memory pattern equivalents. Other constructs are dealt with analogously to values, as shown below.

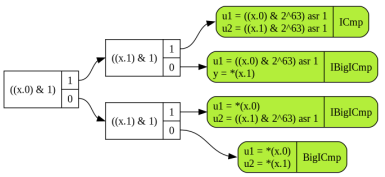
$$\mathbf{pat2mem}(\_, \widehat{\tau}) = \_ \qquad \mathbf{pat2mem}(K(p'), \widehat{\tau}) = \mathbf{ty2pat}(K(p'), \widehat{\tau}/K) \qquad \mathbf{pat2mem}(x, \widehat{\tau}) = x$$

$$\mathbf{pat2mem}(p, \widehat{\tau}) = \mathbf{ty2pat}(p, \widehat{\tau}) \qquad \mathbf{pat2mem}(p \mid q, \widehat{\tau}) = \mathbf{pat2mem}(p, \widehat{\tau}) \mid \mathbf{pat2mem}(q, \widehat{\tau})$$

Example 6.1 (Comparing Zarith Integers). As a running example for this section, we consider a comparison function on Zarith integers depicted in Fig. 7a. The matched value is of type

```
1  pub fn leq(v1:Zarith, v2:Zarith) -> Bool {
2  match (v1,v2) {
3    Small(u1), Small(u2} => ICmp(u1 <= u2),
4    Large(u1), Large(u2) => BigICmp(u1 <= u2),
5    Small(u1), Large(u2) => IBigICmp(u1 <= u2),
6    Large(u1), Small(u2) => IBigICmp(u2 >= u1),
7  }}
```

(a) Source code



(b) Output decision tree

Fig. 7. Running example for Section 6: the `leq` function on Zarith integer.

$\tau_{2z} = \langle \tau_z, \tau_z \rangle$ for which, using the type $\widehat{\tau}_z$ obtained in Example 3.3, we define the memory type $\widehat{\tau}_{2z} = \{\!\{(.0\ as\ \widehat{\tau}_z), (.1\ as\ \widehat{\tau}_z)\}\!\}$. Let us consider the source pattern $p_{2\text{small}} = \langle \text{Small}(x), \text{Small}(y) \rangle$. We have $\widehat{p}_{2\text{small}} = \textbf{pat2mem}(p_{2\text{small}}, \widehat{\tau}_{2z}) = \textbf{ty2pat}(p_{2\text{small}}, \widehat{\tau}_{2z}) = \{\!\{\widehat{p}_0, \widehat{p}_1\}\!\}$, where $\widehat{p}_0 = \textbf{pat2mem}\ (\text{Small}(x), \widehat{\tau}_z)$ and $\widehat{p}_1 = \textbf{pat2mem}\ (\text{Small}(y), \widehat{\tau}_z)$. To compute $\widehat{p}_0$, we proceed analogously to Example 5.1: $\widehat{p}_0 = \textbf{ty2pat}\ \big(\text{Small}(x), \text{Word}_{64} \ltimes [0,1] : (= 1) \ltimes [1,64] : (.\text{Small}\ as\ \text{Word}_{63})\big) = \text{Word}_{64} \ltimes [0,1] : 1 \ltimes [1,64] : x$. The computation of $\widehat{p}_1$ is similar. We thus obtain:

$$\widehat{p}_{2\text{small}} = \{\!\{\text{Word}_{64} \ltimes [0,1] : 1 \ltimes [1,64] : x, \text{Word}_{64} \ltimes [0,1] : 1 \ltimes [1,64] : y\}\!\}$$

6.2 Memory Trees

As we have seen previously, *splits* are an essential part of our formalism, allowing us to handle case disjunction gracefully, even in cases where the discriminant between branches is found in unexpected places. This also encodes a notion of *dependency*: a discriminant must be examined before the appropriate branch is taken. The most seasoned approach to pattern matching compilation [Augustsson 1985; Fessant and Maranget 2001; Maranget 1992, 2008] uses *pattern matrices* where each column encodes a disjunction, and each line encodes a conjunction. This approach is effective in general, but encodes dependencies poorly. Instead, we propose a tree-like structure, dubbed *memory trees*, to compile our patterns.

Memory trees, defined below, are our main intermediate representation during pattern matching compilation. The key idea is to preserve dependencies, yet leave the compiler free to arrange independent operations in any order. Similar to decision trees, a memory tree can be a leaf node $(\mathcal{I})$ where $\mathcal{I}$ is a set of output branch identifiers that may reach this point, or a "decision node" $\textsc{Switch}(\widehat{\pi})\ \{\dots\}$ which inspects the position $\widehat{\pi}$ in memory and picks a branch accordingly. A tree can also be a "bud" $(\mathcal{I}\ \textsc{on}\ \widehat{\tau})$: a leaf containing output branches, but that could be developed further if needed using the memory type $\widehat{\tau}$. Finally, trees can be assembled "in parallel": $\mathcal{T} \parallel \mathcal{T}'$ is a tree where one of $\mathcal{T}$ and $\mathcal{T}'$ is executed first, and the other second, the order being not yet decided. As a shortcut, $\mathcal{T}\,[f]$ substitutes over leaves and buds in a memory tree. For instance, $\mathcal{T}\,[(\mathcal{I}) \to (\mathcal{I} \cup \{i\})]$ adds a new output branch identifier to each leaf.

$$\begin{aligned}
\mathcal{T} ::= {} & (\mathcal{I}) && \text{(leaf with set } \mathcal{I} \text{ of possible output ids)} \\
\mid {} & (\mathcal{I}\ \textsc{on}\ \widehat{\tau}) && \text{(subterm bud with subtype and possible outputs ids)} \\
\mid {} & \mathcal{T}_0 \parallel \dots \parallel \mathcal{T}_{n-1} && \text{(parallel node with } n \text{ branches)} \\
\mid {} & \textsc{Switch}(\widehat{\pi})\ \{w_0 \to \mathcal{T}_0, \dots, w_{n-1} \to \mathcal{T}_{n-1}\} && \text{(switch node with } n \text{ cases)}
\end{aligned}$$

6.3 From Patterns to Memory Trees

We now consider a *matching problem* and describe how to compile it to a sequential decision tree via the use of memory trees. A matching problem is composed of a memory type $\widehat{\tau}$ under scrutiny and of a list of branches $\{\widehat{p}_i \to k_i \mid 0 \le i < n\}$, each branch consisting of a memory pattern $\widehat{p}_i$ (obtained via **pat2mem**) and a branch identifier k_i. We proceed in four steps:

(1) Scaffold (Section 6.3.1) a memory tree template from the memory type $\widehat{\tau}$.

(2) Explode (Section 6.3.2) disjunctions in the matching problem.
(3) Weave (Section 6.3.3) each pattern from the exploded matching problem into the current tree.
(4) Finalize (Section 6.3.4) the memory tree by sequentializing and optimizing it.

Example 6.2 (Example 6.1, cont'). The detailed compilation process will be illustrated on our running example. The successive memory trees are depicted graphically in Figs. 8 and 9. For instance, the memory tree of Fig. 8b has 5 parallel nodes (each having 2 children) and 2 switch nodes. For $\textsc{Switch}(\widehat{\pi})\ \{w_i \to \mathcal{T}_i, \ldots\}$, the memory path is depicted at the top of the node and w_i is the concrete value labeling the branch to subtree $\mathcal{T}_i$. Buds are depicted in yellow, leaves in green.

6.3.1 Scaffolding: Memory Type-Specific Tree Templates. Scaffolding builds a memory tree "template" based on a memory type. This memory tree does not yet contain any actual output branch, since no pattern has been taken into account yet. More precisely, $\mathbf{scaffold}_{\widehat{\pi}}(\mathcal{I}, \widehat{\tau})$ creates a memory tree based on the position $\widehat{\pi}$ (initialized as ϵ) in the type $\widehat{\tau}$, with $\mathcal{I}$ placed at the leaves. The definition is given below. Singleton types are directly turned into leaves, as they do not involve any choice. Subterms (π *as* $\widehat{\tau}$) are turned into buds ($\mathcal{I}$ $\textsc{on}$ $\widehat{\tau}$), keeping the type $\widehat{\tau}$ available for later expansion by nested patterns. Struct types, along with word and pointer types' specifications, are all treated as parallel nodes: indeed, the order in which to explore subtypes is irrelevant, and will be determined later on. Finally, splits are turned into switch nodes. Note that, unlike splits, the discriminant of a switch is absolute, hence the use of $\widehat{\pi}.\widehat{\pi}'$. Additionally, provenances are not useful at this stage anymore, and are thus not recorded in the memory tree.

$\mathbf{scaffold}_{\widehat{\pi}}(\mathcal{I})\{$

$(= w)$	$\longrightarrow (\mathcal{I})$
$(\pi\ as\ \widehat{\tau})$	$\longrightarrow (\mathcal{I}\ \textsc{on}\ \widehat{\tau})$
$\{\!\{\widehat{\tau}_0, \ldots, \widehat{\tau}_{n-1}\}\!\}$	$\longrightarrow \Big\|_{0<i\leq n} \mathbf{scaffold}_{\widehat{\pi}.i}(\mathcal{I}, \widehat{\tau}_i)$
$\mathrm{Word}_\ell \bowtie_{0\leq i<n} \widehat{\pi}_i : \widehat{\tau}_i$	$\longrightarrow \Big\|_{0<i\leq n} \mathbf{scaffold}_{\widehat{\pi}.\widehat{\pi}_i}(\mathcal{I}, \widehat{\tau}_i)$
$\mathrm{Ptr}_{\ell,a}(\widehat{\tau}) \bowtie_{0\leq i<n} \widehat{\pi}_i : \widehat{\tau}_i$	$\longrightarrow \Big\|_{0<i\leq n} \mathbf{scaffold}_{\widehat{\pi}.\widehat{\pi}_i}(\mathcal{I}, \widehat{\tau}_i) \,\|\, \mathbf{scaffold}_{\widehat{\pi}.*}(\mathcal{I}, \widehat{\tau})$
$\mathrm{Split}\,(\widehat{\pi}')\,\{w_i : \mathcal{K}_i \Rightarrow \widehat{\tau}_i \mid 0 \leq i < n\}$	$\longrightarrow \textsc{Switch}\,(\widehat{\pi}.\widehat{\pi}')\,\{w_i \to \mathbf{scaffold}_{\widehat{\pi}}(\mathcal{I}, \widehat{\tau}_i) \mid 0 \leq i < n\}$

$\}$

6.3.2 Exploding Pattern Disjunctions. Variables might be deeply nested inside a succession of disjunctions and aggregates. Keeping track of which bindings are accessible via which paths in the memory tree would be cumbersome. Instead, we simply break up all disjunctions in the initial memory matching problem to produce a larger problem whose patterns are disjunction-free. While exponential in theory, the size of patterns in actual programs make this a non-issue in practice. $\mathbf{explode}\,(\widehat{p})$ yields a list of disjunction-free patterns $\{\widehat{p}_0, \ldots, \widehat{p}_{n-1}\}$ such that $\widehat{p}_0 \mid \cdots \mid \widehat{p}_{n-1}$ is equivalent to $\widehat{p}$. Its full definition is given in Appendix E. We then define the *exploded pattern problem* as $\mathbf{explode}\,(\{\widehat{p}_i \to k_i \mid 0 \leq i < n\}) = \{\widehat{p}_{i,j} \mapsto k_i \mid 0 \leq i < n \text{ and } \widehat{p}_{i,j} \in \mathbf{explode}\,(\widehat{p}_i)\}$. For our running example, no pattern is exploded.

6.3.3 Weaving Patterns into a Memory Tree. We can now *weave* patterns from the matching problem into the previously generated memory tree. For each branch $(\widehat{p}_{\mathrm{id}} \to k_{\mathrm{id}})$ in the exploded matching problem, we define $\mathcal{T}_{\mathrm{id}+1} = \mathbf{weave}_{\epsilon,\mathrm{id}}\,(\widehat{p}_{\mathrm{id}}, \widehat{\tau}, \mathcal{T}_{\mathrm{id}})$ until exhaustion of all patterns. The initial tree $\mathcal{T}_0$ is the output of the scaffolding phase. Each weaving adds relevant choices and outputs from the pattern $\widehat{p}_{\mathrm{id}}$ to the tree. The general form $\mathbf{weave}_{\widehat{\pi},\mathrm{id}}\,(\widehat{p}, \widehat{\tau}, \mathcal{T})$ takes a memory pattern $\widehat{p}$, a memory type $\widehat{\tau}$, and a tree $\mathcal{T}$, along with the current path $\widehat{\pi}$ and an output identifier id, and returns a new memory tree. It inspects both pattern and tree, and integrates the latter into the former. We now present a selection of rules. The full definition is in Appendix E. Some key steps are illustrated on our running example at the end of this section.

Branch Identifiers and Leaves. The general goal of weaving is to add the branch identifier id to each leaf that is relevant to this pattern. By design of scaffolding, leaves always correspond to a singleton type, therefore only simple patterns (variables, wildcards, or constants) can occur there. The WeaveLeaf case simply adds the output identifier id to the leaf.

$$\mathbf{weave}_{\widehat{\pi},\mathrm{id}}\left(\widehat{p}, (= w), (\mathcal{I})\right) = (\mathcal{I} \cup \{\mathrm{id}\}) \tag{WeaveLeaf}$$

Subterms, Buds and Tree Expansion. Subterms and buds enable the memory tree to be expanded as needed to handle nested patterns. This expansion happens conditionally using the following three rules. In WeaveBudWildcard, if a wildcard reaches a bud, there is no need to expand: we simply add the output id to the bud. If there is a proper pattern, then we need to expand using **scaffold** in WeaveBud. Subterms are handled transparently via WeaveSubTerm.

$$\mathbf{weave}_{\widehat{\pi},\mathrm{id}}\left(\_, \widehat{\tau}, (\mathcal{I} \text{ on } \widehat{\tau}')\right) = (\mathcal{I} \cup \{\mathrm{id}\} \text{ on } \widehat{\tau}') \tag{WeaveBudWildcard}$$

$$\mathbf{weave}_{\widehat{\pi},\mathrm{id}}\left(\widehat{p}, \widehat{\tau}, (\mathcal{I} \text{ on } \widehat{\tau}')\right) = \mathbf{weave}_{\widehat{\pi},\mathrm{id}}\left(\widehat{p}, \widehat{\tau}, \mathbf{scaffold}_{\widehat{\pi}}(\mathcal{I}, \widehat{\tau}')\right) \tag{WeaveBud}$$

$$\mathbf{weave}_{\widehat{\pi},\mathrm{id}}\left(\widehat{p}, (\pi \ as \ \widehat{\tau}), \mathcal{T}\right) = \mathbf{weave}_{\widehat{\pi},\mathrm{id}}\left(\widehat{p}, \widehat{\tau}, \mathcal{T}\right) \tag{WeaveSubterm}$$

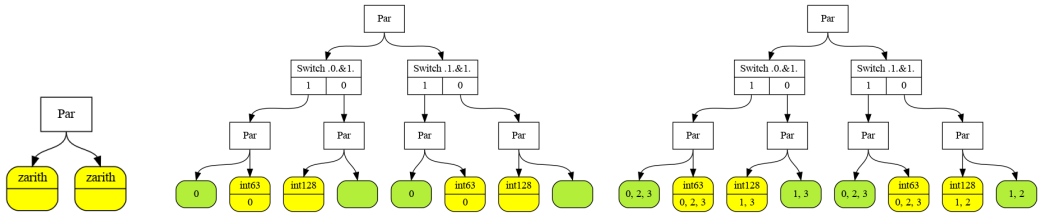
Aggregates and Parallel Nodes. The purpose of parallel nodes is to model "aggregate" constructions, which include structs, words and pointers. In all these cases, the order in which sub-patterns should be explored is not set in stone, and will be decided during sequentialization (Section 6.3.4) based on heuristics. We showcase the rules for structs below. Each rule recursively explores every subtree, using appropriate subtypes and subpatterns. The first rule WeaveStructAny handles the case where the memory pattern $\widehat{p}$ is a wildcard _ by propagating this wildcard to each subtree. The second rule WeaveStruct handles the case where $\widehat{p}$ is a struct pattern $\{\!\{\widehat{p}_0, \ldots, \widehat{p}_{n-1}\}\!\}$ by weaving each subpattern $\widehat{p}_i$ into its associated subtree. All aggregate constructions are handled with two similar rules.

$$\mathbf{weave}_{\widehat{\pi},\mathrm{id}}\left(\_, \{\!\{\widehat{\tau}_0, \ldots, \widehat{\tau}_{n-1}\}\!\}, \bigparallel_{0\le i<n} \mathcal{T}_i\right) = \bigparallel_{0\le i<n} \mathbf{weave}_{\widehat{\pi}.i,\mathrm{id}}\left(\_, \widehat{\tau}_i, \mathcal{T}_i\right) \tag{WeaveStructAny}$$

$$\mathbf{weave}_{\widehat{\pi},\mathrm{id}}\left(\{\!\{\widehat{p}_0, \ldots, \widehat{p}_{n-1}\}\!\}, \{\!\{\widehat{\tau}_0, \ldots, \widehat{\tau}_{n-1}\}\!\}, \bigparallel_{0\le i<n} \mathcal{T}_i\right) = \bigparallel_{0\le i<n} \mathbf{weave}_{\widehat{\pi}.i,\mathrm{id}}\left(\widehat{p}_i, \widehat{\tau}_i, \mathcal{T}_i\right) \tag{WeaveStruct}$$

Splits and Switches. Switches are decision nodes used to model sum-like constructs (i.e., splits). They are handled by the rule WeaveSplit shown below. The main idea is that $\widehat{p}$ should only be woven into branches of the decision tree with which it is compatible. For instance, if the pattern $\widehat{p}$ only accepts the value $\{\!\{0, 42\}\!\}$, it should not be propagated to branches that assume the value at position .0 is 1. For this purpose, we identify the *relative* path used in the switch discriminant (here, $\widehat{\pi}'$), and gather all words that could be accepted by the pattern at this position with $\widehat{\mathbf{focus}}\left(\widehat{\pi}', \widehat{p}\right)$. For each branch, if the discriminant value is accepted by the pattern at this position, we weave the pattern into the corresponding subtree.

$$\frac{\forall i,\ \mathcal{T}_i' = \begin{cases} \mathbf{weave}_{\widehat{\pi},\mathrm{id}}\left(\widehat{p}, \widehat{\tau}_i, \mathcal{T}_i\right) & \text{if } \widehat{\mathbf{focus}}\left(\widehat{\pi}', \widehat{p}\right) = w_i \text{ or } \_ \text{ or } x \\ \mathcal{T}_i & \text{otherwise} \end{cases}}{\mathbf{weave}_{\widehat{\pi},\mathrm{id}}\left(\widehat{p}, \begin{array}{l}\text{Split } (\widehat{\pi}')\ \{ \\ \quad w_0 \ : \ \mathcal{K}_0 \ \Rightarrow \widehat{\tau}_0 \\ \quad \cdot \\ \quad w_{n-1} : \mathcal{K}_{n-1} \Rightarrow \widehat{\tau}_{n-1} \\ \}\end{array}, \begin{array}{l}\textsc{Switch } (\widehat{\pi}.\widehat{\pi}')\ \{ \\ \quad w_0 \to \mathcal{T}_0 \\ \quad \cdot \\ \quad w_{m-1} \to \mathcal{T}_{m-1} \\ \}\end{array}\right) = \begin{array}{l}\textsc{Switch } (\widehat{\pi}.\widehat{\pi}')\ \{ \\ \quad w_0 \to \mathcal{T}_0' \\ \quad \cdot \\ \quad w_{m-1} \to \mathcal{T}_{m-1}' \\ \}\end{array}} \tag{WeaveSplit}$$



(a) After scaffold (b) After weaving the first pattern (c) After weaving all memory patterns

Fig. 8. First compilation steps for the running example

Example 6.3 (Running Example – Scaffolding and Weaving). We begin with $\mathbf{scaffold}_{\epsilon}(\varnothing, \widehat{\tau}_{2z})$. The resulting memory tree is depicted in Fig. 8a. The "struct" rule generates a parallel node; its two children are generated from the subtype $\widehat{\tau}_0 = (.0\ as\ \widehat{\tau}_z)$, which yields a bud of type $\widehat{\tau}_z$. We now have four memory patterns, one for each source pattern. Let us compute $\mathbf{weave}_{\epsilon,0}\left(\widehat{p}_{2\text{small}}, \widehat{\tau}_{2z}, \mathcal{T}_0\right)$, where:

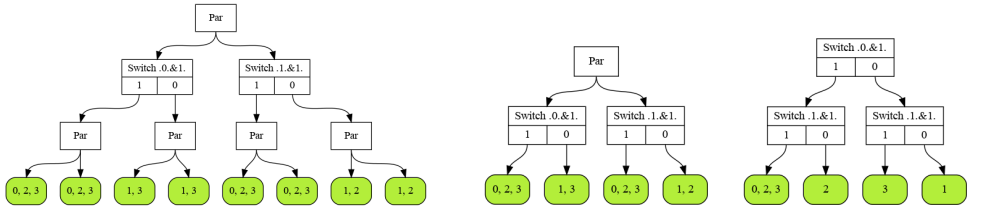
- $\widehat{p}_{2\text{small}} = \left\{\!\!\left\{\text{Word}_{64} \ltimes \left([0,1] = 1\right) \ltimes \left([1,64] = x\right), \text{Word}_{64} \ltimes \left([0,1] = 1\right) \ltimes \left([1,64] = y\right)\right\}\!\!\right\}$ is the first memory pattern of the program (see Example 6.1));
- $\widehat{\tau}_{2z} = \left\{\!\!\left\{(.0\ as\ \widehat{\tau}_z), (.1\ as\ \widehat{\tau}_z)\right\}\!\!\right\}$ is the memory type that underlies all memory patterns;
- $\mathcal{T}_0 = (\varnothing\ \textsc{on}\ \widehat{\tau}_z) \parallel (\varnothing\ \textsc{on}\ \widehat{\tau}_z)$ is the scaffolded tree.

The WeaveStruct rule explores the 'Par' node, whose two children correspond to the two fields of the struct. Let us compute its left child. Given $\widehat{p}_l = \text{Word}_{64} \ltimes [0,1] : 1 \ltimes [1,64] : x$, and after applying WeaveSubterm and WeaveBud, we have $\mathbf{weave}_{.0,0}\left(\widehat{p}_l, (.0\ as\ \widehat{\tau}_z), (\varnothing\ \textsc{on}\ \widehat{\tau}_z)\right) = \mathbf{weave}_{.0,0}\left(\widehat{p}_l, \widehat{\tau}_z, \mathbf{scaffold}_{.0}(\varnothing, \widehat{\tau}_z)\right)$. We thus first compute:

$$
\begin{aligned}
&\mathbf{scaffold}_{.0}(\varnothing, \widehat{\tau}_z) \\
&\quad = \textsc{Switch}\,(.0.[0,1])\ \left\{\begin{array}{l} 0 \to \mathbf{scaffold}_{.0}\left(\varnothing, \text{Ptr}_{64,1}((.\text{Large}\ as\ \text{Word}_{128})) \ltimes \left([0,1] : (=0)\right)\right) \\ 1 \to \mathbf{scaffold}_{.0}\left(\varnothing, \text{Word}_{64} \ltimes \left([0,1] : (=1)\right) \ltimes \left((.\text{Small}\ as\ \text{Word}_{63})\right)\right) \end{array}\right\} \\
&\quad = \textsc{Switch}\,(.0.[0,1])\ \left\{\begin{array}{l} 0 \to (\varnothing) \parallel (\varnothing\ \textsc{on}\ \text{Word}_{128}) \\ 1 \to (\varnothing) \parallel (\varnothing\ \textsc{on}\ \text{Word}_{63}) \end{array}\right\}
\end{aligned}
$$

The split in $\widehat{\tau}_z$ is mirrored by a new switch node, on which we can finally weave the pattern $\widehat{p}_l$. Following WeaveSplit, weaving only explores the branch 1 of the switch, whose discriminant value matches that at the discriminant position in the memory pattern (in our case, 1). The branch identifier 0 is then transferred to the remaining buds and leaves, yielding
$\textsc{Switch}(.0.[0,1])\ \{0 \to (\varnothing) \parallel (\varnothing\ \textsc{on}\ \text{Word}_{128})\ ;1 \to (\{0\}) \parallel (\{0\}\ \textsc{on}\ \text{Word}_{63})\}$.
Finally, this first weaving results in the memory tree depicted in Fig. 8b. After weaving each remaining memory pattern on resultant memory trees, we finally obtain Fig. 8c.

6.3.4 Post-Treatment. At this stage, we have woven all patterns into the memory tree. Our memory tree is thus "complete", in that it contains all information from the matching problem. During weaving, the shape of the tree must not be changed: indeed, it must remain synchronized with the memory type for memory patterns to be woven in the right places. Now that weaving is done, however, we can reshape the tree in arbitrary ways as long as semantics are preserved. The goal of this phase is to simplify and optimize the decision tree to prepare it for *sequential* code generation. As a first step, we "trim" the tree by removing its remaining typing information. We then sequentialize it. At any point, we can apply various classic optimizations, some of which are sketched below.



(a) After trimming (b) After pruning (c) After sequentialization

Fig. 9. Last compilation steps for the running example

Trimming. Since we have explored all patterns, the remaining buds will never be expanded, and can thus be turned into normal leaves. This is done by the following operation:
$\textbf{trim}\,(\mathcal{T}) \triangleq \mathcal{T}\,[(\mathcal{I} \text{ on } \widehat{\tau}) \mapsto (\mathcal{I})]$. From now on, we assume that no buds remain in the tree.

Sequentialization. $\textbf{seq}\,(\mathcal{T})$ is the *sequentialized* version of $\mathcal{T}$, i.e., a semantically equivalent tree that does not contain any parallel nodes. Its definition is given below and is based on the following description. When we encounter a parallel node, we first *pick* a branch i_0 (based on heuristics, for instance following Scott and Ramsey [2000]). We then *graft* the remaining branches onto each leaf of $\mathcal{T}_{i_0}$. $\textbf{graft}\,(\mathcal{T}_{\text{parent}}, \mathcal{T}_{\text{child}})$, defined below, places $\mathcal{T}_{\text{child}}$ at the leaves of $\mathcal{T}_{\text{parent}}$ and specializes the child tree's leaves by intersecting them with the initial parent leaf. This might result in empty leaves (indicating unreachable code), which can be removed later. Finally, we sequentialize the resulting tree. Note that we sequentialize *after* grafting. Sequentializing remaining branches before grafting would produce a faster compilation algorithm, but give less freedom for heuristics to pick an appropriate branch at each grafting point.

$$
\begin{array}{l}
\textbf{Seq}\{ \\
\quad (\mathcal{I}) \longrightarrow (\mathcal{I}) \\
\quad \textsc{Switch}\,(\widehat{\pi})\,\{w_i \to \mathcal{T}_i \mid 0 \le i < n\} \longrightarrow \textsc{Switch}\,(\widehat{\pi})\,\{w_i \to \textbf{seq}\,(\mathcal{T}_i) \mid 0 \le i < n\} \\
\quad \mathcal{T}_0 \parallel \ldots \parallel \mathcal{T}_{n-1} \longrightarrow \textbf{seq}\,(\textbf{graft}\,(\mathcal{T}_{i_0}, \mathcal{T}_{\text{rest}})) \text{ where } \begin{array}{l} i_0 \leftarrow \textbf{pick}(\mathcal{T}_0, \ldots, \mathcal{T}_{n-1}) \\ \mathcal{T}_{\text{rest}} \leftarrow \|_{\substack{0 \le i < n \\ i \neq i_0}} \mathcal{T}_i \end{array} \\
\}
\end{array}
$$

$$\textbf{graft}\,(\mathcal{T}_{\text{parent}}, \mathcal{T}_{\text{child}}) \triangleq \mathcal{T}_{\text{parent}}\,[(\mathcal{I}) \mapsto \mathcal{T}_{\text{child}}\,[(\mathcal{I}') \mapsto (\mathcal{I} \cap \mathcal{I}')]]$$

Tree Optimizations. Optimizations on decision trees from literature can be used at any point after trimming. Sequentialization, in particular, might introduce redundant tests or create unreachable branches. Two optimizations are particularly relevant. **Constant folding** propagates information from switches, such as "position $\widehat{\pi}$ contains value w", and uses it to remove redundant switches. **Dead branch elimination** removes branches that lead to leaves with no outputs (i.e., $(\varnothing)$).

Example 6.4 (Running Example – Trimming and Sequentializing). Fig. 9a is immediately obtained from Fig. 8c by trimming type information from buds. After this step, some redundancy remains, thus we apply an easy simplification to obtain Fig. 9b. We are now ready to remove parallel nodes. Let us pick the left branch (as $\mathcal{T}_0$; $\mathcal{T}_1$ is then the other switch node) and apply $\textbf{graft}\,(\mathcal{T}_0, \mathcal{T}_1)$. $\mathcal{T}_0$ is thus promoted as parent of $\mathcal{T}_1$. All leaves of $\mathcal{T}_0$ are replaced by (a copy of) $\mathcal{T}_1$, in which leaves' sets are intersected. For instance, the right child $\{1, 3\}$ of $\mathcal{T}_0$ is replaced by a Switch(.1&1) whose leaves are $\{1, 3\} \cap \{0, 2, 3\} = \{3\}$ and $\{1, 3\} \cap \{1, 2\} = \{1\}$, hence Fig. 9c.

6.4 Bindings and Extraction

So far, we have built a decision tree that only *recognizes* patterns and orients towards the right match case. However, it does not bind variables present in the matched pattern. We now show how

to build a binding environment adapted to the memory representation. For simplicity, we give a declarative specification here.

We first gather all bindings of an *exploded* memory matching problem m as a set B of quadruplets $(x, \mathrm{id}, k, \widehat{\pi})$ containing a variable, a branch identifier, the next program and a memory path. Concretely, $\mathbf{bindings}(m) \triangleq \left\{(x, \mathrm{id}, k_{\mathrm{id}}, \widehat{\pi}) \;\middle|\; \widehat{\mathbf{focus}}\,(\widehat{\pi}, \widehat{p}) = x \text{ and } (\widehat{p}_{\mathrm{id}} \to k_{\mathrm{id}}) \in m\right\}$.

We then replace each leaf with its associated binding environment and output program. We only keep the smallest branch identifier, since the first match case has priority.

$$\mathbf{BindVars}\,(B, \mathcal{T}) \triangleq \mathcal{T}\,[(\mathcal{I}) \mapsto (\widehat{\sigma}, k)] \text{ where } \mathrm{id}_0 = \min(\mathcal{I}) \text{ and } \widehat{\sigma} = \{(x \mapsto \widehat{\pi}) \mid (x, \mathrm{id}_0, k, \widehat{\pi}) \in \Sigma\}$$

Example 6.5 (Running Example – Bindings). Recall our first pattern $\langle \mathrm{Small}(x), \mathrm{Small}(y)\rangle$ and its memory counterpart: $\{\!\{\mathrm{Word}_{64} \ltimes [0,1] : 1 \ltimes [1,64] : x, \mathrm{Word}_{64} \ltimes [0,1] : 1 \ltimes [1,64] : y\}\!\}$. We must bind x and y to the integer values in the first and second fields of the source tuple. We get the following binding quadruplets: $\{(x, 0, 0, .0.[1,64]), (y, 0, 0, .1.[1,64])\}$, which yield the memory environment $\widehat{\sigma}_0 = \{x \mapsto .0.[1,64]; y \mapsto .1.[1,64]\}$, which we insert in each leaf whose leading identifier is 0, yielding the final decision tree depicted in Fig. 7b. Each variable is then bound to the location of the memory representation of its associated source subterm in the root memory value.

6.5 Soundness

We now state the soundness of pattern matching compilation, along with a sketch of the proof. Details are in Appendix F.

Typing and Evaluation Judgments. To state correctness, we require source typing and evaluation judgements. Given a source matching problem $m = p_0 \to k_i \mid \ldots \mid p_{n-1} \to k_{n-1}$, we also define $m \rhd v \to k, \sigma$ for "m matches value v in the k^{th} branch and binds the variables in σ", where σ is a map from variables x to source paths π. We also write $\Gamma \vdash p : t \to \Sigma$ for "pattern p has type t and binds variables whose types are defined in environment Σ". Both are fully defined in Appendix C.

Analogously, we write $\widehat{m} \blacktriangleright \widehat{v} \to k, \widehat{\sigma}$ for "$\widehat{m}$ matches the memory value $\widehat{v}$ in the k^{th} branch and binds the variables in $\widehat{\sigma}$", where $\widehat{\sigma}$ is a map from variables x to memory paths $\widehat{\pi}$. Note that there is no notion of *memory typing* for memory patterns and values. We also extend agreement to type environments: given Σ and $\widehat{\Sigma}$, we write $\widehat{\Sigma} \Vdash \Sigma$ if $\mathrm{dom}(\widehat{\Sigma}) = \mathrm{dom}(\Sigma)$ and for all $x \in \mathrm{dom}(\Sigma)$, we have $\widehat{\Sigma}(x) \Vdash \Sigma(x)$. Finally, let $(\widehat{\sigma}, k) = \mathcal{T} \blacktriangleright\!\blacktriangleright \widehat{v}$ denote the evaluation of memory tree $\mathcal{T}$ on input $\widehat{v}$. and $\widehat{\Gamma} \vdash \mathcal{T} : \widehat{\tau}$ denote "tree $\mathcal{T}$ inspects memory values of type $\widehat{\tau}$". They are fully defined in Appendix D.

Theorem 6.6. *Let τ, Γ (resp. $\widehat{\tau}, \widehat{\Gamma}$) a source (resp. memory) type and its environment such that $\widehat{\tau} \Vdash \tau$. Let v be a source value and $m = \left\{\widehat{p}_i \to i \;\middle|\; 0 \le i < n\right\}$ a source matching problem such that $\Gamma \vdash v : \tau$ and $\forall i, \Gamma \vdash p_i : \tau \to \Gamma_i$. For each pattern type environment Σ_i, let $\widehat{\Sigma}_i$ such that $\widehat{\Sigma}_i \Vdash \Sigma_i$.*

We define the memory value $\widehat{v} = \boldsymbol{val2mem}(v, \widehat{\tau})$ and the memory matching problem $\widehat{m} = \{\boldsymbol{pat2mem}(p_i, \widehat{\tau}) \to i \mid 0 \le i < n\}$, along with its exploded version $\widehat{m}' = \boldsymbol{explode}\,(\widehat{m}) = \left\{\widehat{p}_j \to k_j \;\middle|\; 0 \le j < n'\right\}$ and its bindings $B = \boldsymbol{bindings}(\widehat{m}')$. Finally, we define the following trees, following our compilation algorithm:

$$\mathcal{T}_{-1} = \boldsymbol{scaffold}_{\epsilon}(\varnothing, \widehat{\tau}) \quad \mathcal{T}_j = \boldsymbol{weave}_{\epsilon,k_j}\left(\widehat{p}_j, \widehat{\tau}, \mathcal{T}_{j-1}\right) \quad \mathcal{T} = \boldsymbol{BindVars}\,(B, \boldsymbol{seq}\,(\boldsymbol{trim}\,(\mathcal{T}_{n'-1})))$$

The source matching problem and its compiled version are equivalent: for all $i \in [0, n-1]$ and $\sigma : \Sigma_i$, we have

$$m \rhd v \to i, \sigma \iff \mathcal{T} \blacktriangleright\!\blacktriangleright \widehat{v} \to i, \widehat{\sigma} \wedge \begin{array}{l} \forall x \in \widehat{\sigma},\ \text{let } \widehat{\tau}_x \in \widehat{\Sigma}_i(x), \pi = \sigma(x) \text{ and } \widehat{\pi} = \widehat{\sigma}(x) \\ \quad \text{then } \widehat{\boldsymbol{focus}}\,(\widehat{\pi}, \widehat{v}) = \boldsymbol{val2mem}(\boldsymbol{focus}\,(\pi, v), \widehat{\tau}_x) \end{array}$$

where $\mathcal{T} \blacktriangleright\!\blacktriangleright \widehat{v}$ denotes the execution of $\mathcal{T}$ with $\widehat{v}$ as input (this judgment is fully defined in Appendix F).

Proof Sketch. We use the *memory-level* pattern matching judgment $\blacktriangleright$ as a bridge between the "constructive" (from source to memory) and "destructive" (from memory objects to trees) parts of our compilation scheme. We first handle the source-to-memory part by proving the equivalence below. We use the agreement criteria from Definition 4.3 to ensure that each component of τ is reflected in a component of $\widehat{\tau}$, then proceed by induction on **val2mem** and **pat2mem**, whose definition closely follows the memory type.

$$p \rhd v \to \sigma \iff \mathbf{pat2mem}(p, \widehat{\tau}) \blacktriangleright \widehat{v} \to \widehat{\sigma} \wedge \begin{array}{r} \forall (x : \widehat{\tau}_x) \in \widehat{\Sigma},\ \text{let } \pi = \sigma(x) \text{ and } \widehat{\pi} = \widehat{\sigma}(x) \\ \text{then } \widehat{\mathbf{focus}}\,(\widehat{\pi}, \widehat{v}) = \mathbf{val2mem}(\mathbf{focus}\,(\pi, v), \widehat{\tau}_x) \end{array}$$

For the memory-to-tree part, we restore typing information discarded by our compilation scheme, using a separate tree typing judgment $\widehat{\Gamma} \vdash \mathcal{T} : \widehat{\tau}$ which ensures that the structure of $\mathcal{T}$ reflects that of the considered memory type $\widehat{\tau}$. Let us first focus on pattern recognition equivalence, putting aside variable bindings.

We first show that scaffolding yields a well-typed tree whose evaluation on a memory value yields the initial output set. In our case, we get $\widehat{\Gamma} \vdash \mathcal{T}_{-1} : \widehat{\tau}$ and $\mathcal{T}_{-1} \blacktriangleright\!\blacktriangleright \widehat{v} \to \varnothing$.

We prove weaving correctness by a "progress and preservation" approach, showing that weaving preserves the "shape" of the tree, encapsulated by tree typing. We use the following invariants: let $\mathcal{T}$ a tree such that $\widehat{\Gamma} \vdash \mathcal{T} : \widehat{\tau}$ and $\mathcal{T} \blacktriangleright\!\blacktriangleright \widehat{v} \to \mathcal{I}$ for some set of identifiers $\mathcal{I}$. Let $(\widehat{p} \to \mathrm{id}) \in m$ an exploded memory pattern. $\mathbf{weave}_{\epsilon,\mathrm{id}}\left(\widehat{p}, \widehat{\tau}, \mathcal{T}\right)$ is defined and yields a memory tree $\mathcal{T}'$ such that $\widehat{\Gamma} \vdash \mathcal{T}' : \widehat{\tau}$ and $\mathcal{T} \blacktriangleright\!\blacktriangleright \widehat{v} \to \mathcal{I} \cup (\{\mathrm{id}\} \text{ if } \widehat{p}_{\mathrm{id}} \blacktriangleright \widehat{v} \text{ and } \varnothing \text{ otherwise})$.

For all subsequent operations, we only need to show that the result of tree evaluation is preserved at each step. For each step, this can be done by a direct induction. We conclude the following for our fully woven tree: $\mathcal{T}_{n'-1} \blacktriangleright\!\blacktriangleright \widehat{v} \to \left\{\mathrm{id} \;\middle|\; (\widehat{p} \to \mathrm{id}) \in m \text{ and } \widehat{p} \blacktriangleright \widehat{v}\right\}$.

At this stage, we have shown that we find the same code identifier in the source and compiled version. We now need to deal with bindings. Since we have already proven that bindings are coherent between source- and memory-level pattern matching, we only have to show that memory bindings are preserved by **Bindings** and **BindVars**, which is immediate from their definitions. □

7 EXTENSIONS

So far and for simplicity, we have described our formalism for a language with "plain" Algebraic Data Types (only sums, products, and references), and comparatively simple split types, with only head constructors in provenances (which, in effect, limits inlining) and only one split discriminant. We have extended our formalism in several ways, including constants in patterns and multiple split discriminants, which are described in Appendix B. We now focus on two extensions we consider essential for the expressivity of our approach: irregular specifications, and arrays. Irregular specifications, along with several other extensions, are included in the formalism detailed in appendices and implemented in ribbit.

7.1 Irregular Specifications

So far, to ease our presentation, we have restricted specifications to memory types in which split branches' provenances solely consist of head constructors. This limits the extent to which one may provide different specifications for different sum type variants, and prevents inlining of deeply nested fields.

Example 7.1 (Arithmetic Expressions). Consider the type of arithmetic expressions on 32-bit integers with variables. Let $\tau_{\mathrm{exp}} = V(\mathrm{string}) + I(\mathrm{Int}_{32}) + E(\mathrm{op}, \tau_{\mathrm{exp}}, \tau_{\mathrm{exp}})$ and $\mathrm{op} = \mathrm{Plus} + \mathrm{Minus} + \ldots$. For our application, we need a representation that supports high sharing of values. Given what we have seen so far, we can immediately choose a bit-stealing representation to distinguish between

the I, V and E cases. The I case is represented immediately with the 32-bit integer nested, the V is a tagged pointer to a string (which can easily be shared), and the E case is a pointer to a structure containing the operator $\widehat{t}_{\text{op}}$ (represented by an 8-bit integer) and the two sub-expressions. To make manipulating pointers to expressions cheap, we avoid tagging the E case.

$$\widehat{t}_{\text{op}} \mapsto \text{Word}_8 \ltimes \text{Split}\ (.)\ \{\ldots\}$$

$$\widehat{t}_{\text{exp}} \mapsto \text{Split}\ ([0,2]) \begin{Bmatrix} 1 : I \Rightarrow \text{Word}_{64} \ltimes ([32,64] : (.I\ \textit{as}\ \text{Word}_{32})) \ltimes ([0,2] : (=1)) \\ 2 : V \Rightarrow \text{Ptr}_{64,2}\left(\widehat{\text{string}}\right) \ltimes ([0,2] : (=2)) \\ 0 : E \Rightarrow \text{Ptr}_{64,2}\left(\widehat{t}_{\text{node}}\right) \ltimes ([0,2] : (=0)) \end{Bmatrix}$$

$$\widehat{t}_{\text{node}} \mapsto \left\{\!\!\left\{\text{Word}_{56}; (.E.0\ \textit{as}\ \widehat{t}_{\text{op}}); (.E.1\ \textit{as}\ \widehat{t}_{\text{exp}}); (.E.2\ \textit{as}\ \widehat{t}_{\text{exp}})\right\}\!\!\right\}$$

While this representation is reasonable, we notice during benchmarking that many expressions have small branches, such as $(3 \times x) + 5$. We now want to leverage the Word_{56} padding in $\widehat{t}_{\text{node}}$ to store an Int_{32} when one of the two subexpressions is directly an integer. This is difficult: we need to make a special representation for values whose shape is $E(\langle \_, I(\_), \_\rangle$ or $E(\langle \_, \_, I(\_)\rangle$. So far, provenances in split branches can only be head constructors of a type. To this end, we extend our setup so that provenances in splits can describe deeper constructors in types. We then use the split construct to provide a specialized representation for values of the aforementioned shapes. We obtain the following modified $\widehat{t}_{\text{node}}$:

$$\left\{\!\!\left\{ (E.0\ \textit{as}\ \widehat{t}_{\text{op}}),\ \begin{array}{l} \text{Split}\ (.0)\ \{ \\ \quad 0 : E(\langle \top, I(\top), \top\rangle) \Rightarrow \left\{\!\!\left\{\text{Word}_{24} = 0; (.E.1.I\ \textit{as}\ \text{Word}_{32}); (.E.2\ \textit{as}\ \widehat{t}_{\text{exp}})\right\}\!\!\right\} \\ \quad 1 : E(\langle \top, \top, I(\top)\rangle) \Rightarrow \left\{\!\!\left\{\text{Word}_{24} = 1; (.E.2.I\ \textit{as}\ \text{Word}_{32}); (.E.1\ \textit{as}\ \widehat{t}_{\text{exp}})\right\}\!\!\right\} \\ \quad 2 : \quad E(\langle \top, \top, \top\rangle) \quad \Rightarrow \left\{\!\!\left\{\text{Word}_{24} = 1; \text{Word}_{32}; (.E.1\ \textit{as}\ \widehat{t}_{exp}); (.E.2\ \textit{as}\ \widehat{t}_{\text{exp}})\right\}\!\!\right\} \\ \} \end{array} \right\}\!\!\right\}$$

This makes constructors and destructors more complex. For instance, let us take the expression $E(\text{Plus}, x, y)$. We need to determine whether x and y carry a V tag, and act accordingly. A "high-level" version of such code would look as follows: $\texttt{match}\ (x, y)\ \{V(\_), \_ \to \ldots \mid \_, V(\_) \to \ldots \mid \_, \_ \to \ldots\}$. Naturally, we want to emit the corresponding low-level code.

We now lift the regularity restriction to allow for aggressive inlining, and review the changes to our memory specification formalism needed to accommodate for such richer representations. Full-fledged *provenances*, denoted prov and defined below, are analogous to simplified source patterns that give the full tree of constructors, with variants, records, references, and wildcard $\top$. The split construct now features a set of full provenances: $\text{Split}\ (\widehat{\pi})\ \left\{w_i : \text{provs}_i \Rightarrow \widehat{\tau}_i \mid 0 \le i < n\right\}$. Finally, $\text{prov} \Cap \text{prov}'$ is the intersection of provenances (which might return $\bot$).

$$\text{prov} ::= K \mid \top \mid \&\text{prov} \mid \langle \text{prov}_0, \ldots, \text{prov}_{n-1}\rangle \mid K(\text{prov})$$

Specialization. Because provenances now have an arbitrary depth, the specialization operation defined in Section 4.1 may return multiple possible types. Indeed, consider the types above: the tag E matches several branches, therefore the source pattern $E(\_)$ must accommodate memory values that follow either of these branches. To this end, specialization now takes a full provenance and returns a *set* of possible memory types. For provenances without $\top$s, it always returns a single memory type.

Validity and Agreement. We adjust memory type validity to ensure that improved provenances from different branches never overlap. In particular, given a split type $\text{Split}\ (\widehat{\pi})\ \left\{\overline{w_i : \text{provs}_i \Rightarrow \widehat{\tau}_i}\right\}$, we enforce that all pairs $\text{prov} \in \text{provs}_i, \text{prov}' \in \text{provs}_j$ for $i \neq j$ are non-overlapping. In addition to memory type validity, we must also refine the agreement criteria presented in Definition 4.3.

Subterm coherence, coverage and distinguishability criteria only receive minimal changes, following the richer specialization operation. Provenance coherence, on the other hand, is more complex. Indeed, it must now allow *inlining*: a memory type might include a subterm's nested constructors and distinguish between them early on, enabling it to rearrange arbitrarily nested elements as desired, without requiring the whole subterm to be represented as a standalone memory value. To this end, we define *source type specialization*, which filters nested constructors of a source type according to a provenance. For instance, $\tau_{\text{exp}}/E(\top, V, \top) = \{E(\text{op}, V(\text{string}), \tau_{\text{exp}})\}$. The full definition is available in Appendix C. Since source type specialization may yield multiple types, we must now define agreement criteria between a memory type and a *set* of source types. These updated agreement criteria are as follows:

All subterms bind source subtypes to their valid representation. (Subterm Coherence)
Either $\widehat{\tau}$ and all $\tau \in T$ are leaves (i.e., $T = \{\text{Int}_\ell\}$ and $\widehat{\tau} = \text{Word}_\ell$) or for all $\widehat{\pi}$ such that $\widehat{\textbf{focus}}\,(\widehat{\pi}, \widehat{\tau}) = (\pi\ as\ \widehat{\tau}_\pi)$, we have $\widehat{\tau}_\pi \Vdash \{\textbf{focus}\,(\pi, \tau) \mid \tau \in T/\pi\}$.

Split provenances and branches are coherent with the source type. (Provenance Coherence)
For all $\widehat{\pi}$ s.t. $\widehat{\textbf{focus}}\,(\widehat{\pi}, \widehat{\tau}) = \text{Split}\,(\widehat{\pi}')\,\{w_i : \text{provs}_i \Rightarrow \widehat{\tau}_i\}$, for each i and each $\text{prov} \in \text{provs}_i$, prov is a valid provenance of at least one $\tau \in T$ and $\widehat{\tau}/\text{prov} \Vdash \{\tau/\text{prov} \mid \tau \in T \wedge \text{prov is valid for } \tau\}$.

All source subtypes are represented within the memory type. (Coverage)
For every π that leads to a leaf in at least one τ (i.e., $\exists \tau \in T, \textbf{focus}\,(\pi, \tau) = \text{Int}_\ell$), $\widehat{\tau}$ *covers* π: every memory type $\widehat{\tau}' \in \widehat{\tau}/\pi$ contains a subterm type for a source position π_0 prefix of π. More precisely, there exist a source and memory path $\pi_0, \widehat{\pi}_0$ such that $\widehat{\textbf{focus}}\,(\widehat{\pi}_0, \widehat{\tau}') = (\pi_0\ as\ \widehat{\tau}_0)$ and $\pi_0 \preceq \pi$.

Memory types provide a way to tell branches of sum types apart. (Distinguishability)
For all source types $\tau_0, \tau_1 \in T$, for all $\text{prov}_0, \text{prov}_1$ such that prov_0 is a valid provenance for τ_0, prov_1 is a valid provenance for τ_1 and $\text{prov}_0 \Cap \text{prov}_1 = \bot$ (i.e., prov_0 and prov_1 are incompatible), $\widehat{\tau}$ *distinguishes* prov_0 and prov_1. More precisely, for all $\widehat{\tau}_0 \in \widehat{\tau}/\text{prov}_0$ and $\widehat{\tau}_1 \in \widehat{\tau}/\text{prov}_1$, there exists a memory path $\widehat{\pi}$ such that $\widehat{\textbf{focus}}\,(\widehat{\pi}, \widehat{\tau}_0) = (= w_0)$, $\widehat{\textbf{focus}}\,(\widehat{\pi}, \widehat{\tau}_1) = (= w_1)$ and $w_0 \neq w_1$.

Using richer provenances opens the way to unrolling recursive types, for instance, representing a list two-by-two. Unfortunately, this might force constructors/destructors to walk the structure recursively. It is unclear how to behave reasonably in this case. For now, we forbid such unrolling altogether.

Val2Mem and Pat2Mem. Both **val2mem** and **pat2mem** rely on every variable corresponding to a subterm type, allowing for the representation of a subvalue to be simply copied, or passed by reference. This is not the case anymore. The solution here is, when presented with a variable at a position whose representation has been mangled, to deconstruct and reconstruct locally the value, until we reach the next available subterm type, allowing us to keep "standalone" and "nested" representations synchronized. Naturally, this requires more runtime computation (a traditional tradeoff of very compact representation). While reasonable on paper, this is quite delicate in practice, and even more so in implementation. RIBBIT only implements simple cases.

7.2 Arrays

Our presentation focused so far on usual algebraic data types. However, a central part of representation choices is interaction between data types and arrays: whether to unbox elements inside an array, the size and alignment of these elements, or the choice between "struct of arrays" or "arrays of struct" representations. All these choices are essential to improve locality and performance or unlock the use of specific instructions such as SIMD. We now give a quick sketch of how RIBBIT could be extended to support arrays. The key idea is to embed iteration variables in memory types and paths to "synchronize" accesses. We illustrate this on an example.

Example 7.2 (Struct of Arrays). Consider $\tau_{\text{tup}} = \langle \text{Int}_{32}, \&\tau, \text{Int}_8 \rangle$ from Example 3.1. We want to represent $\tau_a = [\tau_{\text{tup}}]$, an array of such tuples. We choose a struct-of-array representation:

$$\widehat{\tau_a} = \{\!\{\text{Ptr}_{64}([\![(.[i].0 \ as \ \text{Word}_{32})]\!]_i), \text{Ptr}_{64}([\![(.[i].1.* \ as \ \text{Word}_{64})]\!]_i), \text{Ptr}_{64}([\![(.[i].2 \ as \ \text{Word}_8)]\!]_i)\}\!\}$$

$\widehat{\tau_a}$ is a struct containing three pointers to memory arrays (denoted $[\![\ldots]\!]_i$ for arrays of arbitrary size and indexed by i). Each array contains subterms pointing to a part of the tuple in the source array, for instance $.[i].0$. The path $.[i]$ indicates the i^{th} array element. Additionally, we choose here to unbox the contents of the second array to avoid an indirection.

In this context, rebuilding the memory value corresponding to an array access $x[y]$ requires work similar to irregular representations shown before. For instance, for a of type τ_a, the expression $e = a[i]$ requires building a memory value corresponding to the representation $\widehat{\tau}_{\text{tup}}$ described in Example 3.1, i.e., a structure containing appropriate memory paths and padding:

$$\widehat{e} = \{\!\{\widehat{x}.0.*.[i],\ \widehat{x}.2.*.[i],\ \bar{0}^{24},\ \text{Ptr}_{64}(\widehat{x}.1.*.[i])\}\!\}$$

8 CODE GENERATION

In the previous sections, we have described compilation steps to lower constructors (*i.e*, value allocation) and destructors (*i.e*, pattern matching) to a low-level description operating on memory values. This section deals with the last step of our compilation scheme: producing executable LLVM IR code to build memory values, and to deconstruct such memory values using decision trees. The global code generation procedure is rather straightforward: constructors are turned into memory allocation and initialization code and decision trees are turned into LLVM control flow graphs. Two difficulties remain: deal with LLVM's explicitely typed IR and generate code for memory paths.

8.1 Lowering Memory Types

Our compilation procedure discards type information, yielding a decision tree whose input is a single untyped root memory value. Backend targets such as LLVM IR require each memory value (including the root value, bound subterm values and intermediate switch discriminants) to be explicitely typed. This section describes the process of building an adequate LLVM IR type for each memory value, using information retrieved from the initial memory type.

Since every memory value in the decision tree is expressed as a position within the root memory value, we can deduce their types from the root memory type. However, our memory types contain *splits*, which are not expressable as LLVM types [LLVM Language Reference Manual 2023]. Indeed, LLVM types are unambigous, in that they only describe fully concrete types (in terms of bit width, legal operations, etc.) and do not capture multiple branches. For our purposes, they consist of (1) fixed-width integer types, such as `i32` (2) an opaque pointer type `ptr` (3) structures that aggregate other LLVM types, such as `{i64, ptr}`. For simplicity, we assume LLVM pointer types are by default 64-bit wide with 8 unused bits due to address alignment. Different pointer types requires using LLVM *address spaces* with different data layout specifications (in the LLVM sense).

We now describe, given a type $\widehat{\tau}$, how to build the corresponding LLVM type. This is straightforward for non-split memory types: we map any ℓ-bit-wide word or pointer to the integer type `i`ℓ, discarding any bitword content specification. We do not use the specific `ptr` type yet so as to freely manipulate pointer alignment bits. We map structs to LLVM structs and type variables and subterm types to their bound memory type's LLVM type.

Lowering split types to LLVM IR types is less immediate. We need an LLVM type that is able to store values of any branch, and in which the split discriminant is accessible. By definition, the exact shape of a memory value instanciated from a split type (and an unknown source value) is unknown until we inspect its discriminant. Type validity ensures this is possible: indeed, a split type has one

kind and each branch type must be of this kind. Furthermore, the discriminant location is always accessible in each branch type. Conservatively, we use the "largest" branch type to determine the common type shape. If the kind is **Word**, every branch maps to an LLVM integer type and we take the largest. If the kind is **Block**, every branch maps to an LLVM struct and we recursively find a common type for each field, and keep extra fields. After inspecting the discriminant, we refine the memory type and cast the value to a more precise LLVM type in order to perform operations specific to the identified variant. By validity of memory types, this cast should always be valid.

8.2 Code Generation for Memory Paths

Memory paths $\widehat{\pi}$ are used to specify the discriminant of each switch node and to specify values in binding environments. Code generation transforms memory paths into sequences of instructions extracting the part of the root memory value specified by the path. Memory path operations consist of pointer dereferencing, field access and arbitrary operations on bitwords. Given a target providing instructions for dereferencing, field access and all operations used in bitword content specifications, as well as casts from pointers to words and back, mapping operations on structs and on words to target instructions is immediate. Dereferencing operations require additional care to reset all alignment bits and cast the value to a pointer type before dereferencing it.

Example 8.1. The path $.*$ on $\mathrm{Ptr}_{64,8}(\dots)$ becomes

```
%x1 = and i64 %x0, 0xfffffffffffff00
%x2 = inttoptr i64 %x1 to ptr
%x3 = load ty, ptr %x2
```

9 EXPERIMENTAL EVALUATION

We implemented our DSL, full compilation procedure and final LLVM IR generation in a tool dubbed RIBBIT, which is publicly available. The tool supports ADT declarations with recursive types and user-defined memory layouts, including our extension to irregular layouts (Section 7.1), constant patterns and multi-discriminant splits (Appendix B). Our code generation outputs fully shared decision trees (following Maranget [2008]). Finally, on top of the numerous examples described in the rest of this article, we implemented "full representations", i.e., functions from source types to memory types mimicking existing languages, such as OCaml. In this section we demonstrate:

- that RIBBIT is expressive enough to reproduce the behavior of native OCaml and Rust compilers, on some representative middle-sized examples, with similar performance;
- that acting on low-level memory layouts impacts static characteristics of generated decision trees, as well as execution-time performance.

The latest version of the artifact associated with this article, which includes RIBBIT along with instructions to reproduce these results, is available at https://doi.org/10.5281/zenodo.7994178.

For this purpose, we consider two examples: the red-black trees motivating example from Section 2, and a stack machine interpreter example used by Maranget [2008] to showcase various heuristics for OCaml pattern-matching compilation. The full programs are available in Appendix G. For these two examples, we have implemented native OCaml and Rust versions, two RIBBIT versions mimicking the internal memory representations of OCaml and Rust, along with the Linux-like red-black tree encoding from Section 2.3. Details of internal memory layout were obtained both from non-official sources [Minsky and Madhavapeddy 2021; The Rustonomicon 2023] and by manual inspection of emitted intermediate representations. However, we do not attempt to match compilation heuristics. Experiments have been performed on a desktop machine running Gentoo GNU/Linux x86-64 on an Intel Core i5 CPU @ 1.60GHz ×8 (although RIBBIT is single-threaded), with 32 Gio of RAM. As for compiler versions, we used Rust 1.67.1, OCaml 4.14.0 and LLVM 14.

compiler & layout		stack interpreter exec. time	stack interpreter memsize	stack interpreter #pointers	red-black trees exec. time	red-black trees memsize	red-black trees #pointers
ocamlopt		4.15	12	4	3.13	155	31
rustc		3.31	7	2	3.37	124	30
ribbit	OCaml	1.89	13	4	5.45	156	31
	Rust	1.87	8	3	4.61	125	31
	Linux	–	–	–	4.54	94	31

Fig. 10. Execution times on 10^9 runs, and memory usage (exec. time is in ns, memsize in words)

memory layout		stack interpreter OCaml	stack interpreter Rust	red-black trees OCaml	red-black trees Rust	red-black trees Linux
code size (switches)		38	30	26	21	21
path length (switches)	avg	5.25	4.55	9.25	8.73	8.72
	min	2	2	1	1	1
	max	8	6	13	13	13
deref ops	avg	2.56	2.55	4.45	4.19	2.36
	min	1	1	1	1	0
	max	4	3	7	7	3

Fig. 11. Static metrics of decision trees generated by RIBBIT : code size is the total number of switches; we also compute the number of switches per path in the decision tree; and the number of dereferences along these paths.

In Fig. 10 we first compare the memory size and number of pointers for some concrete values (a small stack program and a red-black tree with 30 nodes), to cross-validate our memory layout specifications (the size of objects in OCaml/Rust versus the size obtained in RIBBIT with our encoding). As shown, the data match for all implementations. We then compare execution-time performance of the generated code. This comparison should be made while keeping in mind that RIBBIT emits code that does far less than native OCaml or Rust: our prototype implementation does not implement sophisticated memory management, or even proper function calls. Additionally, the measured times are very tiny, making measurement difficult. Our goal here is only to show that our technique can emit code of similar efficiency to seasoned industry-ready compilers using their memory layouts, which is indeed the case. The final lesson from these dynamic measurements is that improving the representation of values is very worthwhile: each reduction from the OCaml representation to the Rust and then to the Linux layout significantly reduces memory footprint and execution time. The Linux red-black trees implementation, in particular, is extremely efficient while having a tiny footprint, demonstrating the gain of such sophisticated bit-stealing.

Figure 11 depicts static metrics obtained via RIBBIT for different memory encodings. For both benchmarks, the better performance obtained with the Rust layout and to a further extent with the Linux layout seems to correlate with fewer derefencing operations. Perhaps unsurprisingly, it seems that a good static measurement to predict performance is the amount of indirection. These results suggest that we should complement existing heuristics for pattern matching compilation with new ones taking data layout-related metrics into account, such as number of dereferencing operations and cache friendliness. We plan to explore this in the future.

10 RELATED WORKS

10.1 Memory Representation and Algebraic Data Types

Memory representation in functional polymorphic garbage-collected languages was quickly identified as an important area for performance improvement (Jones and Launchbury [1991]; Leroy [1992]; Peterson [1989]). Our work encourages new development in this area, as it readily supports such layouts and allows to easily experiment with new representations. Unboxing and arrays have been the subject of numerous works and libraries (see [Keller et al. 2010] for a recent Haskell

example). We believe many of the data layouts proposed in these works would enhance our approach, notably regarding mutability and concurrency, which we do not explore. Colin et al. [2018] refine the criterion for recursive yet unboxed types in the OCaml case. We believe a similar refined criterion could be used in the irregular case. Iannetta et al. [2021]; Koparkar et al. [2021] propose completely flattened representations for recursive types, providing excellent cache behavior and parallelism but requiring whole-program transformations. In contrast, our technique provides great manual control over memory representation and follows a more traditional compilation pipeline. Supporting such fully flattened layouts would be highly desirable.

Several approaches attempt to combine polymorphism and optimized data layouts. Leroy [1990] shows how to make polymorphic and monomorphic representations work conjointly and Hall et al. [1994] show how to marry specialization and unboxing. Classically, C++ and Rust rely aggressively on specialization. All these approaches would be compatible with our work.

Rust uses the notion of "niche" to exploit unused bits in words and pointers. They have been steadily adding new layout optimizations that exploit manual annotations, up to exploiting pointer alignment like we do [RFC: Alignment niches for references types 2021]. We believe our memory types offer a more formal definition, which allows to reason about these data layout optimizations, and hope to collaborate on the topic with Rust developers. We also believe the use of explicit memory types and agreement criteria offer a better user interface for such manual optimizations.

10.2 Pattern Languages

Pattern languages have been adopted in mainstream languages with rich static typing such as Haskell, OCaml, F#, Scala or Rust, but also more recently in more general languages such as Python and soon Java. This diversity of host languages, with their very varied compilation techniques and memory representations, make our work all the more relevant. We focused on the core of pattern matching language, with some minor extensions such as disjunctive patterns. Extensions such as ranges, guards, or polymorphic variants [Garrigue 1998] are orthogonal to our work.

Pattern matching is also used pervasively in dependently typed languages [Cockx et al. 2016; Tuerk et al. 2015] and for GADTs [Garrigue and Normand 2015; Karachalias et al. 2015]. In such setting, matching on a term can reveal the type of another term. Memory trees precisely handle such dependencies very well, and we hope to adapt them to such typing disciplines in the future.

Active patterns [Syme et al. 2007] and pattern synonyms [Pickering et al. 2016] allow users to abstract over patterns by exposing "constructors" which do not directly reflect the underlying definition of the algebraic data type. This allows for both a "programmer" view and a "representation" view, similar to our approach. Combined with rich type algebra with unboxing annotations, it enables some representation tricks. However, we are so far not aware of any typing algebra that leverages niches and bit-stealing, making such a combination far more limited than our approach.

10.3 Pattern Matching Optimization

Pattern matching optimization in strict languages has been a rich topic of study. Closest to us, Fessant and Maranget [2001] and later Maranget [2008] champion the "row and column" approach using a pattern matrix, which is currently used in OCaml. As we mentioned before, this approach deals with dependencies between various parts of a given pattern poorly. Sestoft [1996] emits a rough tree of `if` nodes, and relies on a global supercompilation pass to optimize the resulting tree. This could be an interesting alternative to our parallel nodes and sequentialization. Kosarev et al. [2020] explore a different optimization technique by encoding the choice of optimal decision tree into a relational synthesis problem, and solving through miniKanren. Their idea is very promising, but fails to scale to big matches. [Solodkyy et al. 2013] propose patterns-as-library for C++ based

on objects and template meta-programming. In all these cases, DAGs with maximal sharing can be created from trees using hash-consing [Filliâtre and Conchon 2006].

Most approaches rely on heuristics for choosing a column to split. Maranget [2008] introduces "necessity"-based heuristics. A study of heuristics is done in [Scott and Ramsey 2000]. We believe our approach, which exposes much lower-level details than commonly considered in pattern matching algorithms, could benefit from new heuristics taking data layout into account.

11 CONCLUSION

We have presented RIBBIT[4], a language approach to describe bit-aware memory representations of ADTs based on a dual-view approach. From source and memory specifications, RIBBIT compiles constructors and destructors to low-level code. We demonstrate the expressivity and versatility of our approach by encoding classic memory layouts for ADTs (OCaml, Rust) along with numerous memory optimizations such as bit-stealing, unboxing, and inlining. As we provide end-users with an expressive language, we propose a notion of validity and agreement between source and memory types. Together with correctness proofs, these notions strongly increase the trust of the whole process (hence the main title of this article).

Our technique paves the way towards the formalization and description of optimization techniques for memory representation. Indeed, for the moment, end-users write their own memory layout specification. We plan to investigate generating new optimized memory layouts, for instance applying super-optimization or automatically searching for representations optimized for space and cache behavior. RIBBIT can serve as a substantial building block for these future works.

Naturally, to achieve this objective, we also want to extend our approach. Future versions of RIBBIT featuring arbitrary pointer layouts and even lower-level, architecture-dependent details (e.g., endianness) would enable memory layouts specifically tuned for various platforms. Mutability and concurrency in algebraic types have received significant attention from the Rust community, which adapt well to our work. We also hope to capture far more tweaked representations for recursive types, including unrolling and flattening, notably following the work from Koparkar et al. [2021].

A EXAMPLE: WEBKIT-LIKE NAN-BOXING

In this section, we describe a representation that models the memory layout used in the JavaScriptCore engine (built into WebKit) to encode JavaScript values on 64-bit platforms, which uses an optimization dubbed *NaN-boxing*. Our description is based off the implementation available in [WebKit NaN-boxing 2023].

We use the following extensions to our memory description framework:

- irregular specifications with nested provenances, described in Section 7.1;
- "default" branches in splits that capture values which do not match any other branch, which we denote with a wildcard discriminant value _. For instance, Split $(\widehat{\pi})$ $\begin{Bmatrix} w : \text{provs}_1 \Rightarrow \widehat{\tau_1} \\ \_ : \text{provs}_2 \Rightarrow \widehat{\tau_2} \end{Bmatrix}$ describes a memory type that maps values whose provenance is in provs_1 to $\widehat{\tau_1}$, which always contains w at the position $\widehat{\pi}$, and values whose provenance is in provs_2 to $\widehat{\tau_2}$, which never contains w at the position $\widehat{\pi}$;
- pattern matching on integer and floating-point numeric constants, described in Appendix B.1;
- simple arithmetic operations (in our case, addition) on values of constant types: $(\pi + n \ \mathit{as}\ \text{Word}_\ell)$ denotes a memory type that maps a source value v to an ℓ-bit word encoding the numeric value **focus** $(\pi, v) + n$.

We first define a source type *jsval* to model JavaScript values, which consist of [ecm 2023]:

[4] https://gitlab.inria.fr/ribbit/ribbit

- four constants: *undefined*, *null* and the boolean values *true* and *false*;
- *numbers*, which consist of 32-bit integers and double-precision floating-point numbers;
- arbitrary-precision integers (BigInt), character strings, symbols and objects, which we collectively refer to as *cells*. Every cell value is represented behind a pointer; here, we ignore other representation details.

Hence the following source type:

$$\begin{aligned} \mathit{jsval} = {} & \text{Undefined} + \text{Null} + \text{Boolean}(\text{True} + \text{False}) \\ & + \text{Number}(\text{Double}(\text{Float}_{64}) + \text{Integer}(\text{Int}_{32})) \\ & + \text{Cell}(\mathit{cell}) \end{aligned}$$

We now define a memory type $\widehat{\mathit{jsval}}$ that represents *jsval* values according to the layout used in JavaScriptCore. This memory layout is based on the double-precision binary encoding defined by the IEEE 754 standard and takes advantage of unused *NaN* values to represent all JavaScript values as 64-bit words.

More precisely, the IEEE 754 double-precision binary format consists of one sign bit (most significant bit, numbered 63), 11 exponent bits (numbered 52–62) and 52 significand bits (numbered 0–51). NaN values are defined as values whose exponent bits are all set, with quiet (as opposed to signaling) NaN values flagged by setting the most significant bit of the significand (i.e., bit 51). The sign bit is irrelevant.

The space of 64-bit words whose top 13 bits are set (i.e., quiet NaNs with the sign bit set) is therefore available to encode non-double values in the 51 remaining *payload* bits (excluding the zero payload, reserved for NaNs originating from hardware or C library functions). Conversely, valid double-precision encodings are necessarily within the range from 0 inclusive to `0xfff8000000000000` exclusive.

The JavaScriptCore implementation also takes advantage of the fact that no current x86-64 implementation uses more than 2^{48} bytes of virtual address space, that is, 48 bits are sufficient to store any machine pointer. In order to keep pointer dereferencing unencumbered by extra decoding operations, pointers are assigned the range from 0 inclusive to 2^{48} exclusive. The four constants (Undefined, Null, True and False) are mapped to values in this range, using the fixed invalid pointer values `0xa`, `0x2`, `0x7` and `0x6` respectively. Assuming that $\widehat{\mathit{cell}}$ properly represents the type *cell*, the following memory type describes the layout of non-numeric values (i.e., cells and constants):

$$\widehat{\tau}_{\text{other}} = \text{Split}\ (.)\ \begin{cases} \texttt{0x6} : \text{Boolean(False)} \Rightarrow (= \texttt{0x6}) \\ \texttt{0x7} : \text{Boolean(True)} \Rightarrow (= \texttt{0x7}) \\ \texttt{0xa} : \quad \text{Undefined} \quad \Rightarrow (= \texttt{0xa}) \\ \texttt{0x2} : \quad \text{Null} \quad \Rightarrow (= \texttt{0x2}) \\ \_ : \quad \text{Cell} \quad \Rightarrow \text{Ptr}_{48}\left((.\text{Cell}\ \mathit{as}\ \widehat{\mathit{cell}})\right) \end{cases}$$

The range of double-precision numbers is, in turn, offset by 2^{49}, making its exclusive upper bound `0xfffa000000000000` and yielding the following memory type:

$$\widehat{\tau}_{\text{double}} = (.\text{Number.Double} + 2^{49}\ \mathit{as}\ \text{Word}_{64})$$

Finally, 32-bit integer values are assigned the range from `0xfffe000000000000` to `0xfffe0000ffffffff` inclusive, which lies outside both pointer and double-precision ranges (i.e., we use the standard 32-bit integer encoding on the 32 lowest bits and the fixed constant `0xfffe0000` on the 32 highest bits). We define the corresponding memory type

$$\widehat{\tau}_{\text{int}} = \text{Word}_{48} \ltimes [0, 32] : (.\text{Number.Integer}\ \mathit{as}\ \text{Word}_{32}) \ltimes [32, 48] : (= \texttt{0x0000})$$

We finally combine the three memory types previously defined for double, integer, and non-numeric values into a single memory type $\widehat{jsval}$ which is valid and agrees with *jsval*, using the top 16 bits as tag bits (i.e., split discriminant):

$$
\begin{array}{l}
\widehat{jsval} = \text{Split}\ (.[48, 64])\ \{ \\
\quad \texttt{0x0000} : \{\text{Undefined, Null, Boolean, Cell}\} \Rightarrow \text{Word}_{64} \ltimes [0, 48] : \widehat{\tau}_{\text{other}} \ltimes [48, 64] : (= \texttt{0x0000}) \\
\quad \texttt{0xfffe} : \quad \text{Number(Integer)} \quad \Rightarrow \text{Word}_{64} \ltimes [0, 48] : \widehat{\tau}_{\text{int}} \ltimes [48, 64] : (= \texttt{0xfffe}) \\
\quad \_ \quad : \quad \text{Number(Double)} \quad \Rightarrow \text{Word}_{64} \ltimes \epsilon : \widehat{\tau}_{\text{double}} \\
\}
\end{array}
$$

B EXTENSIONS

B.1 Pattern Matching on Constants

We now want to allow pattern matching on constants. Let us consider C the set of constant types. We assume that constant types only have a single subterm whose path is ϵ. As an example, we have $\text{Int}_\ell \in C$. Pattern matching on constant requires a few additions:

- Unlike for sum types, a switch on constants can not cover "the full signature" which contains all the cases. Indeed, some constants types have infinitely many values, such as strings. Even finite ones such as 64bits integers are far too large to enumerate.
- The "interesting" branches are not known during scaffolding, and will need to be added during weaving of each patterns.

For this reason, we first extend memory trees with switches on words with a default case, in Fig. 12b. On first inspection of the memory type, when we encounter a subterm $(\pi\ as\ \widehat{\tau})$ with $\widehat{\tau}$ a constant type, **scaffold** simply emits such switch with an initial default case containing a leaf, as shown in Fig. 12c. Initially, there is only a default case as no other constant patterns has been presented. **weave** then modifies the appropriate branches, as shown in Fig. 12d. There are three cases depending on the woven pattern $\widehat{p}$. If $\widehat{p}$ is a constant pattern w for a branch that is already present in the switch, we simply extend that branch. If $\widehat{p}$ is a wildcard, we propagate in all the branches. Finally, if $\widehat{p}$ is a constant pattern w that is *not* present in the existing branches, we add a new branch initialized as the default case, and propagate in that new branch. In all cases, each branch can only contain a simple leaf, which we (potentially) extend with the new b.

In this presentation, we assume that the constant type is represented by a word. Some constants are more complex, for instance strings. In that case, we could introduce a dedicated switch construct that would be handled specifically. This would be essential for strings, for instance, as switches on strings already benefits from numerous optimisations in compilers.

B.2 Multiple Split Discriminants

Our split types may be made more flexible by the addition of *multiple* discriminant locations. So far, our memory types only allow for a single memory path to indicate the piece of data in memory that distinguishes between different branches of a split. However, one may wish to use a *combination of multiple values* at distinct positions in memory instead.

For instance, consider the following source type:

$$\tau = \text{Blob}(\langle \text{Int}_{63}, \text{Int}_{63} \rangle) + \text{Left}(\langle \text{Int}_{128}, \text{Int}_{63} \rangle) + \text{Right}(\langle \text{Int}_{63}, \text{Int}_{128} \rangle)$$

Each constructor contains two integers, each of which may be small (63 bits) or large (128 bits). Only three combinations are supported: small-small, large-small and small-large, represented by the three constructors.

$$p ::= \cdots \mid c \qquad \text{(Constants)} \qquad\qquad \widehat{p} ::= \cdots \mid w \in \mathbb{W} \qquad \text{(Constant Words)}$$

(a) Grammar for constant patterns and constant memory patterns

$$\mathcal{T} ::= \cdots \mid \textsc{Switch}\,(\widehat{\pi}) \begin{Bmatrix} w_0 \to \mathcal{T}_0 \\ \vdots \\ w_{n-1} \to \mathcal{T}_{n-1} \\ \_ \to \mathcal{T}_* \end{Bmatrix}$$

(b) Grammar for Switch with default cases

$$\begin{array}{l} \textbf{scaffold}_{\widehat{\pi}}(B)\{ \\ \quad \cdots \qquad\qquad \longrightarrow \cdots \\ \quad (\pi\ as\ \widehat{\tau} \in C) \longrightarrow \textsc{Switch}(\widehat{\pi})\ \{\_ \to (B)\} \\ \} \end{array}$$

(c) Scaffolding for constant types

$$\textsc{WeaveAtom}\ \frac{\begin{cases} \text{if } \exists j, \widehat{p} = w_j & B'_j = B_j \cup \{b\} \text{ and } \forall i \neq j,\ B'_i = B_i & B'_* = B_* \\ \text{if } \widehat{p} = \_ & \forall i,\ B'_i = B_i \cup \{b\} & B'_* = B_* \cup \{b\} \\ \text{if } \widehat{p} = w \notin \{w_i\} & \forall i,\ B'_i = B_i \text{ and } B_n = B_* \cup \{b\} & B'_* = B_* \end{cases}}{\textbf{weave}_{\widehat{\pi},b}\left(\widehat{p}, (\pi\ as\ \widehat{\tau}), \textsc{Switch}(\widehat{\pi})\left\{\overline{w_i \to (B_i)}^{\,n}, \_ \to (B_*)\right\}\right) = \textsc{Switch}(\widehat{\pi})\left\{\overline{w_i \to (B'_i)}, \_ \to (B'_*)\right\}}$$

(d) Weaving for constant patterns

Fig. 12. Grammars and semantics for constant patterns

We define the following memory type to represent τ values:

$$\begin{array}{l} \widehat{\tau} = \text{Split}\ (.0.[0,1], .1.[0,1])\ \{ \\ \quad 1,1 :\ \{\text{Blob}\} \Rightarrow \left\{\!\!\left\{ \begin{array}{c} \text{Word}_{64} \ltimes ([0,1] : (=1)) \ltimes ([1,64] : (.\text{Blob.0}\ as\ \text{Word}_{63})), \\ \text{Word}_{64} \ltimes ([0,1] : (=1)) \ltimes ([1,64] : (.\text{Blob.1}\ as\ \text{Word}_{63})) \end{array} \right\}\!\!\right\} \\ \quad 1,0 :\ \{\text{Left }\} \Rightarrow \left\{\!\!\left\{ \begin{array}{c} \text{Ptr}_{64,1}((.\text{Left.0}\ as\ \text{Word}_{128})) \ltimes ([0,1] : (=0)), \\ \text{Word}_{64} \ltimes ([0,1] : (=1)) \ltimes ([1,64] : (.\text{Left.1}\ as\ \text{Word}_{63})) \end{array} \right\}\!\!\right\} \\ \quad 0,1 : \{\text{Right}\} \Rightarrow \left\{\!\!\left\{ \begin{array}{c} \text{Word}_{64} \ltimes ([0,1] : (=1)) \ltimes ([1,64] : (.\text{Right.0}\ as\ \text{Word}_{63})), \\ \text{Ptr}_{64,1}((.\text{Right.1}\ as\ \text{Word}_{128})) \ltimes ([0,1] : (=0)) \end{array} \right\}\!\!\right\} \\ \} \end{array}$$

In all three cases, the two integers are stored in two struct fields and represented on 64 bits, with the lowest bit serving as a tag to distinguish small integers from pointers to large integers. We therefore need to inspect the lowest bit in *both* fields to determine the head constructor. We thus express $\widehat{\tau}$ as a split type with two discriminants that correspond to these two memory locations. Each branch, in turn, has two discriminant values in its left-hand side (one per location).

This memory type is more precise than a simple tuple-like representation, in which each field of the struct would have its own split to indicate the variant of the corresponding source field. Indeed, the multi-discriminant split clearly expresses that memory values in which both lowest bits are set to 0 are *invalid*, in that they never represent any source value. This allows us to avoid emitting code for such invalid combinations during compilation.

This extension requires minor alterations to memory types' grammar and validity:

$$\widehat{\tau} ::= \ldots \mid \text{Split}\,(\widehat{\pi}_0, \ldots, \widehat{\pi}_{n-1}) \left\{ \begin{array}{ccc} w_{0,0}, \ldots, w_{n-1,0} & : & \text{provs}_0 \Rightarrow \widehat{\tau}_0 \\ \vdots, \ddots, \vdots & : & \vdots \Rightarrow \vdots \\ w_{0,m-1}, \ldots, w_{n-1,m-1} & : & \text{provs}_{m-1} \Rightarrow \widehat{\tau}_{m-1} \end{array} \right\}$$

SplitValidity

$$\frac{\widehat{\Gamma} \vDash \widehat{\tau}_j : \widehat{\kappa} \qquad \forall j \neq j', \forall \text{prov} \in \text{provs}_j, \forall \text{prov}' \in \text{provs}_{j'}, \text{prov} \Cap \text{prov}' = \bot \qquad \widehat{\textbf{focus}}\,(\widehat{\pi}_i, \widehat{\tau}_j) = (= w_{i,j})}{\widehat{\Gamma} \vDash \text{Split}\left(\overline{\widehat{\pi}_i}\right) \left\{\overline{w_{i,j} : \text{provs}_j \Rightarrow \widehat{\tau}_j}\right\} : \widehat{\kappa}}$$

Finally, in order to compile a multi-discriminant split type, **scaffold** must emit a switch node for each discriminant and combine them all in a parallel node:

$$\begin{aligned} &\textbf{scaffold}_{\widehat{\pi}}\left(\mathcal{I}, \text{Split}\,(\widehat{\pi}_0, \ldots, \widehat{\pi}_{n-1}) \left\{ \begin{array}{ccc} w_{0,0}, \ldots, w_{n-1,0} & : & \text{provs}_0 \Rightarrow \widehat{\tau}_0 \\ \vdots \ddots \vdots & : & \vdots \Rightarrow \vdots \\ w_{0,m-1}, \ldots, w_{n-1,m-1} & : & \text{provs}_{m-1} \Rightarrow \widehat{\tau}_{m-1} \end{array} \right\}\right) \\ = \; &\|_{0 \leq i < n} \textsc{Switch}(\widehat{\pi}.\widehat{\pi}_i) \left\{ w_{i,j} \to \textbf{scaffold}_{\widehat{\pi}}\left(\mathcal{I}, \widehat{\tau}_j\right) \mid 0 \leq j < m \right\} \end{aligned}$$

B.3 Lists with Irregular Specification

Example B.1 (List). Consider the source type $\tau_{\text{list}} = \text{Nil} + \text{Cons}(\langle \text{Int}_{32}, \text{list} \rangle)$ and the type environment $\Gamma = \{\text{list} \mapsto \tau_{\text{list}}\}$. The specification scheme described so far is not expressive enough to inline several list items: we may only define different representations for Nil and Cons variants, and cannot "look ahead" inside Cons' second field to match against nested items.

Consider the following memory type within the memory type environment $\widehat{\Gamma} = \left\{\widehat{\text{list}} \mapsto \widehat{\tau}_{\text{list}}\right\}$:

$$\widehat{\tau}_{\text{list}} = \begin{array}{l} \text{Split}\,([0,0])\,\{ \\ \quad 0 : \text{Cons} \Rightarrow \text{Ptr}_{64,1}\left(\left\{\!\!\left\{(.\text{Cons}.0 \; \textit{as} \; \text{Word}_{32}), \text{Word}_{32}(.\text{Cons}.1 \; \textit{as} \; \widehat{\text{list}})\right\}\!\!\right\}\right) \ltimes [0,0] : (=0) \\ \quad 1 : \;\; \text{Nil} \;\; \Rightarrow \text{Word}_{64} \ltimes [0,0] : (=1) \\ \} \end{array}$$

This representation of lists uses the same pointer-tagging scheme than previously presented. Since the content of the list are 32-bit integers, we introduce padding inside the Cons case.

We now would like to pack two elements of a list together in a single 64-bit word: we thus need to specify how to represent values of the form $\text{Cons}(\langle i_1, \text{Cons}(\langle i_2, \ell \rangle)\rangle)$ differently. Unfortunately, the split construct only accept provenances with a single *head* constructor, without regards for nested constructors. To specify this packed list representation, we need more expressive provenances. Using the richer provenances from Section 7.1, we obtain:

$$\widehat{\tau}'_{\text{list}} = \begin{array}{l} \text{Split}\,([0,1])\,\{ \\ \quad 0 : \qquad \text{Nil} \qquad\quad \Rightarrow \text{Word}_{64} \ltimes . : (=1) \\ \quad 1 : \quad \text{Cons}(\langle \top, \text{Nil} \rangle) \quad \Rightarrow \text{Word}_{64} \ltimes ([32,64] : (.\text{Cons}.0 \; \textit{as} \; \text{Word}_{32})) \ltimes ([0,32] : (=1)) \\ \quad 2 : \text{Cons}(\langle \top, \text{Cons}(\top) \rangle) \Rightarrow \text{Ptr}_{64,1}\left(\left\{\!\!\left\{ \begin{array}{c} (.\text{Cons}.0 \; \textit{as} \; \text{Word}_{32}), \\ (.\text{Cons}.1.\text{Cons}.0 \; \textit{as} \; \text{Word}_{32}), \\ (.\text{Cons}.1.\text{Cons}.1 \; \textit{as} \; \widehat{\text{list}}'), \end{array} \right\}\!\!\right\}\right) \ltimes [0,1] : (=2) \\ \} \end{array}$$

The Nil case in unchanged. In the first case (a list with a single element), the highest bits store the 32-bit integer and the lowest bits the tag. Note that this type would be invalid with the simple

provenance Cons: the subterm Cons.1 would be missing. We require the improved provenance Cons.1.Nil. The last case (a list with at least two elements) is represented by a pointer to a cell containing both elements, and the appropriate subterms. The requirements that we cover all branches and can distinguish between branches wouldn't hold with "simple" provenances.

This change of representation has numerous consequences, both semantically (the behavior of sharing, notably), and codegen-wise (how to extract values during pattern matching?) and should be evaluated carefully. Crucially, this change of representation leave the source code unchanged, allowing for easier experiments.

C SOURCE LANGUAGE

This appendix is available in the full version of this article at https://inria.hal.science/hal-04165615.

D MEMORY LANGUAGE

This appendix is available in the full version of this article at https://inria.hal.science/hal-04165615.

E FULL COMPILATION RULES

This appendix is available in the full version of this article at https://inria.hal.science/hal-04165615.

F FULL SOUNDNESS PROOF

This appendix is available in the full version of this article at https://inria.hal.science/hal-04165615.

G EXPERIMENTS ADDITIONAL INFORMATION

This appendix is available in the full version of this article at https://inria.hal.science/hal-04165615.

The material contained in this appendix is also part of the artifact available at https://doi.org/10.5281/zenodo.7994178.

REFERENCES

2023. ECMAScript Language Types. "https://262.ecma-international.org/14.0/#sec-ecmascript-language-types". [Online; accessed 17-July-2023].

AoS and SoA 2023. AoS and SoA. https://en.wikipedia.org/w/index.php?title=AoS_and_SoA&oldid=1068565041. [Online; accessed 22-February-2023].

Lennart Augustsson. 1985. Compiling pattern matching. In *Functional Programming Languages and Computer Architecture*, Jean-Pierre Jouannaud (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 368–381.

Rod M Burstall. 1977. Design considerations for a functional programming language. *The software revolution* (1977).

Rod M. Burstall, David B. MacQueen, and Donald Sannella. 1980. HOPE: An Experimental Applicative Language. In *Proceedings of the 1980 LISP Conference, Stanford, California, USA, August 25-27, 1980*. ACM, 136–143. https://doi.org/10.1145/800087.802799

Zilin Chen, Ambroise Lafont, Liam O'Connor, Gabriele Keller, Craig McLaughlin, Vincent Jackson, and Christine Rizkallah. 2023. Dargent: A Silver Bullet for Verified Data Layout Refinement. *Proc. ACM Program. Lang.* 7, POPL, Article 47 (jan 2023), 27 pages. https://doi.org/10.1145/3571240

Jesper Cockx, Dominique Devriese, and Frank Piessens. 2016. Eliminating dependent pattern matching without K. *J. Funct. Program.* 26 (2016), e16. https://doi.org/10.1017/S0956796816000174

Simon Colin, Rodolphe Lepigre, and Gabriel Scherer. 2018. Unboxing Mutually Recursive Type Definitions in OCaml. *arXiv preprint arXiv:1811.02300* (2018).

Fabrice Le Fessant and Luc Maranget. 2001. Optimizing Pattern Matching. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001*, Benjamin C. Pierce (Ed.). ACM, 26–37. https://doi.org/10.1145/507635.507641

Jean-Christophe Filliâtre and Sylvain Conchon. 2006. Type-safe modular hash-consing. In *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*, Andrew Kennedy and François Pottier (Eds.). ACM, 12–19. https://doi.org/10.1145/1159876.1159880

Jacques Garrigue. 1998. Programming with polymorphic variants. In *ML Workshop*, Vol. 13. Baltimore.

Jacques Garrigue and Jacques Le Normand. 2015. GADTs and Exhaustiveness: Looking for the Impossible. In *Proceedings ML Family / OCaml Users and Developers workshops, ML Family/OCaml 2015, Vancouver, Canada, 3rd & 4th September 2015 (EPTCS, Vol. 241)*, Jeremy Yallop and Damien Doligez (Eds.). 23–35. https://doi.org/10.4204/EPTCS.241.2

Cordelia V. Hall, Simon L. Peyton Jones, and Patrick M. Sansom. 1994. Unboxing using Specialisation. In *Proceedings of the 1994 Glasgow Workshop on Functional Programming, Ayr, Scotland, UK, September 12-14, 1994 (Workshops in Computing)*, Kevin Hammond, David N. Turner, and Patrick M. Sansom (Eds.). Springer, 96–110. https://doi.org/10.1007/978-1-4471-3573-9_7

Paul Iannetta, Laure Gonnord, and Gabriel Radanne. 2021. Compiling pattern matching to in-place modifications. In *GPCE '21: Concepts and Experiences, Chicago, IL, USA, October 17 - 18, 2021*, Eli Tilevich and Coen De Roover (Eds.). ACM, 123–129. https://doi.org/10.1145/3486609.3487204

Simon L. Peyton Jones and John Launchbury. 1991. Unboxed Values as First Class Citizens in a Non-Strict Functional Language. In *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings (Lecture Notes in Computer Science, Vol. 523)*, John Hughes (Ed.). Springer, 636–666. https://doi.org/10.1007/3540543961_30

Georgios Karachalias, Tom Schrijvers, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2015. GADTs meet their match: pattern-matching warnings that account for GADTs, guards, and laziness. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, Kathleen Fisher and John H. Reppy (Eds.). ACM, 424–436. https://doi.org/10.1145/2784731.2784748

Gabriele Keller, Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon L. Peyton Jones, and Ben Lippmeier. 2010. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 261–272. https://doi.org/10.1145/1863543.1863582

Chaitanya Koparkar, Mike Rainey, Michael Vollmer, Milind Kulkarni, and Ryan R. Newton. 2021. Efficient Tree-Traversals: Reconciling Parallelism and Dense Data Representations. *Proc. ACM Program. Lang.* 5, ICFP, Article 91 (2021), 29 pages. https://doi.org/10.1145/3473596

Dmitry Kosarev, Petr Lozov, and Dmitry Boulytchev. 2020. Relational Synthesis for Pattern Matching. In *Processings of the 18th Programming Languages and Systems Asian Symposium, APLAS 2020, Fukuoka, Japan, 2020 (Lecture Notes in Computer Science, Vol. 12470)*, Bruno C. d. S. Oliveira (Ed.). Springer, 293–310. https://doi.org/10.1007/978-3-030-64437-6_15

Xavier Leroy. 1990. Efficient Data Representation in Polymorphic Languages. In *Programming Language Implementation and Logic Programming, 2nd International Workshop PLILP'90, Linköping, Sweden, August 20-22, 1990, Proceedings (Lecture Notes in Computer Science, Vol. 456)*, Pierre Deransart and Jan Maluszynski (Eds.). Springer, 255–276. https://doi.org/10.1007/BFb0024189

Xavier Leroy. 1992. Unboxed Objects and Polymorphic Typing. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, Ravi Sethi (Ed.). ACM Press, 177–188. https://doi.org/10.1145/143165.143205

Xavier Leroy and Antoine Miné. 2010. *Zarith*. https://archive.softwareheritage.org/browse/directory/56fa3bba77ee6c8a40b4230eba576600641a4db1/?origin_url=https://github.com/ocaml/Zarith&revision=e67bccbe69362866d2ce182bb4ba8a1a458cd387&snapshot=3f57262eeea9a011da7b93770318fc6411728b6a

LLVM Language Reference Manual 2023. LLVM Language Reference Manual. "http://web.archive.org/web/20230208202541/https://www.llvm.org/docs/LangRef.html#type-system".

Luc Maranget. 1992. Compiling Lazy Pattern Matching. In *Proceedings of the Conference on Lisp and Functional Programming, LFP 1992, San Francisco, California, USA, 22-24 June 1992*, Jon L. White (Ed.). ACM, 21–31. https://doi.org/10.1145/141471.141499

Luc Maranget. 2007. Warnings for pattern matching. *J. Funct. Program.* 17, 3 (2007), 387–421. https://doi.org/10.1017/S0956796807006223

Luc Maranget. 2008. Compiling pattern matching to good decision trees. In *Proceedings of the ACM Workshop on ML, 2008, Victoria, BC, Canada, September 21, 2008*, Eijiro Sumii (Ed.). ACM, 35–46. https://doi.org/10.1145/1411304.1411311

Yaron Minsky and Anil Madhavapeddy. 2021. *Real World OCaml*. Chapter Memory Representation of Values. https://dev.realworldocaml.org/runtime-memory-layout.html

John Peterson. 1989. Untagged Data in Tagged Environments: Choosing Optimal Representations at Compile Time. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, Joseph E. Stoy (Ed.). ACM, 89–99. https://doi.org/10.1145/99370.99377

Matthew Pickering, Gergo Érdi, Simon Peyton Jones, and Richard A. Eisenberg. 2016. Pattern synonyms. In *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*, Geoffrey Mainland (Ed.). ACM, 80–91. https://doi.org/10.1145/2976002.2976013

RFC: Alignment niches for references types 2021. RFC: Alignment niches for references types. https://github.com/rust-lang/rfcs/pull/3204.

AMR Sabry and Matthias Felleisen. 1993. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation* 6 (1993), 289–360. https://doi.org/10.1007/BF01019462

Kevin Scott and Norman Ramsey. 2000. When do match-compilation heuristics matter. *University of Virginia, Charlottesville, VA* (2000).

Peter Sestoft. 1996. ML pattern match compilation and partial evaluation. In *Partial Evaluation*. Springer, 446–464.

Yuriy Solodkyy, Gabriel Dos Reis, and Bjarne Stroustrup. 2013. Open pattern matching for C++. In *Generative Programming: Concepts and Experiences, GPCE'13, Indianapolis, IN, USA - October 27 - 28, 2013*, Jaakko Järvi and Christian Kästner (Eds.). ACM, 33–42. https://doi.org/10.1145/2517208.2517222

Don Syme, Gregory Neverov, and James Margetson. 2007. Extensible pattern matching via a lightweight language extension. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, Ralf Hinze and Norman Ramsey (Eds.). ACM, 29–40. https://doi.org/10.1145/1291151.1291159

The Rust Reference 2023. The Rust Reference. "https://web.archive.org/web/20221225222340/https://doc.rust-lang.org/reference/type-layout.html#the-default-representation".

The Rustonomicon 2023. The Rustonomicon. "https://doc.rust-lang.org/nomicon/data.html".

Linus Torvalds. 2023. The Linux Kernel. https://archive.softwareheritage.org/browse/content/sha1_git:45b6ecde3665aa744f790cd915445fe07595181c/?origin_url=https://github.com/torvalds/linux&path=include/linux/rbtree_types.h&release=v6.2&snapshot=de81d8ff32247a7edaa935cf0468bf16237d25c5

Thomas Tuerk, Magnus O. Myreen, and Ramana Kumar. 2015. Pattern Matches in HOL: - A New Representation and Improved Code Generation. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9236)*, Christian Urban and Xingyuan Zhang (Eds.). Springer, 453–468. https://doi.org/10.1007/978-3-319-22102-1_30

Philip Wadler. 1987. Efficient compilation of pattern-matching. *The implementation of functional programming languages* (1987).

WebKit NaN-boxing 2023. WebKit NaN-boxing. https://archive.softwareheritage.org/browse/content/sha1_git:1eec01b377e2c5e9f6a1b9ba27bdc322919256dd/?origin_url=https://github.com/WebKit/WebKit&path=Source/JavaScriptCore/runtime/JSCJSValue.h&revision=e7780d735de3975ec8bf854e4ed70d07c7648497&snapshot=df63564a304482d8c3aa219d0c9a1ce47dee6573#L402

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