Compiling Morphisms of Algebraic Data Types

Thaïs Baudon, Gabriel Radanne, Laure Gonnord

Compiling Morphisms of Algebraic Data Types

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

Now integrated in mainstream languages, Algebraic Data Types (ADTs) have established themselves as a nice way to reason about data structures and their manipulation using *pattern matching*. However, their use in low-level programming remains limited despite efforts, notably from the Rust community.Recently, Baudon et al. [2023] propose to let programmers express the precise memory layout of a given Algebraic Data Type, while still enjoying high-level programming constructs. Their compilation procedure covers efficient pattern matching, but leaves out constructors and struggles with arbitrarily mangled memory layouts.

So far, the literature on ADT compilation rarely mentions constructors, which are indeed a non-issue on simple memory layouts. However, when data pieces are broken and scattered in memory, this task becomes particularly challenging. Even simple accessors might require constructing new values. This is the case for many low-level representations such as network packets, instruction sets, database data-structures, or aggressively packed representations.

In this article, we go one step further, by enabling optimized compilation of any morphisms between ADTs for arbitrary mangled memory representations. We also provide full synthesis of bijections between memory representations of the same type. We subsume existing compilation algorithms, and extend them to emit CFG-style programs with explicit memory allocation and full support for recursive types.

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

1 INTRODUCTION

Algebraic Data Types (ADTs) have proven themselves to be an essential tool for high-level programming: they allow to concisely model data thanks to sums, which indicate potential alternatives, and products, which group different pieces of data together. Thanks to their declarative nature, they let programmers manipulate data unbothered by the nitty-gritty details of its actual memory representation. That declarative nature allows compilers to verify and optimise code manipulating algebraic data, notably through pattern matching [Augustsson 1985; Maranget 2008, 2007]. This versatility and simplicity has allowed them to gain popularity, from their original grounds in functional programming languages [Burstall et al. 1980] like OCaml and Haskell, to mainstream languages such as Typescript, Python, and most recently Java.

Unfortunately, low-level programmers have so far not reaped the benefits of ADTs: they must often fall back to manual handling of memory layout to implement their data manipulation code, even in languages such as Rust which offer both ADTs and low-level programming. One main reason is that memory layouts for low-level data structures are indisputably *weird*: Red-Black Trees in the Linux kernel leverage low bits in aligned pointers to store information using the now classic bit-stealing technique [Torvalds 2023]; high-performance code regularly uses AoS (array of structs), SoA or AoSoA representations to mangle data collections for better locality [*AoS and SoA* 2023]; binary representations of data such as instruction sets and network packets regularly cut data into tiny pieces to minimise overall memory size. The general mold of Algebraic Data Types does not provide enough control over memory layout to model such mangled representations. As one might

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

imagine, the code required to manipulate such memory layouts is complex, error-prone, and hard to automatically verify and optimise.

Our goal is to provide high-level data-modelling constructs via ADTs, specify their precise memory layout, and obtain low-level efficient code conforming to that layout. Some works have attempted to bridge this gap. Dargent [Chen et al. 2023] lets programmers give high-level layout descriptions and generates certified C accessors and constructors. It however doesn't provide full language constructs like pattern matching. LoCal [Vollmer et al. 2019] and Gibbon [Koparkar et al. 2021] provide efficient compilation specialised for code operating on serialised and dense data representations. More generally, many programming languages such as Rust or Haskell provide both low-level vector types and high-level ADTs in separate manners, forcing programmers to resort to low-level code when they want to fine-tune their memory layout.

More recently, Ribbit [Baudon et al. 2023] proposes a dual-view compilation approach: a high-level type is paired with a low-level memory layout. A compilation algorithm then takes high-level pattern matching to low-level code respecting the layout. Their layout specification is expressive, allowing to specify many of the examples we just highlighted and scaling to fairly complex real-world examples. Their compilation algorithm, however, suffers from one crucial drawback: it can only *access and deconstruct* values. More formally, it only handles *surjective morphisms* between ADT representations. This not only limits the kind of code that can be compiled, but in fact also limits the choice of representations themselves, as highly mangled layouts might need data to be deconstructed, reshaped, and rebuilt differently. This is the case for aggressive struct packing, flattening or AoS/SoA transformations. To better understand this limitation, let us study two real-world examples of ADTs with complex layouts: the RISC-V instruction set with its binary representation, and optimised arithmetic expressions.

2 MOTIVATION

To demonstrate the complexity of real-world memory layouts for ADTs, we first consider a restricted version of the 32-bit RISC-V assembly language consisting of four instructions (`add`, `addi`, `sw`, and `jal`) in Fig. 1. We will use Ribbit's DSL to specify as a memory layout the encoding described in the instruction set (ISA) documentation [Waterman et al. 2019]. A RISC-V machine has 32 registers, `x0` to `x31` (encoded on 5 bits). As shown in Fig. 1a, RISC-V 32-bit instructions have different formats based on their addressing mode. Further characteristics of our four instructions are in Fig. 1b.

Already, we see complications: in general, instruction characteristics (type, instruction name, involved registers, ...) are spread over `opcode`, `funct3` and `funct7`, which are stored non-consecutively. Moreover, the latter two are sometimes not present in the 32-bit instruction value. Immediates are particularly mangled, and cannot be readily extracted from the binary representation. For our particular (simple) subset : (i) the four instructions are distinguishable from their `opcode` *only*, stored in bits 0 to 7 inclusive; (ii) the destination registers of `add` and `addi` are at the same location, bits 7 to 11; (iii) the immediate value (`imm`) for the `sw` instruction is split and stored in two bit ranges: bits 7 to 11 and 25 to 31; (iv) the 20-bit immediate value for the `jal` instruction can be recovered from bits 12 to 31 but we need to *rebuild* this immediate from four separate bit ranges.

We demonstrate the modelling of RISC-V registers and instructions with ADTs in Figs. 1c and 1d, using Ribbit [Baudon et al. 2023] syntax. Additionally, all examples in this article can be tried interactively at https://ribbit.gitlabpages.inria.fr/ribbit/. In addition to ADTs, Ribbit lets us define their *memory layouts*, which describe how concrete values are encoded in memory. Registers are encoded on 5 bits, similar to a C enum (e.g., `x2` is the 5-bit integer "3"). Instructions (`Op32` type) are encoded on 32 bits (`w32`). The `split` construct, on Line 3, allows to distinguish between the four possible cases using the 7 lowest bits (opcode). We only showcase the `Sw` and `Addi` branches of this

split. For `sw`, Line 13 specifies that the opcode value is `0x23` (see Fig. 1b). The immediate operand is broken down into two parts, which are encoded on bits 7 to 11 inclusive (Line 15) and 25 to 31 (Line 20) within the 32-bit word representing the full instruction.

2.1 Compilation of Constructors and Destructors

Now that types and layouts have been defined, high-level data manipulation constructs can be compiled to code which directly manipulates memory. For instance, Fig. 2 represents `sw(x1, x2, imm)` in memory. From a high-level perspective, this is a simple constructor: a typical representation for such a value would simply allocate an adequate amount of memory, and encode `x1`, `x2` and `imm` as integers at their assigned positions. Our representation, however, is not so straightforward: since `imm` is stored non-consecutively, we need to break it down into two pieces. In essence, we need to synthesize code manifesting the *isomorphism* between the previous representation of `imm` (here, a straight `i12`) and the representation embedded in `Op32` (two pieces at positions $[25:7]$ and $[7:5]$). Such implicit recombination of subterms is common in the context of embedded and low-level memory representations. A simple struct flattening and reordering already exhibits a similar behavior. Baudon et al. [2023]'s algorithm is unable to synthesize code that properly manipulates such representations.

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[0 : 12]		rs1	funct3	rd	opcode	I-type
imm[5 : 7]	rs2	rs1	funct3	imm[0 : 5]	opcode	S-type
imm[20]	imm[1 : 10] imm[11]	imm[12 : 7]		rd	opcode	J-type

(a) RISCV Core instruction format, excerpt. "rs1,2" are source registers, "rd" a destination register. "imm[n]" denotes the n-th bit of imm. "imm$[o:\ell]$" means "ℓ bits starting from o in the binary representation of imm".

Inst	Name	Type	Opcode	funct3	funct7	Description (in C)
add	Add	R	0x33	0	0	`rd = rs1 + rs2`
addi	Add Immediate	I	0x13	0	—	`rd = rs1 + imm`
sw	Store Word	S	0x23	2	—	`*(rs1+imm) = rs2`
jal	Jump And Link	J	0x6F	—	—	`rd = PC+4; PC += imm`

(b) Instruction semantics and encoding, excerpt.

```
1 enum Reg { X0, X1, X2, X3, ... } // cut
2 represented by C
3
4 enum Op32 {
5   Add(Reg, Reg, Reg), // add rd, rs1, rs2
6   Addi(Reg, Reg, i12), // addi rd, rs1, imm12
7   Jal(Reg, i20), // jal rd, imm20
8   Sw(Reg, Reg, i12), // sw rs1, rs2, imm12
9 }
```

(c) Algebraic Data Types for 32-bit RISC-V instructions. iN types are predefined as "integers on N bits".

```
1  Op32 represented as
2  // discriminant in the 7 lowest bits
3  split .[0:7] {
4    | 0x13 from Addi =>
5      w32
6      with .[0:7] = 0x13 // opcode
7      with .[7:5] : (_.Addi.0 as Reg32)
8      with .[12:3] = 0
9      with .[15:5] : (_.Addi.1 as Reg32)
10     with .[20:12] : (_.Addi.2 as w12)
11   | 0x23 from Sw =>
12     w32
13     with .[0:7] = 0x23 // opcode
14     // 5 lowest bits from imm
15     with .[7:5]:(_.Sw.2.[0:5] as w5)
16     with .[12:3] = 2 // funct3
17     with .[15:5]:(_.Sw.0 as Reg32) // rs1
18     with .[20:5]:(_.Sw.1 as Reg32) // rs2
19     // 7 highest bits from imm
20     with .[25:7]:(_.Sw.2.[5:7] as w7)
21   | // other cases cut for readability
```

(d) Ribbit modelling for RISC-V instructions. wN types are predefined as "words on N bits". Bit ranges consist of a left bound and a size.

Fig. 1. 32-bit RISC-V instructions – Specification, Type and Memory Layout in Ribbit

```
1 let out o = alloc(32);
2 o.[12:+3] := 2; o.[0:+5] := 0x23; // funct3, opcode
3 o.[15:+5] := 1; o.[20:+5] := 2; // rs1 = X1, rs2 = X2
4 o.[25:+7] := imm.[5:+7]; o.[7:+5] := imm.[0:+5]; // imm
```

Fig. 2. Code building the memory representation of `Sw(X1, X2, imm)` in the root memory location o.

```
fn is_compressible(o : Op32) -> Bool {
  match o {
    Jal(X1, off) => off < 4096,
    Add(rd, rs1, rs2) => rd == rs1 && rs1 != X0 && !(rs2 == X0),
    Addi(rd, rs, imm) => rd == rs && rs != X0 && imm < 64,
    Sw(rbase, roff, imm) =>
      X8 <= rbase && rbase <= X15 && X8 <= roff && roff <= X15 && (imm & 0b110000011111) == 0,
    _ => False,
  }}
```

Fig. 3. Function determining whether a given 32-bit instruction can be compressed into a 16-bit one, in simplified Ribbit syntax. Full interactive example available at https://ribbit.gitlabpages.inria.fr/ribbit/#riscv.rbt
Conditions taken from [Waterman et al. 2019, chapter RISCV-C].

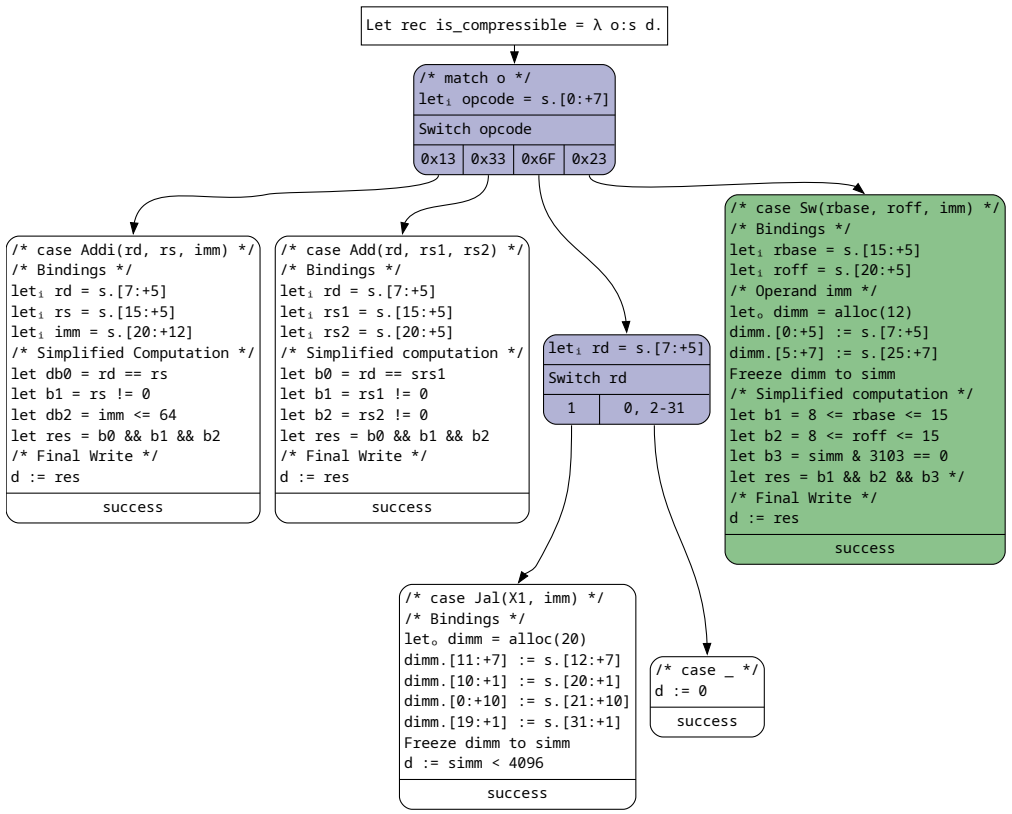


Fig. 4. Simplified CFG for `is_compressible`. From the input `i`, it identifies the head constructor using the 7 lowest bits, then extracts subterms such as destination and source registers for `Add` or the 12-bit `imm` for `Sw` (in bold), and finally stores the result in `dest`.

Such transformations can also be necessary when *destroying* values. Figure 3 defines a function that determines whether a given 32-bit RISC-V instruction can be compressed into a 16-bit RISC-V instruction. However, again, we can not immediately bind the `imm` value during pattern matching compilation and need to first find all needed pieces and combine them together. Baudon et al. [2023]'s algorithm is thus not capable of compiling the `is_compressible` function.

Our compilation procedure is capable of emitting the control flow graph depicted in Fig. 4, which:

- *inspects* the internal representation of an input `Op32` value to determine its head constructor (`Add`, `Addi`, `Jal` or `Sw`), as well as the nested register constructor in `Jal`;
- *extracts* from this representation all subterms that are bound to variables in the matched pattern. For instance, in the `Sw` case, the parts of the immediate `imm` are combined in `simm` in order to reconstruct a value that can be used in a mask;
- *allocates* and *initialises* memory to represent the appropriate values. For instance, the `imm` value just mentioned is first allocated as `dimm`, filled, then promoted to a read-only value `simm` before being used.

2.2 Morphisms of Complex Types

So far, the type we had to transform, while having an intricate representation, consisted of fairly flat simple structures. Let us consider a richer example with simple arithmetic expressions over 32-bit integers in Fig. 5. As described by the `Exp` type, arithmetic expressions consist of variable names, 32-bit integer constants and operations with two arguments (for simplicity of the example, we only have additions here). The host machine is assumed to be a 64-bit architecture. A naive representation of operands would be a pointer to a 3-word structure: a first word containing a few bits (here, 32) representing the operation, followed by two 64-bit-wide aligned pointers to the operands. However, in cases such as Plus($I(12), \ldots$), this is wasteful: we could simply pack the 32-bit operator with the left operand, and use one less memory word. In Fig. 5a, we define this optimized memory layout. Integer terms are boxed in a 64-bit word when isolated (Line 13), but are inlined (i.e., directly represented as their 32-bit integer value) when they appear as operands of a binary operation (Line 7). While this reduces memory usage and indirections compared to a naive, uniform layout, it pervasively impacts the compilation process. Since the very structure (pointers/indirections, struct shapes) of the memory representation of an integer expression is different depending on the context in which it appears, some seemingly simple constructs – pattern matching and value construction – require complex code to properly manipulate data. For all these reasons, such optimizations are only done by programmers when absolutely necessary (such as extremely performance-sensitive code). As we will see, our approach can compile any valid program given its layout, making such complex layouts completely transparent to client code.

The `eval` function (shown in Fig. 5b), reduces all arithmetic expressions which can be evaluated. When both operands are integers, it reduces further (Fig. 5b, Line 5). Figure 5c shows the normalized program representation we use in the rest of this article, and Fig. 5d shows the final compiled version, using a Control Flow Graph in *Destination Passing Style* [Shaikhha et al. 2017] with segregated input (read-only) and output (write only) memory locations, allowing fine control over memory. Corresponding elements of `eval` in normalized source and final target code are colorized accordingly.

Consider the code which extracts both operands of Plus (on Line 4). Given the complex layout, this is not a straight-up pointer dereference: we need to reconstruct values fitting the type `Exp` to be used as arguments to `eval`. In other words, we need to *synthesize an isomorphism* between the stored memory representation of *.Plus*.0 (inlined integer), and the desired representation (a standard `Exp`). The generated code creates two destinations to receive contents, inspects the inner tag (`tag2`, stored in $x. * .0$) to determine which operands, if any, are inlined integers, and emits code to fill them. As we can see, a simple accessor at the source level might require us to emit code that rebuild values.

Consider now the final code returning Plus($e1', e2'$) (on Line 13). It is naturally split in three cases: depending on whether $e1'$ and $e2'$ are I constructors or not. Note that this decision is factorized

```
enum Exp { V(String), I(i32), Plus(Exp, Exp) }
represented as
split .[0, 2] {
| 0 from Plus =>
  &<64, 2>(split .0 {
    | 0 from Plus(I, _) =>
      {{w32, _.Plus.0.I as w32, _.Plus.1 as Exp}}
    | 1 from Plus(Plus|V, I) =>
      {{w32, _.Plus.1.I as w32, _.Plus.0 as Exp}}
    | 2 from (Plus(Plus|V,Plus|V)) =>
      {{w32, _32, _.Plus.0 as Exp, _.Plus.1 as Exp}}
  })
| 1 from I => w64 with .[32, 32] : (_.I as w32)
| 2 from V => &<64, 2>(_.V as String)
}
```

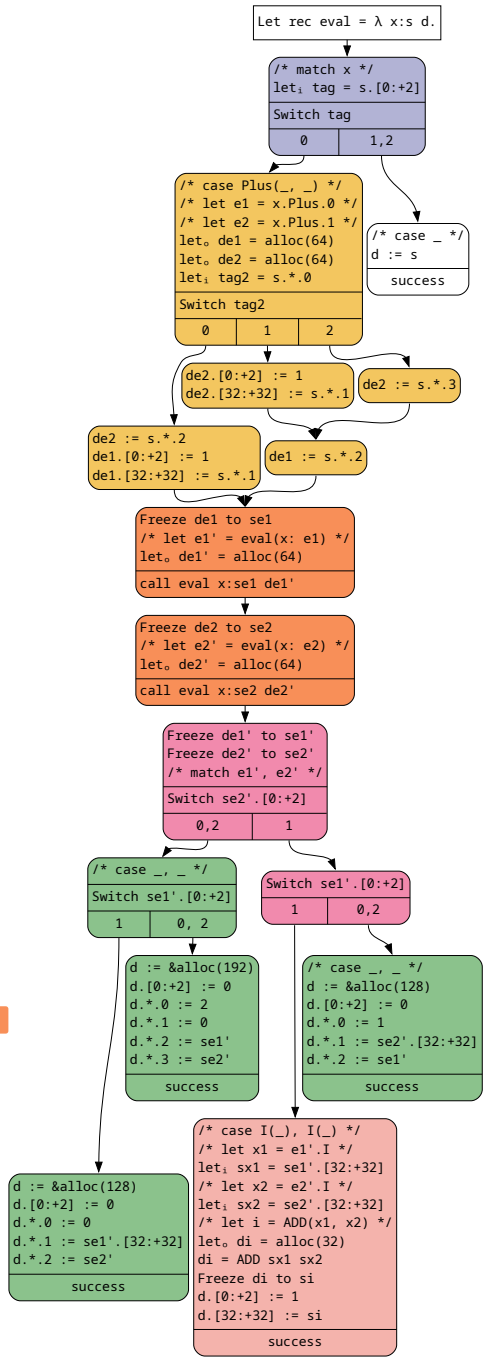
(a) Arithmetic expressions and their representation. On the first layer, this representation stores the tag distinguishing between V, I, and Plus in the two lowest bits. This potentially leverages bit-stealing in pointer alignment bits. For Plus constructors, we use a struct whose first field indicates the more precise layout. Some space could be used to store an operator (`MULT`, …) as well. Fields are arranged to keep pointers in aligned positions.

```
fn eval(x:Exp) -> Exp {
  match x {
    I(_) | V(_) => x,
    Plus(e1,e2) => match eval(e1), eval(e2) {
      I(x1), I(x2) => I(x1+x2),
      e1', e2' => Plus(e1', e2')
    }}}
```

(b) User implementation of eval.

```
fn eval(x: Exp) : Exp {
  match x {
    Plus(_, _) =>
      let e1:Exp = x.Plus.0; let e2:Exp = x.Plus.1;
      let e1':Exp = eval(e1); let e2':Exp = eval(e2);
      match e1', e2' {
        I(_), I(_) =>
          let x1 : i32 = e1'.I;
          let x2 : i32 = e2'.I;
          let i : i32 = ADD(x1, x2);
          I(i),
        _, _ =>
          Plus(e1', e2'),
      },
    _ => x,
    }}
```

(c) Normalized representation of eval. Explicitly typed A-Normal Form version of the code, where patterns have no variables, and accesses are made through paths instead. For instance x.Plus.0 is the first operand under a Plus constructor in x. Such accesses are only valid under the right pre-conditions.



(d) Simplified CFG of eval. For pedagogic and readability purposes, code has been simplified (block sinking, variable renaming, simple constant propagation).

Fig. 5. Arithmetic expressions, their representation, and the eval function in Ribbit. Full interactive example available at https://ribbit.gitlabpages.inria.fr/ribbit/#arith_expr_icfp.rbt.

with the pattern matching above. Again, in all cases, the emitted code allocates and places every bit in memory. Again, we had to synthesize a morphism between the memory representation of the present pieces ($e1'$ and $e2'$) and the one contained inside a Plus constructor.

Naturally, to reap the full power of our tweaked representation, one would need to unroll the `eval` function, allowing to completely skip some intermediary products. We consider such transformations orthogonal to our contribution, and focus on emitting straightforward code that is easily optimized by existing state-of-the-art transformations (here, unrolling and constant propagation).

As shown here, in order to properly compile programs manipulating complex representations, Ribbit can generate morphisms: code converting between different memory representations of the same source type. In fact, programmers themselves can also leverage that feature! For instance, given a naive AST-style representation of arithmetic expressions `ExpAST` and our optimised version `Exp`, both sharing the same source definition, we can readily write the following function:

```
fn convert(x:ExpAST) -> Exp {x}
```

Our approach will readily generate the full (recursive!) code that manifests the isomorphism between these two structures.

All these behaviors, and all examples developed in this article, can be experimented upon interactively on https://ribbit.gitlabpages.inria.fr/ribbit/. Our prototype online implementation can verify validity of types, memory layouts and programs; compile programs to our intermediate representation, show the obtained programs, and run them. In addition, it will verify correction of the obtained compiled code against a reference source interpreter.

2.3 Contribution and Outline

In this article, we define a general procedure which compiles (potentially nested) accessors, constructors, pattern matching, and *both* together, without introducing superfluous work.

Our approach works for rich and custom memory representations in the style of Baudon et al. [2023], and is able to compile any well-typed high-level ADT-manipulating program, including in the presence of recursive types and recursive code emission. For this purpose, our approach can synthesise, when necessary, the code manifesting any isomorphism between memory representations of compatible types. We emit low-level code with precise memory manipulation thanks to the use of a destination passing intermediary representation [Shaikhha et al. 2017]. We also state the correctness of our algorithms against a source-level semantics. Finally, we implemented our algorithm (an online version can be found at https://ribbit.gitlabpages.inria.fr/ribbit/), evaluated it on numerous examples, and tested the correctness of our compilation procedure against a reference interpreter.

Section 3 describes our input language based on types and memory layouts from Baudon et al. [2023]. Section 4 presents our target intermediate representation. Our approach relies on existing pattern matching compilation techniques, which we detail in Section 5. From a first partial algorithm, in Section 6, working on simple (not "intricate") representations, we then derive our main contribution, detailed in Section 7: a complete compilation algorithm for expressions constructing ADT values according to custom memory layouts. Finally, we sketch the metatheoretic properties of our algorithms in Section 8.

3 ALGEBRAIC DATA TYPES AND THEIR LAYOUTS

We first present our input language. From the type perspective, as highlighted in Section 2, we use a two-tiered view: *algebraic data types* used for programming and *memory layouts* detailing how to represent them in memory. The formal presentation of these types and layouts extends Baudon et al. [2023] with most of its limitations lifted. In addition, we introduce a complete expression language to both destruct and construct values of such types.

$$
\begin{array}{rll}
\tau \in \mathit{Types} ::= & t \in \mathit{TyVars} & \text{(variables)} \\
& \mid I_\ell & (\ell\text{-bit wide integers}) \\
& \mid \langle \tau_0, \ldots, \tau_{n-1} \rangle & \text{(tuples)} \\
& \mid K_1(\tau_1)\text{`|`} \ldots \text{`|`} K_n(\tau_n) & \text{(sums)} \\
v \in \mathit{Values} ::= & c \mid \langle v_0, \ldots, v_{n-1} \rangle \mid K(v) & \text{(values)}
\end{array}
\qquad
\begin{array}{rll}
p \in P \subseteq \mathit{Provs} ::= & \_ \mid \langle p_0, \ldots, p_{n-1} \rangle \mid K(p) & \\
& & \text{(provenances and their sets)} \\
\pi \in \mathit{Paths} ::= & \epsilon & \text{(empty path)} \\
& \mid \pi.[o:\ell] & (\ell \text{ bits at offset } o) \\
& \mid \pi.i & \text{(field access)} \\
& \mid \pi.K & \text{(constructor access)}
\end{array}
$$

Fig. 6. Algebraic Data Types and other high-level objects

As general meta-syntactic conventions:

- Syntactic categories are in *italics* (*TyVars*, *Exprs*, …).
- Each syntactic category has a dedicated meta-syntactic letter (e for elements of *Exprs*, …) and a capital for sets (E for sets of *Exprs* elements, …)
- Lists, sets and maps are indicated either *in-extenso* ($\tau_0, \ldots, \tau_{n-1}$), with a Python-like notation ($\{e_i \mid 0 \leq i < n\}$), or with a sequence-like notation ($(x_i)_{1 \leq i \leq n}$).

3.1 Algebraic Data Types

The grammar for Algebraic Data Types is presented in Fig. 6. We denote types using τ and type variables with $t \in \mathit{TyVars}$. Primitive types consist of fixed-width signed integers, denoted I_ℓ. We denote all tuples with angle brackets, for instance $\langle I_{32}, I_{32} \rangle$ for pairs of 32-bit integers. Constructors of sums are marked with a capital letter and separated from each other with vertical bars, for instance "None | Some(t)" is an option type. In examples, we use K as shortcut for $K(\langle\rangle)$ and $K(\tau_1, \ldots, \tau_n)$ for $K(\langle \tau_1, \ldots, \tau_n \rangle)$. *Values* of Algebraic Data Types, dubbed "high-level values", consist of integer constants denoted c, tuples and constructors.

We now define two related constructs: *provenances* and *paths*. Provenances give partial information on what a value looks like: a tuple, a head constructor with nested provenances, or anything (wildcard: _). Paths precisely indicate the position of a subterm in a given type, value or provenance, using position accesses $.i$ to deconstruct tuples, head constructors $.K$ for constructor values and bit ranges $.[o:\ell]$ consisting of an offset o and a width ℓ for integers.

Focusing and specialisation. We also define two accompanying operations, which will be demonstrated on examples below. The *focusing* operation, denoted **focus** (π, θ) where $\theta \in \mathit{Types} \cup \mathit{Values} \cup \mathit{Provs} \cup \mathit{Paths}$ is a high-level object (type, value, provenance or path), accesses the subterm of θ located at π. It follows the syntax to extract the subterm at position π. The *specialisation* operation on ADTs, denoted $\tau \,/\, p$, restricts a type τ to values that match a given provenance p. It syntactically filters irrelevant constructors. The full definitions for both can be found in Appendix A.

Example 3.1 (High-level types and values). Our high-level type for RISC-V 32-bit instructions is a sum type with four constructors:

$$\tau_{\text{RISC-V}} = \text{Add}(\tau_{\text{reg}}, \tau_{\text{reg}}, \tau_{\text{reg}}) \mid \text{Addi}(\tau_{\text{reg}}, \tau_{\text{reg}}, I_{12}) \mid \text{Jal}(\tau_{\text{reg}}, I_{20}) \mid \text{Sw}(\tau_{\text{reg}}, \tau_{\text{reg}}, I_{12})$$

where τ_{reg} is a simple sum type enumerating the 32 available RISC-V registers: $\tau_{\text{reg}} = X_0 \mid \cdots \mid X_{31}$.

Given this definition, the high-level value $\text{Addi}(X_2, X_2, 3)$ of type $\tau_{\text{RISC-V}}$ represents the instruction `addi x2, x2, 3`. The path .Addi.0 designates the first argument of any `addi` instruction, namely its destination register. We can focus on "the part that is relevant to .Addi.0" in the type $\tau_{\text{RISC-V}}$:

$$\textbf{focus}\,(.\text{Addi.0}, \tau_{\text{RISC-V}}) = \textbf{focus}\left(.\text{Addi.0}, \text{Addi}(\tau_{\text{reg}}, \tau_{\text{reg}}, I_{12})\right) = \tau_{\text{reg}}$$

$$
\begin{array}{lll}
\widehat{\tau} \in \widehat{\mathit{Types}} ::= t \in \mathit{TyVars} & & \text{(type variable)} \\
\quad \mid I_\ell & & (\ell\text{-bit wide integers}) \\
\quad \mid (c)_\ell & & (\ell\text{-bit wide constant}) \\
\quad \mid \__\ell & & (\ell\text{-bit wide opaque word}) \\
\quad \mid \&_\ell\,(\widehat{\tau}) & & (\ell\text{-bit wide pointer to a } \widehat{\tau} \text{ value}) \\
\quad \mid \widehat{\tau} \bowtie\limits_{0 \le i < n} [o_i : \ell_i] : \widehat{\tau}_i & & (\widehat{\tau} \text{ value with } n \text{ extra values stored in unused bit ranges}) \\
\quad \mid \{\!\{\widehat{\tau}_0, \ldots, \widehat{\tau}_{n-1}\}\!\} & & (n\text{-field struct}) \\
\quad \mid (\pi \text{ as } \widehat{\tau}) & & (\text{fragment representing the subterm at position } \pi \text{ as } \widehat{\tau}) \\
\quad \mid \text{split}\,(\widehat{\pi})\,\left\{c_i \text{ from } P_i \Rightarrow \widehat{\tau}_i \;\middle|\; 1 \le i \le n\right\} & & (n\text{-branch split with discriminant } \widehat{\pi}) \\
\widehat{\pi} \in \widehat{\mathit{Paths}} ::= \epsilon & & \text{(empty memory path)} \\
\quad \mid .[o : \ell].\widehat{\pi} & & (\text{extract } \ell \text{ bits from offset } o \text{ from a word}) \\
\quad \mid .b_0 \ldots b_{\ell-1}.\widehat{\pi} \qquad b_i \in \{0, 1\} & & \text{(bitwise “and”)} \\
\quad \mid .*.\widehat{\pi} & & \text{(pointer dereference)} \\
\quad \mid .i.\widehat{\pi} & & \text{(struct field access)}
\end{array}
$$

Fig. 7. Memory layouts and paths

and similarly, in our high-level value: $\mathbf{focus}\,(.\mathrm{Addi}.0, \mathrm{Addi}(X_2, X_2, 3)) = X_2$. The provenance $\mathrm{Sw}(X_2, \_, \_)$ matches `sw` instructions whose destination register is `x2`. The specialisation of $\tau_{\text{RISC-V}}$ for this provenance is the following type: $\tau_{\text{RISC-V}} \,/\, \mathrm{Sw}(X_2, \_, \_) = \mathrm{Sw}(X_2, \tau_{\text{reg}}, I_{12})$ △

3.2 Memory Layouts

Each ADT is associated with a *memory layout* which specifies how its values should be represented in memory. As a general convention, memory elements are distinguished with a hat. Memory layouts, denoted $\widehat{\tau}$ and defined in Fig. 7, feature two types of constructs: some describe concrete memory structures, others refer back to the represented high-level type. Concrete memory structures consist of opaque words $\__\ell$ representing ℓ unused contiguous bits, fixed-width integer constants and pointers, or structs aggregating several fields together. Finally, composite words, denoted $\widehat{\tau} \bowtie_{0 \le i < n} [o_i : \ell_i] : \widehat{\tau}_i$, specify the contents of n unused bit ranges within a word type $\widehat{\tau}$. Unused bits include all parts of opaque words and architecture/implementation-dependent locations such as address alignment bits in pointers. In any case, the n bit ranges must not overlap. Constructs that refer back to the represented high-level type consist of fixed-width encodings I_ℓ for values of integer types (the precise encoding used is implementation-dependent), and of two more complex constructs: fragments and splits.

Fragments refer to subterms within the high-level type and indicate their layout: the fragment $(\pi \text{ as } \widehat{\tau})$ stands for the representation according to the layout $\widehat{\tau}$ of the subterm located at π within the high-level value being represented. Fragments may refer to arbitrarily nested subterms.

Splits create disjunctions between different possible layouts by designating a position in memory as a *discriminant*: $\text{split}\,(\widehat{\pi})\,\left\{c_i \text{ from } P_i \Rightarrow \widehat{\tau}_i \;\middle|\; 1 \le i \le n\right\}$ indicates that if the contents of memory at position $\widehat{\pi}$ are equal to c_i, then the memory value under scrutiny follows the layout $\widehat{\tau}_i$ and represents a high-level value whose provenance belongs to the set P_i. The validity of a split construct depends on each $\widehat{\tau}_i$ actually containing the value c_i (specified as a constant $(c_i)_\ell$) at position $\widehat{\pi}$: it ensures

that it is possible to retrieve the provenance of any value from its representation only, by inspecting its contents at $\widehat{\pi}$.

Example 3.2 (Memory layout for RISC-V). Here are the layouts of the types from Example 3.1:

$$\widehat{\tau}_{\text{reg}} = \text{split}\,(\epsilon)\begin{Bmatrix} 0 \;\text{ from } X_0 \;\Rightarrow (0)_5 \\ \vdots \\ 31 \text{ from } X_{31} \Rightarrow (31)_5 \end{Bmatrix} \quad \widehat{\tau}_{\text{RISC-V}} = \text{split}\,(.[0:7])\begin{Bmatrix} \texttt{0x33} \text{ from Add}(\_,\_,\_) & \Rightarrow \widehat{\tau}_{\text{Add}} \\ \texttt{0x13} \text{ from Addi}(\_,\_,\_) & \Rightarrow \widehat{\tau}_{\text{Addi}} \\ \texttt{0x6F} \text{ from Jal}(\_,\_) & \Rightarrow \widehat{\tau}_{\text{Jal}} \\ \texttt{0x23} \text{ from Sw}(\_,\_,\_) & \Rightarrow \widehat{\tau}_{\text{Sw}} \end{Bmatrix}$$

The split describes how to distinguish between instructions, by inspecting the 7 lowest bits (`0xXX` are fixed constants). Layouts for individual instructions (e.g., Sw) are built from an opaque 32-bit word $\__{32}$ whose contents are specified by constraints on subranges of bits:

$$\begin{aligned} \widehat{\tau}_{\text{Sw}} = \__{32} &\ltimes [0:7] : (\texttt{0x23})_7 \ltimes [12:3] : (2)_3 && \text{(opcode and funct3 const)} \\ &\ltimes [15:5] : (.\text{Sw.0 as } \widehat{\tau}_{\text{reg}}) \ltimes [20:5] : (.\text{Sw.1 as } \widehat{\tau}_{\text{reg}}) && \text{(registers rs1 and rs2)} \\ &\ltimes [7:5] : (.\text{Sw.2.}[0:5] \text{ as } I_5) \ltimes [25:7] : (.\text{Sw.2.}[5:7] \text{ as } I_7) && \text{(imm in split ranges)} \end{aligned}$$

For instance, the fragment $(.\text{Sw.0 as } \widehat{\tau}_{\text{reg}})$ expresses that bits 15 to 19 inclusive contain the representation of the first argument of Sw (base register, `rs1`) according to the $\widehat{\tau}_{\text{reg}}$ layout. △

Example 3.3 (Memory layout for lists). Consider lists of 32-bit integers $\tau = \text{Nil} \mid \text{Cons}(I_{32}, \tau)$, and a packed layout with up to two elements per level of indirection, with three branches: empty or a singleton list – both immediately encoded – or a pointer to a block of two integers and a list:

$$\begin{aligned} &\widehat{\tau}_p = \text{split}\,(.[0:2])\,\{ \\ &\quad 0 \text{ from Nil} && \Rightarrow \__{64} \\ &\quad 1 \text{ from Cons}(\_, \text{Nil}) && \Rightarrow \__{64} \ltimes [2:32] : (.\text{Cons.0 as } I_{32}) \\ &\quad 2 \text{ from Cons}(\_, \text{Cons}(\_,\_)) && \Rightarrow \\ &\qquad \&_{64}\,(\{\!\{(.\text{Cons.0 as } I_{32}), (.\text{Cons.1.Cons.0 as } I_{32}), (.\text{Cons.1.Cons.1 as } \widehat{\tau}_p)\}\!\}) \\ &\} \end{aligned}$$

△

Memory Paths and Focusing. Similarly to high-level types, we define paths, provenances and focusing for memory layouts. Memory paths, denoted $\widehat{\pi}$ (see Fig. 7), indicate positions in layouts. Memory path operations consist of struct field accesses, pointer dereference, and two operations that deconstruct composite words: bit range extraction to focus on the contents of a specific bit range and bitwise "and" to discard ranges that lie outside of a bit mask. Similarly to high-level paths, we define the *memory focus* operation $\widehat{\textbf{focus}}\,(\widehat{\pi}, \widehat{\tau})$ that extracts the subterm at position $\widehat{\pi}$ within $\widehat{\tau}$ (it is undefined on fragments and splits; the full definition is in Appendix A).

Example 3.4 (Paths and Focusing for $\widehat{\tau}_{\text{RISC-V}}$). The memory path $.[15:5]$ denotes the 5 bits starting from offset 15 (included) within a word. Focusing on this path in $\widehat{\tau}_{\text{Sw}}$ yields the fragment corresponding to the base register of a `sw` instruction: $\widehat{\textbf{focus}}\,(.[15:5], \widehat{\tau}_{\text{Sw}}) = (.\text{Sw.0 as } \widehat{\tau}_{\text{reg}})$. △

3.3 Input Programs

We can now define high-level programs that manipulate ADT values according to a given memory layout specification. Our general goal is to compile such programs to low-level programs that operate on memory directly, with respect to the specified memory layout. Our input language consists of expressions, denoted e and defined in Fig. 8. For presentation purposes, we consider an explicitly-typed, already simplified input language, corresponding to the internal representation

$$
\begin{array}{lll}
u \in \mathit{ValueExprs} ::= x.\pi & \text{(variable accessor)} & e \in \mathit{Exprs} ::= u \qquad \text{(value building)} \\
\mid c & \text{(integer constant)} & \mid \mathsf{let}\ x : \tau\ \mathsf{as}\ \widehat{\tau} = e\ \mathsf{in}\ e' \\
\mid \langle u_0, \ldots, u_{n-1} \rangle & \text{(tuple)} & \qquad\qquad \text{(let-binding)} \\
\mid K(u) & \text{(constructor)} & \mid \mathsf{match}(x)\{p_1 \to e_1 \mid \cdots \mid p_n \to e_n\} \\
 & & \qquad\qquad \text{(pattern matching)}
\end{array}
$$

Fig. 8. Program syntax.

$$
\begin{array}{rll}
a \in \mathit{Addrs} \qquad \varsigma : \mathit{Addrs} \to \widehat{\mathit{Values}} & & \text{(addresses and stores)} \\
\widehat{v} \in \widehat{\mathit{Values}} ::= \__\ell \mid (c)_\ell \mid \&_\ell\,(a) \mid \widehat{v} \underset{0 \le i < n}{\bowtie} [o_i : \ell_i] : \widehat{v}_i \mid \{\!\{\widehat{v}_0, \ldots, \widehat{v}_{n-1}\}\!\} & & \text{(memory values)} \\
\widehat{p} \in \widehat{\mathit{Provs}} ::= \__\ell \mid (c)_\ell \mid \&_\ell\,(\widehat{p}) \mid \widehat{p} \underset{0 \le i < n}{\bowtie} [o_i : \ell_i] : \widehat{p}_i \mid \left\{\!\!\left\{\widehat{p}_0, \ldots, \widehat{p}_{n-1}\right\}\!\!\right\} & & \text{(memory shapes)}
\end{array}
$$

Fig. 9. Memory contents and shapes.

from Fig. 5c. Expressions are in A-normal form: every value is let-bound to a variable. Programs manipulate ADT values through *pattern matching* to destruct values, and *value-building expressions* to construct values according to a given memory layout.

A pattern-matching expression contains a list of *cases* of the form $p \to e$, each consisting of a provenance on the left-hand side and an expression on the right-hand side. Note that provenances contain no variables. In some of our examples, we will use *or-patterns* such as $p_1 \mid p_2 \to e$.

Valuexpressions, denoted u, allow to define values with reference to other already-bound values, using accessors of the form $x.\pi$. Such accessors designate subterms at position π within an existing value bound to x.

Example 3.5 (Valuexpression). The valuexpression $u = \mathrm{Jal}(x.\mathrm{Jal}.0, 42)$ builds a `jal` instruction using the same destination register as another `jal` instruction bound to x, and the constant immediate 42. Note that focusing on a valuexpr does *not* evaluate accessors: we have $\mathbf{focus}\,(.\mathrm{Jal}.0, u) = x.\mathrm{Jal}.0$. △

Example 3.6 (RISC-V destination register binding). The following program extracts the destination register of a RISC-V instruction if it exists, and returns X_0 otherwise:

$$
\mathrm{get\_dest}(x : \tau_{\text{RISC-V}}\ \mathsf{as}\ \widehat{\tau}_{\text{RISC-V}}) : \tau_{\mathrm{reg}}\ \mathsf{as}\ \widehat{\tau}_{\mathrm{reg}} = \mathsf{match}(x)\ \left\{ \begin{array}{ll} \mathrm{Add}(\_,\_,\_) & \to x.\mathrm{Add}.0 \\ \mathrm{Addi}(\_,\_,\_) & \to x.\mathrm{Addi}.0 \\ \mathrm{Jal}(\_,\_) & \to x.\mathrm{Jal}.0 \\ \mathrm{Sw}(\_,\_,\_) & \to X_0 \end{array} \right\} \qquad \triangle
$$

3.4 Memory Model

We model memory contents with values and stores, defined in Fig. 7. Memory values, denoted $\widehat{v}$, follow roughly the same syntax as layouts, excluding constructs that do not relate to concrete memory contents (integer encodings, fragments and splits). The main difference is that pointers do not directly contain the pointee memory value, but refer to it with an address a. Interpretation of memory values therefore relies on a store, denoted ς, which maps addresses to memory values.

Example 3.7 (RISC-V memory values). Consider the high-level value $v = \mathrm{Sw}(X_2, X_2, 3)$, of type $\tau_{\text{RISC-V}}$. The memory value representing X_2 according to $\widehat{\tau}_{\mathrm{reg}}$ is $(2)_5$. We can similarly follow the layout $\widehat{\tau}_{\text{RISC-V}}$ to represent v, by instantiating each split, fragment and non-constant integer with v contents, and get the following memory value (which corresponds to the "assembled" version of

$$
\begin{array}{ll}
\textbf{shape\_of}_\Delta\{ & \\
\quad t & \longrightarrow \textbf{shape\_of}_\Delta(\Delta(t)) \\
\quad I_\ell \mid \__\ell & \longrightarrow \__\ell \\
\quad (c)_\ell & \longrightarrow (c)_\ell \\
\quad \&_\ell\,(\widehat{\tau}) & \longrightarrow \&_\ell\,(\textbf{shape\_of}_\Delta(\widehat{\tau})) \\
\quad \widehat{\tau} \underset{0\le i<n}{\ltimes} [o_i : \ell_i] : \widehat{\tau}_i & \longrightarrow \textbf{shape\_of}_\Delta(\widehat{\tau}) \underset{0\le i<n}{\ltimes} [o_i : \ell_i] : \textbf{shape\_of}_\Delta(\widehat{\tau}_i) \\
\quad \{\!\{\widehat{\tau}_0, \ldots, \widehat{\tau}_{n-1}\}\!\} & \longrightarrow \{\!\{\textbf{shape\_of}_\Delta(\widehat{\tau}_0), \ldots, \textbf{shape\_of}_\Delta(\widehat{\tau}_{n-1})\}\!\} \\
\quad (\pi \text{ as } \widehat{\tau}) & \longrightarrow \textbf{shape\_of}_\Delta(\widehat{\tau}) \\
\quad \text{split}\,(\widehat{\pi})\,\{c_i \text{ from } P_i \Rightarrow \widehat{\tau}_i \mid 1 \le i \le n\} & \longrightarrow \__\ell \quad \text{where } \ell = \max_{1\le i\le n} |\widehat{\tau}_i| \\
\} &
\end{array}
$$

Fig. 10. Translation from layouts to shapes in type environment $\Delta : \mathit{TyVars} \to \widehat{\mathit{Types}}$

the instruction):

$$
\begin{aligned}
\widehat{v} = \__{32} \ltimes [0:7] : (\texttt{0x23})_7 \ltimes [7:5] : (3)_5 \ltimes [12:3] : (2)_3 \ltimes [15:5] : (2)_5 \\
\ltimes [20:5] : (2)_5 \ltimes [25:7] : (0)_7
\end{aligned}
\qquad \triangle
$$

Memory shapes. Finally, we formalize the notion of a memory value "fitting" a given layout with *memory shapes*, denoted $\widehat{p}$ and defined in Fig. 9. A shape exactly describes the "concrete" part of a memory type. The function **shape_of** in Fig. 10 is a conservative, best-effort translation from memory layouts to shapes used to gather static knowledge about memory layout. It uses a notion of size, which is defined in Appendix A. Here, the shape of a split is an opaque word large enough to contain any of its branches. However, a more precise definition is also possible, and even desirable, as we will see later.

Example 3.8 (Shapes for RISC-V). The shape of the layout used for RISC-V instructions in our running example (Example 3.2), considered in an empty type environment, depends on the size of each split branch. The layout in each of these branches is a composite word based on a 32-bit opaque word, hence $|\widehat{\tau}_{\text{Add}}| = |\widehat{\tau}_{\text{Addi}}| = |\widehat{\tau}_{\text{Jal}}| = |\widehat{\tau}_{\text{Sw}}|$ and $\textbf{shape\_of}(\widehat{\tau}_{\text{RISC-V}}) = \__{32}$. Similarly, we have $\textbf{shape\_of}(\widehat{\tau}_{\text{reg}}) = \__5$. The shape of the layout used for `sw` instructions follows its composite word structure and includes all of its fixed immediates:

$$
\begin{aligned}
\textbf{shape\_of}(\widehat{\tau}_{\text{Sw}}) = \__{32} \ltimes [0:7] : (\texttt{0x23})_7 \ltimes [7:5] : \__5 \ltimes [12:3] : (2)_3 \ltimes [15:5] : \__5 \\
\ltimes [20:5] : \__5 \ltimes [25:7] : \__7
\end{aligned}
\qquad \triangle
$$

4 TARGET IN DESTINATION PASSING STYLE

Before formally defining our target intermediate representation, let us illustrate it on a first example.

Example 4.1 (Target IR for RISC-V destination register binding). The program from Example 3.6 reads from an input location x_{in} representing a high-level value of type $\tau_{\text{RISC-V}}$ using the layout $\widehat{\tau}_{\text{RISC-V}}$, and writes its result of type τ_{reg} to an output location x_{out} using the layout $\widehat{\tau}_{\text{reg}}$. Its compiled

$$
\begin{array}{rll}
\mathcal{E} ::= & \texttt{success} \mid \texttt{fail} & \text{(return statements)} \\
\mid & \texttt{switch}\,(x_{\text{in}})\,\{c_1 \to \mathcal{E}_1 \mid \cdots \mid c_n \to \mathcal{E}_n \mid \_ \to \mathcal{E}\} & \text{(switch with } n \text{ non-default branches)} \\
\mid & \mathcal{I};\ \mathcal{E} & \text{(Instruction Sequence)}
\end{array}
$$

$$
\begin{array}{rll}
\mathcal{I} ::= & \texttt{call}\ f((x_{\text{in},1}, \ldots, x_{\text{in},n}), x_{\text{out}}) & \text{(function call)} \\
\mid & \texttt{let}_{\texttt{in}}\ x'_{\text{in}} = \texttt{freeze}(x_{\text{out}}) & \text{(freezing location)} \\
\mid & \texttt{let}_{\texttt{in}}\ x'_{\text{in}} = x_{\text{in}}.\widehat{\pi} & \text{(input sub-location)} \\
\mid & \texttt{let}_{\texttt{out}}\ x_{\text{out}} = \texttt{alloc}(\ell) & \text{(new } \ell\text{-bit output location)} \\
\mid & \texttt{let}_{\texttt{out}}\ x'_{\text{out}} = x_{\text{out}}.\widehat{\pi} & \text{(output sub-location)} \\
\mid & x_{\text{out}} := \text{rhs} & \text{(write to output location)} \\
\mid & \texttt{cast}\ x_{\text{out}}\ \texttt{to}\ \widehat{p} & \text{(refine the shape of a location)}
\end{array}
$$

$$
\begin{array}{rll}
\text{rhs} ::= & c & \text{(constant)} \\
\mid & x_{\text{in}} & \text{(input contents)} \\
\mid & \texttt{\&alloc}(\ell) & \text{(Allocation)}
\end{array}
$$

$$
\begin{array}{rll}
\mathcal{P} ::= & \texttt{let}\ f(X_{\text{in}}, x_{\text{out}}) = \mathcal{E};\ \mathcal{P} & \text{(function declaration)} \\
\mid & \mathcal{E} & \text{(Main)}
\end{array}
$$

Fig. 11. Target IR — Programs $\mathcal{P}$, Expressions $\mathcal{E}$ and Instructions $\mathcal{I}$.

version, in our IR, will be:

$$
\text{aux}(x_{\text{in}} : \widehat{\tau}_{\text{RISC-V}}, x_{\text{out}} : \widehat{\tau}_{\text{reg}}) = \boxed{\texttt{let}_{\texttt{in}}\ x = x_{\text{in}}.[7:5];\ x_{\text{out}} := x;\ \texttt{success}}
$$

$$
\text{get\_dest}(x_{\text{in}} : \widehat{\tau}_{\text{RISC-V}}, x_{\text{out}} : \widehat{\tau}_{\text{reg}}) = \boxed{\begin{array}{l}
\texttt{let}_{\texttt{in}}\ x = x_{\text{in}}.[0:7]; \\
\texttt{switch}\,(x) \left\{ \begin{array}{l}
\texttt{0x33} \to \texttt{call aux}(x_{\text{in}}, x_{\text{out}});\ \texttt{success} \\
\texttt{0x13} \to \texttt{call aux}(x_{\text{in}}, x_{\text{out}});\ \texttt{success} \\
\texttt{0x6F} \to \texttt{call aux}(x_{\text{in}}, x_{\text{out}});\ \texttt{success} \\
\texttt{0x23} \to x_{\text{out}} := 0;\ \texttt{success}
\end{array} \right\}
\end{array}}
$$

The compiled function get_dest considers its input x_{in} as a 32-bit word and focuses on its 7 lowest bits with a new *location* x_0. Depending on x_0's value, it either directly writes 0 to the destination x_{out} (it has recognised `sw`), or calls the auxiliary function aux, which outputs to destination x_{out} 5 bits corresponding to the destination register of instructions `add/addi/jal` (Fig. 1a, register `rd` is always located at the same place in the memory format). △

Our program representation thus makes the following tasks explicit:

- reading from locations in memory and switching on their values;
- writing results to their appropriate memory locations;
- allocating and initializing memory following the shape of the intended output value.

To this end, we depart from [Baudon et al. 2023] and define a new IR, described in Fig. 11, in Destination Passing Style [Shaikhha et al. 2017]. The essence of destination passing style is that each function takes a *destination* argument which indicates where it should write its result.

Similarly, in our IR, *memory locations*, usually denoted x, are identifiers for unaligned pointers. We distinguish between read-only *input locations*, ("x_{in}") and write-only *output locations*, ("x_{out}"). This distinction allows us to formally segregate "analysis" code (i.e., pattern matching) from "building" code (i.e., constructors). Destination passing style is thus immediately visible in function declarations and calls: the last argument of $\texttt{call}\ f((x_1, \ldots, x_n), y)$ is an output memory location that should be filled with the result computed by f. Input and output sub-locations are obtained by focusing an existing location with a memory path, for instance with the instruction $\texttt{let}_{\texttt{out}}\ x' = x.\widehat{\pi}$. Additionally, an output location can be *frozen* to get an input location with $\texttt{let}_{\texttt{in}}\ x_{\text{in}} = \texttt{freeze}(x_{\text{out}})$. Constructing values requires memory to write into: either by focusing an existing output location with a memory path ($\texttt{let}_{\texttt{out}}\ x' = x.\widehat{\pi}$), or by claiming a given amount of unused memory ($\texttt{let}_{\texttt{out}}\ x =$

Data: $x_1, \ldots, x_n$ the input locations
Data: $\widehat{\tau_1}, \ldots, \widehat{\tau_n} \in \widehat{Types}$ the input memory layouts
Data: $m = \{p_{i,1}, \ldots, p_{i,n} \to \mathcal{E}_i \mid 1 \leq i \leq N\}$ a matching problem whose branches map provenances to target expressions
Result: Target code corresponding to the decision DAG computed by Baudon et al. [2023]
1 **function** DESTRUCT $((x_1 : \widehat{\tau_1}, \ldots, x_n : \widehat{\tau_n}), m) : \mathcal{E}$

Algorithm 1: DESTRUCT interface.

$\texttt{alloc}(\ell)$). This memory is then filled using `write` instructions, with several kinds of contents: constants denoted c, the contents of an input location, or the address of newly allocated memory of a given size denoted $\texttt{\&alloc}(\ell)$.

Traditional control flow relies on the `switch` construct with a default branch marked by "_", along with `success` and `fail` return statements which do not return any value.

Finally, in the context of RIBBIT, switching on a location *reveals information about values*, notably details about their concrete shapes. To materialize this in the IR, we introduce *cast instructions* which refine the shape of a given output location to a more precise shape.

In the rest of this article, target expressions will be displayed with a plain frame around them. Let us finally point out that sharing is not explicit in the IR, even though we use a control-flow-graph style representation underneath.

Example 4.2 (Target IR on a program with lists). Using the memory type defined in Example 3.3, the following function captures and returns the first element of a list, 0 if empty.

$$\text{get\_value}(x : \tau \text{ as } \tau_p) : I_{32} = \text{match}(x) \left\{ \begin{array}{ll} \text{Nil} & \to 0_{(32)} \\ \text{Cons}(\_, \_) & \to x.Cons.0 \end{array} \right\}$$

will be compiled as:

$$\text{get\_value}(x_{\text{in}} : \widehat{\tau_p}, x_{\text{out}} : I_{32}) = \boxed{\begin{array}{l} \texttt{let}_{\texttt{in}}\ x = x_{\text{in}}.[0:2]; \\ \texttt{switch}\ (x) \left\{ \begin{array}{ll} 0 & \to x_{\text{out}} := 0;\ \texttt{success} \\ 1 & \to \texttt{let}_{\texttt{in}}\ x' = x_{\text{in}}.[2:32];\ x_{\text{out}} := x';\ \texttt{success} \\ 2 & \to \texttt{let}_{\texttt{in}}\ x' = x_{\text{in}}.*.0;\ x_{\text{out}} := x';\ \texttt{success} \end{array} \right\} \end{array}} \triangle$$

5 COMPILATION OF DESTRUCTORS: A PRIMER

Looking at the compilation of our expression language described in Section 3.3, one important task is to compile pattern matching. This is a well-studied topic [Maranget 2008; Sestoft 1996] with several general procedures which take as input a list of patterns – usually with no variables nor right-hand-side expressions – and produce a nest of "switch" nodes, either following an automaton or a DAG. In the context of customizable memory layouts like the ones we consider, Baudon et al. [2023] provides a layout-aware compilation procedure emitting decision DAGs. As our goal is to consider such a compilation procedure in a full language, we will reuse their compilation algorithm, which already accommodates nested provenances.

The DESTRUCT algorithm, whose interface is described in Algorithm 1, takes as argument an input location to inspect, its memory type, and a matching problem. It emits code that inspects the *memory representation* of the value at the given memory location, to determine which provenance in the list, if any, matches the *original high-level value*. As a side effect, it also copies the right-hand side expressions of the pattern matching problem.

Example 5.1 (Matching Compilation for lists). We consider the "matching part" of the function defined in Example 4.2, on which we apply Destruct:

$$\textsc{Destruct}((x : \widehat{\tau}_p), \begin{Bmatrix} \text{Nil} & \Rightarrow \mathcal{E}_1 \\ \text{Cons}(\_,\_) & \Rightarrow \mathcal{E}_2 \end{Bmatrix}) = \boxed{\texttt{let}_{\texttt{in}}\ x' = x.[0:2];\ \texttt{switch}\,(x') \begin{Bmatrix} 0 \to \mathcal{E}_1 \\ 1 \to \mathcal{E}_2 \\ 2 \to \mathcal{E}_2 \end{Bmatrix}} \qquad \triangle$$

Some remarks about our adaptation. We adapt the original algorithm in the following ways. The original algorithm emits decision trees, whose translation to our target IR is straightforward (it is sufficient to bind intermediate input locations for switches). Destruct takes a list of provenances with their target expressions $\mathcal{E}$, which are placed at the leaves of the decision tree. The original algorithm by Baudon et al. [2023] attempts to handle binders in patterns, albeit in an incomplete way. We ignore this aspect, and will instead handle memory accesses in a more general way.

6 COMPILATION OF PATHS: A FIRST ATTEMPT

Considering compilation of our input language, another thorny construction remains: valuexpressions, which combine constructors, variables, and path-based accessors. We now make a first (unsuccessful, as we will see) attempt at a simple compilation procedure for path-based accessors, and take this opportunity to introduce some tooling required for the full compilation procedure.

6.1 Exploring Layouts with Focus and Specialise

To compile high-level expressions in a way that fits a given memory layout, we need a way to explore both a type and its layout conjointly. Indeed, the full inner structure is only revealed when considering both the type, which defines nested terms and subterms, and the layout, which describes the exact operations required to access those subterms, represented as fragments. Let us consider a simple example demonstrating this idea.

Example 6.1 (Branch-dependent layout on the list example). Following our list example from Example 3.3, consider the following pattern matching branch: $\text{Cons}(\_,\_) \to \ldots\ x.\text{Cons.0}\ \ldots$.

How to compile the accessor $x.\text{Cons.0}$ to a memory access? This requires locating the associated fragment in the memory layout $\widehat{\tau}_p$. However, the location of this fragment depends on other values in memory because of a split in $\widehat{\tau}_p$. We thus need to *explore* several possibilities, indexed by various provenances of x. For instance, if x matches $\text{Cons}(\_, \text{Nil})$, it will be at position $x.[2:32]$ in memory. If x matches $\text{Cons}(\_, \text{Cons}(\_,\_))$ however, it will be at position $x.*.0$ in memory. This distinction between provenances will appear in our generated code, as shown in Example 4.2. $\triangle$

The **Explore** function defined in this section takes as input an already-known provenance p, a type τ and a layout $\widehat{\tau}$, and returns the list of *branches*, i.e. possible versions of τ represented as $\widehat{\tau}$ which are "compatible" with p. Semantically, a branch characterises the parts of τ that match the provenance p. A branch is thus defined as quadruplet $(p_i, \tau_i, \widehat{\tau}_i, \text{frags}_i)$ consisting of the provenance, type and layout refined for that specific branch, and of a list of fragments contained therein.

Example 6.2 (Explore the list type (Example 3.3)). We can explore $\widehat{\tau}_p$ with the provenance $\text{Cons}(\_,\_)$:

$$\textbf{Explore}(\text{Cons}(\_,\_), \tau_{\text{list}}, \widehat{\tau}_p) = \begin{Bmatrix} (\text{Cons}(\_, \text{Nil}) & , \text{Cons}(I_{32}, \text{Nil}) & , \widehat{\tau}_{C1}, F_1) \\ (\text{Cons}(\_, \text{Cons}(\_,\_)) & , \text{Cons}(I_{32}, \text{Cons}(I_{32}, \tau_{list})) & , \widehat{\tau}_{C2}, F_2) \end{Bmatrix}$$

where $F_1 = \left[\ .[2:32] \mapsto (.\text{Cons.0 as } I_{32})\ \right]$ and $F_2 = \begin{bmatrix} .*.0 \mapsto (.\text{Cons.0 as } I_{32}) \\ .*.1 \mapsto (.\text{Cons.1.Cons.0 as } I_{32}) \\ .*.2 \mapsto (.\text{Cons.1.Cons.1 as } \widehat{\tau}_p) \end{bmatrix}$.

Note that the types and layouts of branches are *refined* according to their provenance. $\triangle$

To precisely define **Explore**, we need a new operation to refine layouts: specialisation.

1 **function Explore**($p_0 \in \mathit{Provs}, \tau_0 \in \mathit{Types}, \widehat{\tau_0} \in \widehat{\mathit{Types}}$):
2 **for** $(p, \widehat{\tau}) \in \widehat{\tau_0} \,/\, p_0$ **do**
3 $\tau \leftarrow \tau_0 \,/\, p$
4 frags $\leftarrow \left\{ (\widehat{\pi} \mapsto \pi \text{ as } \widehat{\tau'}) \;\middle|\; \widehat{\textbf{focus}}\ (\widehat{\pi}, \widehat{\tau}) = (\pi \text{ as } \widehat{\tau'}) \right\}$
5 **yield** $(p, \tau, \widehat{\tau}, \mathit{frags})$

Algorithm 2: Explore.

Layout Specialisation. Similar to type specialisation, layout specialisation, denoted $\widehat{\tau} \,/\, p$, filters a layout $\widehat{\tau}$ to exclude parts which are incompatible with a given provenance p. It removes all splits from $\widehat{\tau}$ (up to fragments) by filtering out branches whose provenance set excludes p. It returns a list of pairs of the form $(p', \widehat{\tau'})$, where p' is a refined version of p and $\widehat{\tau'}$ is the restriction of $\widehat{\tau}$ to values that match p'. We demonstrate it on an example, the full definition is in Appendix A, Fig. 14.

Example 6.3 (Layout specialisation). For instance:

$$\widehat{\tau_p} \,/\, \mathrm{Cons}(\_, \_) = \left\{ \begin{array}{r} \mathrm{Cons}(\_, \mathrm{Nil}),\ {}_{\_64} \ltimes [0:2] = 1 \ltimes [2:32] : (.\mathrm{Cons}.0 \text{ as } I_{32}) \\ (\mathrm{Cons}(\_, \mathrm{Cons}(\_, \_)),\ \&_{64}\,(\dots) \ltimes [0:2] = 2 \end{array} \right\} \qquad \triangle$$

Explore. We are now ready to properly define **Explore**, in Algorithm 2. Given an initial provenance p_0, type τ_0 and layout $\widehat{\tau_0}$, it returns the list of all branches of $\widehat{\tau_0}$ that represent τ_0 values matching p_0. **Explore**, and many of the algorithms described in this article, uses Python-style generators using the "**yield**" keyword, and "for-each" style loops.

Using the specialisation $\widehat{\tau_0} \,/\, p_0$, we gather all refinements pairs of $\widehat{\tau_0}$ compatible with p_0. Each pair consists of a more precise provenance p and a specialised layout $\widehat{\tau}$. We then derive all information relevant to this case from p and $\widehat{\tau}$: the refined type $\tau_0 \,/\, p$, and a list of the form $(\widehat{\pi'} \mapsto \pi' \text{ as } \widehat{\tau'})$ containing every position $\widehat{\pi'}$ such that $\widehat{\tau}$ contains a fragment at $\widehat{\pi'}$. From these results, we construct a branch.

6.2 A Naive Compilation Algorithm for Accessors

An intuitive idea to compile accessors would be to find "the right memory path" corresponding to a given high-level path. It would proceed as follows: given a variable x, its type τ, layout $\widehat{\tau}$, and a path-based accessor $x.\pi$, we can *explore* all branches of $\widehat{\tau}$ and for each possible layout, return the memory path that leads to a fragment corresponding to the high-level path π.

The Extract procedure, defined in Algorithm 3, implements this idea. It takes a description $(x, \tau, \widehat{\tau}, p \triangleleft f : \text{as o})$ the input value, an output location y and a path π. It emits code to store in y the representation of the subterm located at π within the input value. Here, we are looking for "the right memory path", as such, there should exist a fragment within $\widehat{\tau}$ that exactly corresponds to the (high-level) subterm π. All we have to do now is to find *where*.

- In the base case (Line 2), if $\pi = \epsilon$, then we simply copy input contents into the output location.
- As we have seen, the location of the fragment might depend on other parts of the layout, hence the need for **Explore**. For this purpose, we gather all branches (Line 5) and generate code that dynamically determines the appropriate branch by inspecting the input value. We can readily reuse the pattern matching compilation algorithm Destruct for this purpose (Line 12).
- For each branch of the input layout, we round up all fragments that appear in its layout and their positions, then focus into the one that represents a parent of the target subterm ($\pi_f \preceq \pi$) (Line 8) to recursively look for π (Line 9).
- If no such fragment exists, we fail.

*Example 6.4 (*Extract *on the list example– Algorithm 3).* Let us consider again the first integer element of a (non empty) list, at position .Cons.0. Let x_{out} a fresh output location and x an input location

Data: $(x \triangleleft p : \tau \text{ as } \widehat{\tau})$ the input description
Data: y the output location
Data: π the path in the input to the desired value
Result: Code binding y to the representation of the subterm at position π in the input

function EXTRACT($(x \triangleleft p : \tau \text{ as } \widehat{\tau}), y, \pi$):
  if $\pi = \epsilon$ **then**
    return $\boxed{y := x;\ \texttt{success}}$
  else
    $B \leftarrow$ **for** $p_b, \tau_b, \widehat{\tau}_b, frags_b \in$ **Explore**$(p, \tau, \widehat{\tau})$ **do**
      if $\exists(\widehat{\pi}_f \mapsto \pi_f \text{ as } \widehat{\tau}_f) \in frags_b, \exists \pi', \pi = \pi_f.\pi'$ **then**
        $x' \leftarrow$ fresh symbol
        $\tau', p' \leftarrow$ **focus** (π_f, τ_b) , **focus** (π_f, p_b)
        $\mathcal{E} \leftarrow \boxed{\texttt{let}_{\texttt{in}}\ x' = x.\widehat{\pi}_f;\ \text{EXTRACT}((x' \triangleleft p' : \tau' \text{ as } \widehat{\tau}_f), y, \pi')}$
      else $\mathcal{E} \leftarrow \boxed{\texttt{fail}}$
      yield $(p_b, \mathcal{E})$
    return $\boxed{\text{DESTRUCT}\ (x, \widehat{\tau}, B)}$

Algorithm 3: EXTRACT: Naive Path compilation algorithm.

assumed to contain the representation of a Cons. EXTRACT $\big((x \triangleleft \text{Cons}(\_, \_) : \tau_{\text{list}} \text{ as } \widehat{\tau}_p), x_{\text{out}}, .\text{Cons}.0\big)$ starts by exploring $\widehat{\tau}_p$. Following Example 6.2, we have two branches:

- The first branch is $(\text{Cons}(\_, \text{Nil}), \tau_{C1}, \widehat{\tau}_{C1}, F_1)$ where $F_1 = \big[\, .[2:32] \mapsto (.\text{Cons}.0 \text{ as } I_{32}) \,\big]$. The fragment at $.[2:32]$ covers .Cons.0. We can now focus on it and proceed to the (first) recursive call. Let x' a fresh symbol. By focusing, we obtain the recursive arguments $\tau' =$ **focus** $(.\text{Cons}.0, \tau_{C1}) = I_{32}$, $p' = \_$ and $\pi' = \epsilon$, leading to EXTRACT $((x' \triangleleft p' : \tau' \text{ as } I_{32}), x_{\text{out}}, \pi') = \boxed{x_{\text{out}} := x';\ \texttt{success}}$.
  We finally obtain $\mathcal{E}_1 = \boxed{\texttt{let}_{\texttt{in}}\ x' = x.[2:32];\ y := x';\ \texttt{success}}$ and yield $(\text{Cons}(\_, \text{Nil}), \mathcal{E}_1)$.
- The second branch is $(\text{Cons}(\_, \text{Cons}(\_, \_)), \tau_{C2}, \widehat{\tau}_{C2}, F_2)$ with $(. * .0 \mapsto .\text{Cons}.0 \text{ as } I_{32}) \in F_2$. We proceed similarly, with a recursive call to get a target expression $\mathcal{E}_2$, yielding the pair $(\text{Cons}(\_, \text{Cons}(\_, \_)), \mathcal{E}_2)$.

  We finally emit the following code, which is coherent with Example 4.2:

$$\text{DESTRUCT}((x_0 : \widehat{\tau}_{\text{list}}), \{(\text{Cons}(\_, \text{Nil}) \mapsto \mathcal{E}_1), (\text{Cons}(\_, \text{Cons}(\_, \_)) \mapsto \mathcal{E}_2)\}) =$$

$$\boxed{\texttt{let}_{\texttt{in}}\ x = x_{\text{in}}.[0:2];\ \texttt{switch}\ (x) \left\{ \begin{array}{l} 0 \longrightarrow x_{\text{out}} := 0;\ \texttt{success} \\ 1 \longrightarrow \texttt{let}_{\texttt{in}}\ x' = x_{\text{in}}.[2:32];\ x_{\text{out}} := x';\ \texttt{success} \\ 2 \longrightarrow \texttt{let}_{\texttt{in}}\ x' = x_{\text{in}}. * .0;\ x_{\text{out}} := x';\ \texttt{success} \end{array} \right\}} \qquad \triangle$$

To implement EXTRACT, we assumed that, once we have narrowed down the layout to a single branch, any given source path can be translated to a single memory path. While this is true for "simple" layouts, it is not the case in general. For instance, the immediate at .Sw.2 in the RISC-V layout (Fig. 1d and Example 3.2) is broken into two pieces, which need to be assembled to get a whole integer. Similarly, the arithmetic expression (Fig. 5a) at position .Plus.1 in the case Plus(_, I(_)) doesn't even exist in memory anymore as only the inner integer is stored: we need to rebox it to get an expression. Finally, in our list example (Example 3.3), if we had to implement *tail*, we would have to reconstruct a new list from Cons.1, which does not exist either: Nil is implicit in the case Cons(_, Nil) and the tail list is broken up in the pointed struct in the case Cons(_, Cons(_, _)).

Nevertheless, the Extract procedure demonstrates how focusing, specialisation and **Explore** can be combined to design fairly complex compilation procedures. In fact, Extract already subsumes the subset of binders handled by Baudon et al. [2023], in only 12 lines of code. We can now use these tools to develop a full compilation procedure for our language.

7 FULL COMPILATION

We are, at last, ready to present our main contribution: a general procedure, implemented into Ribbit, which can compile both pattern matching and value-building expressions in a unified manner for arbitrary memory layouts. As we have seen, in the presence of complex layouts, it is necessary to destruct and reconstruct values, making it necessary to treat compilation of accessors and constructors conjointly. Before going any further, let us look at the toplevel compilation process for full expressions.

7.1 Compilation of Expressions

Our main compilation algorithm is presented in Algorithm 4. It takes as input a collection of binders Γ, an output triplet $(x_{\text{out}} : \tau_{\text{out}} \text{ as } \widehat{\tau}_{\text{out}})$, and an expression e. Γ contains binders of the form $(x \triangleleft p : \tau \text{ as } \widehat{\tau})$ indicating that x is an available input memory location which represents a value of type τ with the layout $\widehat{\tau}$ and which is known to match the provenance p. In each case, Compile delegates emission of code for given tasks to several sub-procedures. First off, compilation of pattern matching is immediately delegated to Destruct, as previously described in Section 5. The other procedures are described in the upcoming subsections, and comprise the following tasks:

- Seek (Section 7.2) a path π within the memory representation of its parent input value x to produce a piece of data corresponding to the accessor $x.\pi$. Seek extends Extract to handle arbitrary memory layout combinations, including those where the desired subterm is broken into multiple pieces scattered over the parent representation.
- Assemble these pieces to Rebuild (Section 7.3) the representation of the valuexpression u.
- Refine (Section 7.4) the contents of a memory location to allocate all necessary pieces and set constants.
- Finally, Section 7.6 describes how to ensure termination of our algorithm and emit recursive code when necessary.

7.2 Seek a path

Seek, in Algorithm 5, compiles an accessor $x.\pi$ to target code. We consider the memory representation of the value bound to x (which is already built and read-only at execution time) as the *input value* and identify it with the quadruple $(x_{\text{in}} \triangleleft p_{\text{in}} : \tau_{\text{in}} \text{ as } \widehat{\tau}_{\text{in}})$, composed of its input location, type, layout and provenance respectively. That is, at execution time, x_{in} is expected to contain the memory representation according to $\widehat{\tau}_{\text{in}}$ of some value of type τ_{in} matching the provenance p_{in}. Similarly, we refer to the memory representation of the desired subterm (which is the piece of data currently being built) as the *output value* and identify it with the triple $(x_{\text{out}} : \tau_{\text{out}} \text{ as } \widehat{\tau}_{\text{out}})$ composed of its output location, type and layout respectively. Our goal is to emit code that, given an input location x_{in} containing the memory representation of v following the layout $\widehat{\tau}_{\text{in}}$ and an output location x_{out}, stores in x_{out} the representation of **focus** (π, v) following the layout $\widehat{\tau}_{\text{out}}$.

Seek is similar to the Extract procedure presented in Section 6.2. The differences are highlighted in orange. As we have seen, there are some cases where there is no single fragment that covers the desired path. Alternatively, we might reach $\pi = \epsilon$, but $\widehat{\tau}_{\text{in}} \neq \widehat{\tau}_{\text{out}}$, indicating that we should exhibit an isomorphism between $\widehat{\tau}_{\text{in}}$ and $\widehat{\tau}_{\text{out}}$ to get the desired value. In both cases, we need to

Data: $\Gamma = (x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau_i})_{0 \le i < n}$ a set of input descriptions
Data: e the input expression
Data: $(x_{\text{out}} : \tau_{\text{out}} \text{ as } \widehat{\tau}_{\text{out}})$ the output description
Result: Code computing the value of the expression e
function Compile(Γ, $(x_{\text{out}} : \tau_{\text{out}} \text{ as } \widehat{\tau}_{\text{out}})$, e) = **case**(e){

match(y) {
  $p_i \rightarrow e_i \mid (0 \le i < n)$
}
$\longrightarrow$
// Delegate pattern matching compilation to Destruct (Section 5)
$(y \triangleleft p : \tau \text{ as } \widehat{\tau}) \in \Gamma$
$\forall i. \mathcal{E}_i \leftarrow$ Compile(Γ, $(x_{\text{out}} : \tau_{\text{out}} \text{ as } \widehat{\tau}_{\text{out}})$, e_i)
Destruct $((y : \widehat{\tau}), \{p_i \rightarrow \mathcal{E}_i \mid 1 \le i \le n\})$

$y.\pi$ $\longrightarrow$ Seek $(\Gamma(y), (x_{\text{out}} : \tau_{\text{out}} \text{ as } \widehat{\tau}_{\text{out}}), \pi)$ // Seek the path π in y (Section 7.2)

u $\longrightarrow$ Rebuild $(\Gamma, (x_{\text{out}} : \tau_{\text{out}} \text{ as } \widehat{\tau}_{\text{out}}), u)$ // Rebuild constructor u (Section 7.3)

let $y : \tau$ as $\widehat{\tau} = e_1$ in e_2 $\longrightarrow$
y_{out} fresh.
// Allocate new memory location for e_1 with Refine (Section 7.4)
$\texttt{let}_{\texttt{out}}\ y_{\text{out}} = \texttt{alloc}(|\widehat{\tau}|)$; Refine$(y_{\text{out}}, _{|\widehat{\tau}|}, \textbf{shape_of}(\widehat{\tau}))$;
Compile(Γ, $(y_{\text{out}} : \tau \text{ as } \widehat{\tau}), e_1$) // Emit e_1
$\texttt{let}_{\texttt{in}}\ y = \texttt{freeze}(y_{\text{out}})$;
Compile($\Gamma \cup (y \triangleleft \_ : \tau \text{ as } \widehat{\tau}), (x_{\text{out}} : \tau_{\text{out}} \text{ as } \widehat{\tau}_{\text{out}}), e_2$) // Emit e_2

}

Algorithm 4: Compile – Compile the expression e given the environment Γ.

break up the value even more in order to reconstruct it differently. This task is precisely done on Line 11 via our next procedure: Rebuild.

7.3 Rebuild constructors

Rebuild (Algorithm 6) compiles an arbitrary valuexpression u to target code that constructs it. Whereas Seek retrieves a (possibly mangled) relevant fragment from a single input value, Rebuild inspects the shape of the output value to assemble its constituent pieces, which include constant parts of the memory layout as well as fragments extracted from multiple input values. It reuses the general structure of Seek (and Extract), but considers n input values described by quadruples $(x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau_i})$ with $1 \le i \le n$.

Firstly, when u is a constant integer and the output layout is a fixed-width integer encoding, we simply write this constant to the output location (Line 3). In other cases, we operate in a similar manner as in other procedures: we **Explore** the various branches (Line 6), whose code we will eventually combine with Destruct (Line 18).

A first subtlety appears: both algorithms need to maintain a precise description of the current case under scrutiny, and to distinguish between all possible subcases. Indeed, after a few recursive calls, we might be exploring deep into input and output layouts. This is done by maintaining the already determined provenances p_{in} through recursive calls. This is essential to emit code that respects invariants stemming from previous branch choices. For the output, however, we do not have a ready-made provenance. We obtain one in Line 5 via the **prov_of** function, which is a simple syntactic translation from expressions to provenances. We can then use the obtained provenance p_{out} in **Explore** to only obtain branches that are relevant to u.

Conversely, while exploring a branch characterised by p_b, we must perform a similar maneuver in the opposite direction. Indeed, at execution time, the only way to determine the right branch of $\widehat{\tau}_{\text{out}}$ is to examine input values. By definition, the only variable parts of p_{out} are those at the

Data: $(x_{\text{in}} \triangleleft p_{\text{in}} : \tau_{\text{in}} \text{ as } \widehat{\tau}_{\text{in}})$ the input description
Data: $(x_{\text{out}} : \tau_{\text{out}} \text{ as } \widehat{\tau}_{\text{out}})$ the output description
Data: π the path in the input to the desired subterm
Result: Code binding the memory value at position π in the input

1 **function** Seek$((x_{in} \triangleleft p_{in} : \tau_{in} \text{ as } \widehat{\tau}_{in}), (x_{out} : \tau_{out} \text{ as } \widehat{\tau}_{out}), \pi)$**:**
  // Invariant: π and p_{in} are compatible
2 **if** $\pi = \epsilon \wedge \widehat{\tau}_{in} = \widehat{\tau}_{out}$ **then** // Input and output representations are the same, we return.
3 **return** $x_{out} := x_{in}$; success
4 **else** // Otherwise, Explore all cases.
5 $B \leftarrow$ **for** $p_b, \tau_b, \widehat{\tau}_b, frags_b \in$ **Explore**$(p_{in}, \tau_{in}, \widehat{\tau}_{in})$ **do**
  // Seek a fragment containing the piece of data at π.
6 **if** $\exists(\widehat{\pi}_f \mapsto \pi_f \text{ as } \widehat{\tau}_f) \in frags_b, \exists \pi', \pi = \pi_f.\pi'$ **then** // Found one. Focus and search inside.
7 $x' \leftarrow$ fresh symbol
8 $\tau', p' \leftarrow$ **focus** (π_f, τ_b), **focus** (π_f, p_b)
9 $\mathcal{E} \leftarrow$ let$_{\text{in}}$ $x' = x.\widehat{\pi}_f$; Seek $\left((x' \triangleleft p' : \tau' \text{ as } \widehat{\tau}_f), (x_{\text{out}} : \tau_{\text{out}} \text{ as } \widehat{\tau}_{\text{out}}), \pi'\right)$
10 **else** // If none exists, Rebuild from smaller pieces.
11 $\mathcal{E} \leftarrow$ Rebuild$((x_{\text{in}} \triangleleft p_{\text{in}} : \tau_{\text{in}} \text{ as } \widehat{\tau}_{\text{in}}), (x_{\text{out}} : \tau_{\text{out}} \text{ as } \widehat{\tau}_{\text{out}}), x_{\text{in}}.\pi)$
12 **yield** $(p_b, \mathcal{E})$
13 **return** Destruct $((x_{in} : \widehat{\tau}_{in}), B)$ // Assemble the code of these branches via a decision tree.

Algorithm 5: Seek – Seek the memory location representing π in x_{in}.

positions of accessors in u: there is a bijection between the possible cases of p_{out} and the possible combinations of p_i subcases. At Line 8, we derive a new provenance p_i' for each input, which is at least as precise as p_i and also integrates information from the considered output subcase p_b. We use these refined input provenances to emit code that inspects the n input values to determine the right branch of $\widehat{\tau}_{\text{out}}$, at Line 18.

For each branch, we need two categories of code: how to define the constant part of the value, and how to fill variable parts depending on input arguments. Let us first focus on variable parts. Each variable part exactly corresponds to a fragment. On Line 12, we look for a fragment $(\pi_f \text{ as } \widehat{\tau}_f)$ available in the current branch. If there is an accessor $x.\pi$ linked to an input readily available at a corresponding position in u, it is sufficient to Seek our desired piece in x (Line 13). If the fragment corresponds to no existing input argument, we need to break it down further, by calling Rebuild again (Line 16). While we have found all variable pieces, we have not yet allocated the surrounding memory in which to assemble them! This is done Line 7 via the Refine procedure.

7.4 Refine memory locations

So far, we implicitely assumed that the code emitted by a Seek or Rebuild call would always be executed in a context where the output location x_{out} contains a memory value of the same shape as $\widehat{\tau}_{\text{out}}$. This section focuses on the task of emitting allocation and cast instructions to maintain this invariant. We must ensure that whenever the output shape becomes more precise – by replacing a split or fragment in $\widehat{\tau}_{\text{out}}$ with a concrete layout – the shape of x_{out} is *refined* accordingly by allocating, casting and initializing memory at this position to match the new shape.

The intuition behind our shape refinement procedure is as follows: for both Seek and Rebuild, we know that the contents of the output location x_{out} conform to the shape **shape_of**$(\widehat{\tau}_{\text{out}})$. This shape might be *opaque* (of the form $\__\ell$), for instance if $\widehat{\tau}_{\text{out}}$ is a split. In Rebuild, we **Explore** the

Data: $(x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i)_{1 \le i \le n}$ the n input descriptions
Data: $(x_{\text{out}} : \tau_{\text{out}} \text{ as } \widehat{\tau}_{\text{out}})$ the output description
Data: u the desired valuexpr
Result: Code constructing the memory value corresponding to expression u

1 **function** Rebuild$((x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i)_{1 \le i \le n}, (x_{out} : \tau_{out} \text{ as } \widehat{\tau}_{out}), u)$**:**
2 **if** $u = c \wedge \widehat{\tau}_{out} = I_\ell$ **then** // Target value is a constant encoded in an immediate type.
3 **return** $\boxed{x_{out} := c;\ \texttt{success}}$
4 **else** // Otherwise, Explore all cases.
5 $p_{\text{out}} \leftarrow$ **prov_of**(τ_{out}, u)
6 $B \leftarrow$ **for** $p_b, \tau_b, \widehat{\tau}_b, frags_b \in$ **Explore**$(p_{out}, \tau_{out}, \widehat{\tau}_{out})$ **do**
 // Allocate memory, cast and fill in constant parts of the target memory layout to fit its shape.
7 $\mathcal{E}_{\text{consts}} \leftarrow \boxed{\text{Refine}\ (x_{\text{out}}, \textbf{shape\_of}(\widehat{\tau}_{\text{out}}), \textbf{shape\_of}(\widehat{\tau}_b))}$
 // Rebuild target fragments from input values, which we specialize for the current branch.
8 $\forall i. p'_i \leftarrow p_i\ [\_ \text{ at } \pi_{\text{in}}.\pi \leftarrow \textbf{focus}\ (\pi_{\text{out}}.\pi, p_b) \mid \textbf{focus}\ (\pi_{\text{out}}, u) = x_i.\pi_{\text{in}}]$
9 $\mathcal{E}_{\text{frags}} \leftarrow$ **for** $(\widehat{\pi}_f \mapsto \pi_f \text{ as } \widehat{\tau}_f) \in frags_b$ **do**
10 $x'_{\text{out}} \leftarrow$ fresh symbol
11 $\tau_f \leftarrow \textbf{focus}\ (\pi_f, \tau_b)$
 // Look at the subterm containing π_f in u
12 **if** $\exists i, \pi, \pi_{in}, \pi_{out}. \textbf{\textit{focus}}\ (\pi_{out}, u) = x_i.\pi_{in} \wedge \pi_{out}.\pi = \pi_f$ **then** // Found an input path $x_i.\pi_{\text{in}}.\pi$
13 **yield** $\boxed{\texttt{let}_{\texttt{out}}\ x'_{out} = x_{out}.\widehat{\pi}_f;\ \text{Seek}\ ((x_i \triangleleft p'_i : \tau_i \text{ as } \widehat{\tau}_i), (x'_{out} : \tau_f \text{ as } \widehat{\tau}_f), \pi_{in}.\pi)}$
14 **else** // Otherwise, break it down further.
15 $u' \leftarrow \textbf{focus}\ (\pi_f, u)$
16 **yield** $\boxed{\texttt{let}_{\texttt{out}}\ x'_{out} = x_{out}.\widehat{\pi}_f;\ \text{Rebuild}\ ((x_i \triangleleft p'_i : \tau_i \text{ as } \widehat{\tau}_i)_{1 \le i \le n}, (x'_{out} : \tau_f \text{ as } \widehat{\tau}_f), u')}$
17 **yield** $\left((x_i \triangleleft p'_i)_{1 \le i \le n}, \boxed{\mathcal{E}_{consts}; \mathcal{E}_{frags}}\right)$
18 **return** $\boxed{\text{Destruct}\ ((x_i : \widehat{\tau}_i)_{1 \le i \le n}, B)}$ // Assemble these branches into a decision tree.

Algorithm 6: Rebuild – Rebuild the value in memory representing u from the x_is.

output type and layout, yielding a more precise layout $\widehat{\tau}_b$. This is precisely the time at which we might need to allocate: if we discover that $\widehat{\tau}_b$ contains a pointer to new structures, we should allocate them for future use. Refine creates a skeleton that will be filled later on. This can happens either while rebuilding values, on Line 7 in Algorithm 6, or while compiling let-bindings, in Algorithm 4.

This leads us to define Refine (Algorithm 7), which takes an output location x, an imprecise shape $\widehat{p}_{\text{old}}$ and a more precise shape $\widehat{p}_{\text{new}}$, and emits a piece of code that allocates memory and initializes constants everywhere $\widehat{p}_{\text{new}}$ is more precise than $\widehat{p}_{\text{old}}$. For instance, when discovering a new pointer, we allocate a new address to fill x. Furthermore, we explore recursively the shapes to fill out all the pieces, for instance in the case of structs. Note that refining composite words $\widehat{p} \bowtie_{0 \le i < n} [o_i : \ell_i] : \widehat{p}'_i$ takes peculiar care: indeed, $\widehat{p}$ could yield a new pointer, which should be masked appropriately to not overwrite values obtained by the right hand sides. For this purpose, we craft a new mask using the $\textbf{mask}_\ell$ functions to generate ℓ-bit masks that precisely focus on the leftmost/base layout of a composite word.

Note that our refinement procedure is directly linked to the precision of the **shape_of** function (Fig. 10): a more precise shape analysis, for instance that inspects inside pointers, would yield earlier, easily factorised, memory allocations.

Data: x an input location
Data: $\widehat{p}_{\text{old}}$ the input provenance and $\widehat{p}_{\text{new}}$ the refined provenance
Result: Code to allocate and initialise the skeleton of x
function $\textsc{Refine}(x, \widehat{p}_{\text{old}}, \widehat{p}_{\text{new}}) = \textbf{case}(\widehat{p}_{\text{old}}, \widehat{p}_{\text{new}})$ {

$\widehat{p}$	, $\widehat{p}$	$\longrightarrow$	$\texttt{success}$
$\__\ell$	, $(c)_\ell$	$\longrightarrow$	$x := c;\ \texttt{success}$
$\__\ell$	, $\&_\ell\ (\widehat{p})$	$\longrightarrow$	$x := \texttt{\&alloc}(\lvert\widehat{p}\rvert);\ \textsc{Refine}(x, \&_\ell\ (\__{\lvert\widehat{p}\rvert}), \&_\ell\ (\widehat{p}))$
$\__{\lvert\widehat{p}\rvert}$	, $\widehat{p} \bowtie_{0\le i<n} [o_i : \ell_i] : \widehat{p}_i$	$\longrightarrow$	$\texttt{cast } x \texttt{ to } \__{\lvert\widehat{p}\rvert} \bowtie_{0\le i<n} [o_i : \ell_i] : \__{\ell_i};$ $\textsc{Refine}\left(x, \__{\lvert\widehat{p}\rvert} \bowtie_{0\le i<n} [o_i : \ell_i] : \__{\ell_i}, \widehat{p} \bowtie_{0\le i<n} [o_i : \ell_i] : \widehat{p}_i\right)$
$\__{\lvert\widehat{p}_0\rvert+\cdots+\lvert\widehat{p}_{n-1}\rvert}$	, $\{\!\{\widehat{p}_0, \ldots, \widehat{p}_{n-1}\}\!\}$	$\longrightarrow$	$\texttt{cast } x \texttt{ to } \{\!\{\__{\lvert\widehat{p}_0\rvert}, \ldots, \__{\lvert\widehat{p}_{n-1}\rvert}\}\!\};$ $\textsc{Refine}\left(x, \{\!\{\__{\lvert\widehat{p}_0\rvert}, \ldots, \__{\lvert\widehat{p}_{n-1}\rvert}\}\!\}, \{\!\{\widehat{p}_0, \ldots, \widehat{p}_{n-1}\}\!\}\right)$
$\&_\ell\ (\widehat{p})$	, $\&_\ell\ (\widehat{p}')$	$\longrightarrow$	$\texttt{let}_{\texttt{out}}\ x' = x.*;\ \textsc{Refine}(x', \widehat{p}, \widehat{p}')$
$\widehat{p} \bowtie_{0\le i<n} [o_i : \ell_i] : \widehat{p}_i$	, $\widehat{p}' \bowtie_{0\le i<n} [o_i : \ell_i] : \widehat{p}'_i$	$\longrightarrow$	$\texttt{let}_{\texttt{out}}\ x' = x.\textbf{mask}_{\lvert\widehat{p}\rvert}([o_0 : \ell_0], \ldots, [o_{n-1} : \ell_{n-1}]);$ $\textsc{Refine}(x', \widehat{p}, \widehat{p}');$ $\texttt{let}_{\texttt{out}}\ x_0 = x.[o_0 : \ell_0];\ \textsc{Refine}(x_0, \widehat{p}_0, \widehat{p}'_0);\ \ldots$ $\texttt{let}_{\texttt{out}}\ x_{n-1} = x.[o_{n-1} : \ell_{n-1}];\ \textsc{Refine}(x_{n-1}, \widehat{p}_{n-1}, \widehat{p}'_{n-1})$
$\{\!\{\widehat{p}_0, \ldots, \widehat{p}_{n-1}\}\!\}$	, $\{\!\{\widehat{p}'_0, \ldots, \widehat{p}'_{n-1}\}\!\}$	$\longrightarrow$	$\texttt{let}_{\texttt{out}}\ x_0 = x.0;\ \textsc{Refine}(x_0, \widehat{p}_0, \widehat{p}'_0);\ \ldots$ $\texttt{let}_{\texttt{out}}\ x_{n-1} = x.(n-1);\ \textsc{Refine}(x_{n-1}, \widehat{p}_{n-1}, \widehat{p}'_{n-1})$

}

Algorithm 7: $\textsc{Refine}(x, \widehat{p}_{\text{old}}, \widehat{p}_{\text{new}})$ – Memory shape refinement instructions.

7.5 Example of compilation

Let us now consider the following function on lists, in particular its Cons branch:

$$\text{incr\_head}(x) = \text{match}(x) \left\{ \begin{array}{ll} \text{Nil} & \longrightarrow \text{Nil} \\ \text{Cons}(\_, \_) & \longrightarrow \text{let } h = x.\text{Cons.0 in let } h' = h + 1 \text{ in } \boxed{\text{Cons(h', x.Cons.1)}} \end{array} \right\}$$

In Example 6.4, we compiled x.Cons.0 with Extract. Our Compile function would call Seek, which would proceed similarly. Now, let us compile $u =$ Cons(h', x.Cons.1) with Rebuild.

Here is the call-graph of our various procedures in this case, where d is our destination location and $\Gamma = \{(x \triangleleft \text{Cons}(\_, \_) : \tau_{\text{list}} \text{ as } \widehat{\tau}_p), (h' \triangleleft \_ : I_{32} \text{ as } I_{32})\}$.
$\textsc{Rebuild}\ (\Gamma, (d : \tau_{\text{list}} \text{ as } \widehat{\tau}_p), u) =$
Explore$(\text{Cons}(\_, \_), \tau_{\text{list}}, \widehat{\tau}_p)$ (see Example 6.2)

- **branch** $(\text{Cons}(\_, \text{Nil}), \text{Cons}(I_{32}, \text{Nil}), \widehat{\tau}_{C1}, F_1)$
  - **call** $\mathcal{E}_c = \textsc{Refine}(d, \__{64}, \__{64} \ltimes [32 : 32] : \__{32} \ltimes [0 : 2] = 1) = \boxed{\texttt{let}_{\texttt{out}}\ d_c = d.[0 : 2];\ d_c := 1}$
  - **fragment** $.[32 : 32] \mapsto (.\text{Cons.0 as } I_{32})$ where **focus** $(.\text{Cons.0}, u) = h'$ and $\boxed{\texttt{let}_{\texttt{out}}\ d_f = d.[32 : 32]}$
    call $\mathcal{E}_f = \textsc{Seek}(\Gamma(h'), (d_f : I_{32} \text{ as } I_{32}), \epsilon) = \boxed{d_f := h'}$ **(base case)**
  - **yield** $(\text{Cons}(\_, \text{Nil}), \boxed{\mathcal{E}_c; \mathcal{E}_f})$
- **branch** $(\text{Cons}(\_, \text{Cons}(\_)), \text{Cons}(I_{32}, \text{Cons}(I_{32}, \tau_{\text{list}})), \widehat{\tau}_{C2}, F_2)$
  - **call** $\mathcal{E}'_c = \textsc{Refine}(d, \__{64}, \&_{64}\ (\{\!\{\__{32}, \__{32}, \__{64}\}\!\}) \ltimes [0 : 2] = 2)$ **(allocates)**

- **fragment** $(.*.0 \mapsto .\text{Cons.0 as } I_{32})$ where **focus** $(.\text{Cons.0}, u) = h'$ and $\boxed{\texttt{let}_{\texttt{out}}\ d_{f0} = d.*.0}$.
  call $\mathcal{E}'_{f1} = \textsc{Seek}(\Gamma(h'), (d_{f0} : I_{32} \text{ as } I_{32}), \epsilon) = \boxed{d_{f0} := h'}$ **(base case)**
- **fragment** $(.*.1 \mapsto \pi_l = .\text{Cons.1.Cons.0 as } I_{32})$ where **focus** $(\pi_l, u) = x.\pi_l$ and $\boxed{\texttt{let}_{\texttt{out}}\ d_{f1} = d.*.1}$.
  call $\mathcal{E}'_{f1} = \textsc{Seek}(\Gamma(x), (d_{f1} : I_{32} \text{ as } I_{32}), \pi_l) = \boxed{\texttt{let}_{\texttt{in}}\ s_{f1} = x.\text{Cons.0.Cons.1};\ d_{f1} := s_{f1}}$
- **fragment** $(.*.2 \mapsto \pi_r = .\text{Cons.1.Cons.1 as } \widehat{\tau_p})$ where **focus** $(\pi_r, u) = x.\pi_r$ and $\boxed{\texttt{let}_{\texttt{out}}\ d_{f2} = d.*.2}$.
  call $\mathcal{E}'_{f1} = \textsc{Seek}(\Gamma(x), (d_{f2} : I_{32} \text{ as } I_{32}), \pi_r) = \boxed{\texttt{let}_{\texttt{in}}\ s_{f2} = x.\text{Cons.1.Cons.1};\ d_{f2} := s_{f2}}$

yield $(\text{Cons}(\_, \text{Cons}(\_, \_)), \boxed{\mathcal{E}'_c; \mathcal{E}'_{f1}; \mathcal{E}'_{f2}; \mathcal{E}'_{f3}})$

return $\textsc{Destruct}\left((x : \widehat{\tau_p}), \left\{(\text{Cons}(\_, \text{Nil}), \boxed{\mathcal{E}_c; \mathcal{E}_f}), (\text{Cons}(\_, \text{Cons}(\_), \boxed{\mathcal{E}'_c; \mathcal{E}'_{f1}; \mathcal{E}'_{f2}; \mathcal{E}'_{f3}}))\right\}\right)$

7.6 Recursive Constructors

Although the algorithm presented so far is sufficient to handle most situations, it does not necessarily terminate in the presence of recursive types and layouts. Let us consider an example.

Example 7.1 (Recursive rebuilding of linked lists). Consider simply-linked lists of 32-bit integers $\tau = \text{Nil} \mid \text{Cons}(I_{32}, \tau)$ with two possible layouts. $\widehat{\tau_l}$ is a traditional "pointers and blocks" layout with a pointer for each element. $\widehat{\tau_p}$ is the packed layout already shown in Example 3.3. We can ask Rebuild to emit conversion code that transforms inputs of type τ with representation $\widehat{\tau_l}$ into an output of type τ with representation $\widehat{\tau_p}$: $\textsc{Rebuild}((x \triangleleft \_ : \tau \text{ as } \widehat{\tau_l}), (y : \tau \text{ as } \widehat{\tau_p}), \epsilon)$.

Naturally, such conversion code is not so simple: it requires recursive code that walks the structure and fuse blocks two by two. Our current algorithms will not terminate, as it tries to Rebuild each block in the list, infinitely. △

To properly handle such cases, we must emit *recursive constructor code*. Naturally, we could also refuse to emit such code (in contexts when recursion is not acceptable). In both cases, we need to detect recursion. We now sketch the main idea; full details are available in Appendix B. Intuitively, a call to Seek or Rebuild leads to infinite recursion if it attempts to recursively rebuild an output value with the same type, layout and relative position from an input value with the same type, layout and provenance. This indicates that the output value contains a subterm which must be rebuilt in the exact same way: the only way to emit correct code is to introduce an explicit recursive node and emit recursive calls at this position. For this purpose, we *memoise* Seek and Rebuild on their *anonymized* arguments, i.e. the arguments without any input or output memory locations. For instance, **prov_of**(u) is kept instead of u. We record when we enter one of the algorithms, and generate a fresh function symbol f. If we enter this function again, we emit a call $\texttt{call}\ f(x, y)$ with the appropriate arguments. Afterwards, we can use simple deforestation to get rid of extra functions. Note that, on top of emitting recursive code, this also improves sharing.

Example 7.2 (Linked lists, cont'd). Using memoisation, the Rebuild call from our previous example terminates and emits recursive code. We explore $\widehat{\tau_p}$ and get three branches from its split. The first two branch with the provenances Nil and Cons(_, Nil) are straightforward. The third branch Cons(_, Cons(_, _)) requires rebuilding the fragment $(.*.2 \mapsto .\text{Cons.1.Cons.1 as } \widehat{\tau_p})$, which is more involved. Indeed, the position .Cons.1.Cons.1 in the original list type $\widehat{\tau_l}$ represents itself a tail of type $\widehat{\tau_l}$. Eventually, we try to Rebuild a piece of type τ represented as $\widehat{\tau_p}$ from the same piece represented as $\widehat{\tau_l}$, i.e., the same task as the initial Rebuild call. Thanks to memoisation, this task is now associated with a function symbol f and we finally emit the target code in Fig. 12. △

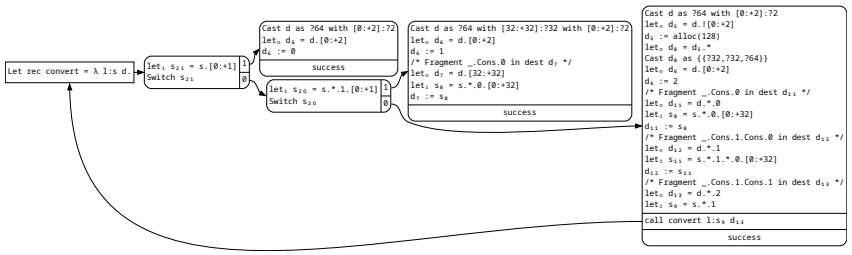


Fig. 12. Generated code for rebuilding linked lists s is the entry list and d the destination name.

8 METATHEORY

We now briefly sketch the semantics of our source and target languages and state the correctness of our compilation approach. Most of the definitions are delegated to the Appendix.

First, let us specify valid programs. We define a source typing judgement, denoted $\Gamma \vdash e : \tau$. This typing judgement is a straightforward monomorphic type system with ADTs. Additionally, we follow Baudon et al. [2023] by defining the *agreement* between a type and its layout, denoted $\mathbf{agree}(\tau, \widehat{\tau})$. This agreement for instance, ensures that paths are coherent, branches are all represented, etc. Both are defined in Appendix C.1. We define the shape of a memory value in a store, denoted $\mathbf{shape\_of}_{\varsigma}(\widehat{v})$, in a similar way to the shape of a memory layout (Appendix C, Fig. 20). The equivalence judgment between store/memory value pairs, denoted $\varsigma, \widehat{v} \sim \varsigma', \widehat{v}'$, holds if and only if $\widehat{v}$ and $\widehat{v}'$ are identical up to address renaming. Both are defined in Appendix C.2.

In order to give a precise account of our semantics, and be able to compare between source and compiled programs, we provide a semantic for the source language which precisely account for the state of memory, following the specified memory layout. For this purpose, we define a small-step evaluation judgment for our source language, denoted $\Gamma, \widehat{\sigma}, \varsigma, \widehat{e} \hookrightarrow \Gamma', \widehat{\sigma}', \varsigma', \widehat{e}'$, where:

- $\Gamma : \mathit{Vars} \to \mathit{Types} \times \widehat{\mathit{Types in shape}}$ is a typing environment (assigning both a type *and* a layout);
- $\widehat{\sigma} : \mathit{Vars} \to \widehat{\mathit{Values}}$ is a memory value environment (with the same domain as Γ);
- ς is a memory store used to interpret the contents of $\widehat{\sigma}$;
- $\widehat{e}$ is a *memory expression* that represents the memory value currently being built. Its full grammar is given in Appendix C.2 (Fig. 19); it includes both source expressions and memory values, along with intermediate stages containing both concrete memory structures and unevaluated triples $(u : \tau \text{ as } \widehat{\tau})$.

Normal forms are states where $\widehat{e}$ is a memory value. In order to reduce a source expression e to a memory value $\widehat{v}$ (along with its store ς), $\hookrightarrow$ must handle both high-level constructs (e.g., let, match) and memory-level construction of values (i.e., represent some valuexpression u according to a given layout $\widehat{\tau}$). Its full definition is available in Appendix C.2 (Figs. 22 and 23); here, we quickly demonstrate it on an example.

Example 8.1 (Source semantics on lists). Let us recall our list example with layout $\widehat{\tau_p}$ from Example 3.3, and evaluate the following expression in initially empty environments and store (environments Γ and $\widehat{\sigma}$ are merged for conciseness): $e = \text{let } x : \tau_{\text{list}} \text{ as } \widehat{\tau_p} = \text{Cons}(42, \text{Nil}) \text{ in } x.\text{Cons}.0$.

$$
\begin{aligned}
&\emptyset, \emptyset, \text{let } x : \tau_{\text{list}} \text{ as } \widehat{\tau_p} = \text{Cons}(42, \text{Nil}) \text{ in } (x.\text{Cons}.0 : I_{32} \text{ as } I_{32}) \\
\rightsquigarrow\ &\emptyset, \emptyset, \text{let } x : \tau_{\text{list}} \text{ as } \widehat{\tau_p} = \__{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (42 : I_{32} \text{ as } I_{32}) \text{ in } (x.\text{Cons}.0 : I_{32} \text{ as } I_{32}) \\
\rightsquigarrow\ &\emptyset, \emptyset, \text{let } x : \tau_{\text{list}} \text{ as } \widehat{\tau_p} = \__{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (42)_{32} \text{ in } (x.\text{Cons}.0 : I_{32} \text{ as } I_{32}) \\
\rightsquigarrow\ &\{x : \tau_{\text{list}} \text{ as } \widehat{\tau_p} \mapsto a.\epsilon\}, \{a \mapsto \__{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (42)_{32}\}, (x.\text{Cons}.0 : I_{32} \text{ as } I_{32}) \\
\rightsquigarrow\ &\{\ldots, x' : I_{32} \text{ as } I_{32} \mapsto a.[2:32]\}, \{\ldots\}, (x'.\epsilon : I_{32} \text{ as } I_{32}) \\
\rightsquigarrow\ &\{\ldots\}, \{\ldots\}, (42)_{32}
\end{aligned}
$$

△

For the target IR, we use a more standard small step semantics with a store modeling the content of the memory. Our target evaluation judgement is denoted $\rho_{\text{in}}, \rho_{\text{out}}, \varsigma, \mathcal{E} \rightsquigarrow \rho'_{\text{in}}, \rho'_{\text{out}}, \varsigma', \mathcal{E}'$, where:

- ρ_{in} and ρ_{out} map input and output location identifiers to concrete locations, which consist of an address and a memory path (i.e., unaligned pointers);
- ς is a memory store used to interpret the contents of ρ_{in} and ρ_{out};
- $\mathcal{E}$ is the target expression to evaluate.

Normal forms are states where $\mathcal{E}$ is a return statement (`success` or `fail`). The location environments ρ_{in} and ρ_{out} enforce a distinction between input and output locations, and directly reflect the effects of `let in` and `let out` instructions; for instance, we evaluate $\texttt{let}_{\texttt{in}}\ x' = x.\widehat{\pi}$ by looking up x in ρ_{in}, appending $\widehat{\pi}$ to $\rho_{\text{in}}(x)$ and binding the result to x' in ρ_{in}. The store ς is only used by instructions that allocate memory or perform actual reads/writes, such as switches and writes. The full judgment is available in appendix (Fig. 24).

We can now state our main result: given a source expression and its compiled target expression, their evaluation eventually yields identical memory contents.

Theorem 8.2 (Soundness). *Let τ a type, $\widehat{\tau}$ a layout and e a source expression such that $\widehat{\tau}$ agrees with τ and $\Gamma \vdash e : \tau$. Let x free in e and a a fresh address. Finally, let ς_0 and $\widehat{v}_0$ the initial memory store and value where $\widehat{v}_0$ is compatible with $\widehat{\tau}$, i.e. $\textbf{\textit{shape\_of}}_{\varsigma_0}(\widehat{v}_0) = \textbf{\textit{shape\_of}}(\widehat{\tau})$.*

The following conditions hold:

- Compile $(\emptyset, (x : \tau \text{ as } \widehat{\tau}), e)$ *succeeds and yields a target expression $\mathcal{E}$;*
- *the source evaluation reaches a normal form: $\emptyset, \emptyset, \emptyset, e \rightsquigarrow^* \Gamma, \widehat{\sigma}, \varsigma_s, \widehat{v}_s$;*
- *the target evaluation reaches a successful state: $\emptyset, \{x \mapsto a.\epsilon\}, \varsigma_0 \cup \{a \mapsto \widehat{v}_0\}, \mathcal{E} \rightsquigarrow^* \rho_{in}, \rho_{out}, \varsigma_t, \texttt{success}$ with $(x \mapsto a.\epsilon) \in \rho_{out}$ and there exists $\widehat{v}_t$ such that $(a \mapsto \widehat{v}_t) \in \varsigma_t$;*
- *the results of source and target evaluation are equivalent: $\varsigma_s, \widehat{v}_s \sim \varsigma_t, \widehat{v}_t$.*

9 RELATED WORK

Algebraic Data Types and low-level programming. ADTs, pattern matching compilation and compact memory representations all have long histories. We summarise the work directly related to low-level programming.

Our approach directly extends (and subsumes) [Baudon et al. 2023]. In particular, their "regular" case is covered by our initial naive algorithm in Section 6.2. Our full procedure covers all possible cases, including the so-called "irregular" ones which they only sketch.

Many of the links between ADTs and low-level programming were initially made for *verification*. Notably, Dargent [Chen et al. 2023] allows to specify memory representations in an external DSL which outputs C code for accessors, and Isabelle/HOL theorems; with the aim of formally verifying embedded systems. Swamy et al. [2022] propose a similar approach to formally verify binary format parsers in F^*. These approaches are precise, leveraging their host proof assistant, but do not provide

language-integrated constructs such as pattern matching. They also provide far less optimisations than what we propose.

LoCal [Vollmer et al. 2019] and Gibbon [Koparkar et al. 2021], on the other hand, provide DSLs tailored to describe and manipulate low-level and serialised representations. Their memory layouts are less flexible than what we presented, making it impossible to provide truly customised representations, but allowing them numerous powerful optimisations we do not provide, such as leveraging parallelism. We hope to combine our approaches in the future.

Finally, some general-purpose languages provide ways to improve data layout. Rust's niches [*RFC: Alignment niches for references types* 2021] provide semi-automatic layout optimisations, but are quite limited. Unboxed constructors [Colin et al. 2018; Keller et al. 2010] allow for manual optimisations, but prevent the use of nice high-level constructs, falling back to a C-like programming style. By contrast, our approach allows using only a high-level view, while giving full control over memory layout.

Intermediate Representation. We use a Destination-Passing Style [Shaikhha et al. 2017] representation in *A-normal form* [2023]. This provides us precise control over memory management and input/output arguments, and could enable further memory improvements, such as using stack allocation when appropriate and applying tail-call modulo cons [Bour et al. 2021]. Another avenue would naturally be to use Continuation-Passing Style [Appel 1992], notably to simplify handling of recursive calls in Section 7.6. This is in line with numerous compilers for functional languages [Hall et al. 1992; Vincent Laviron 2023] and easily allows moving to SSA representations such as Rust's MIR and LLVM.

CONCLUSION

We presented a unified compilation procedure for constructors and destructors of Algebraic Data Types using a Destination-Passing Style intermediate representation. Our work allows providing arbitrary memory layouts for ADTs and compiles high-level code to low-level programs accordingly. In the future, we hope to investigate memory management strategies, for instance following Lorenzen et al. [2023].

REFERENCES

A-normal form. https://en.wikipedia.org/w/index.php?title=A-normal_form&oldid=1121147927. [Online; accessed 19-February-2023]. (2023).

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

Andrew W. Appel. 1992. *Compiling with Continuations*. Cambridge University Press. ISBN: 0-521-41695-7.

Lennart Augustsson. 1985. "Compiling pattern matching". In: *Functional Programming Languages and Computer Architecture*. Ed. by Jean-Pierre Jouannaud. Springer Berlin Heidelberg, Berlin, Heidelberg, 368–381. ISBN: 978-3-540-39677-2.

Thaïs Baudon, Gabriel Radanne, and Laure Gonnord. Aug. 2023. "Bit-Stealing Made Legal: Compilation for Custom Memory Representations of Algebraic Data Types". *Proc. ACM Program. Lang.*, 7, ICFP, (Aug. 2023). DOI: 10.1145/3607858.

Frédéric Bour, Basile Clément, and Gabriel Scherer. 2021. "Tail Modulo Cons". *CoRR*, abs/2102.09823. https://arxiv.org/abs/2102.09823 arXiv: 2102.09823.

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. DOI: 10.1145/800087.802799.

Zilin Chen, Ambroise Lafont, Liam O'Connor, Gabriele Keller, Craig McLaughlin, Vincent Jackson, and Christine Rizkallah. Jan. 2023. "Dargent: A Silver Bullet for Verified Data Layout Refinement". *Proc. ACM Program. Lang.*, 7, POPL, (Jan. 2023). DOI: 10.1145/3571240.

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

Cordelia V. Hall, Kevin Hammond, Will Partain, Simon L. Peyton Jones, and Philip Wadler. 1992. "The Glasgow Haskell Compiler: A Retrospective". In: *Functional Programming, Glasgow 1992, Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, UK, 6-8 July 1992* (Workshops in Computing). Ed. by John Launchbury and Patrick M. Sansom. Springer, 62–71. DOI: 10.1007/978-1-4471-3215-8_6.

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*. Ed. by Paul Hudak and Stephanie Weirich. ACM, 261–272. DOI: 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. DOI: 10.1145/3473596.

Anton Lorenzen, Daan Leijen, and Wouter Swierstra. Aug. 2023. "FP^2: Fully in-Place Functional Programming". *Proc. ACM Program. Lang.*, 7, ICFP, (Aug. 2023). DOI: 10.1145/3607840.

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*. Ed. by Eijiro Sumii. ACM, 35–46. DOI: 10.1145/1411304.1411311.

Luc Maranget. 2007. "Warnings for pattern matching". *J. Funct. Program.*, 17, 3, 387–421. DOI: 10.1017/S0956796807006223.

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

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

Amir Shaikhha, Andrew Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. 2017. "Destination-Passing Style for Efficient Memory Management". In: *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing* (FHPC 2017). Association for Computing Machinery, Oxford, UK, 12–23. ISBN: 9781450351812. DOI: 10.1145/3122948.3122949.

Nikhil Swamy et al.. 2022. "Hardening attack surfaces with formally proven binary format parsers". In: *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*. Ed. by Ranjit Jhala and Isil Dillig. ACM, 31–45. DOI: 10.1145/3519939.3523708.

[SW exc.] Linus Torvalds, "Red-Black Trees in Linux", from *The Linux Kernel* version 6.2, 2023. LIC: GPL-2.0 WITH Linux-syscall-note. URL: https://github.com/torvalds/linux, SWHID: ⟨swh:1:cnt:45b6ecde3665aa744f790cd915445fe07595181c;origin=https://github.com/torvalds/linux;visit=swh:1:snp:de81d8ff32247a7edaa935cf0468bf16237d25c5;anchor=swh:1:rel:32758e7a720e4752a824c6062e75f107314e5598;path=/include/linux/rbtree_types.h⟩.

Mark Shinwell Vincent Laviron Pierre Chambart. 2023. "Efficient OCaml Compilation with Flambda 2". *OCaml*. https://icfp23.sigplan.org/details/ocaml-2023-papers/8/Efficient-OCaml-compilation-with-Flambda-2.

Michael Vollmer, Chaitanya Koparkar, Mike Rainey, Laith Sakka, Milind Kulkarni, and Ryan R. Newton. 2019. "LoCal: a language for programs operating on serialized data". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. Ed. by Kathryn S. McKinley and Kathleen Fisher. ACM, 48–62. DOI: 10.1145/3314221.3314631.

Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. Dec. 2019. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 20191213*. Tech. rep. RISCV fundation, (Dec. 2019).

A FOCUSING AND SPECIALISATION

We now give full details of the definition of focusing and specialisation.

Focusing in Types and Layouts. "Focusing", defined in Fig. 13 allows to focus on a specific part of a type or layout, according to a given path. Focus in the high-level language is denoted $\mathbf{focus}\ (\pi, \theta)$ where θ is a type, an expression, a provenance, or another path, and returns an object of the same kind. It simply follows the syntax to extract the subterm at position π. For instance, we can consider "the part that is relevant to .Sw.0" in the type $\tau_{\text{RISC-V}}$(Example 3.1):

$$\mathbf{focus}\ (.\text{Sw}.0, \tau_{\text{RISC-V}}) = \mathbf{focus}\ \left(.\text{Sw}.0, \text{Sw}(\tau_{\text{reg}}, \tau_{\text{reg}}, I_{12})\right) = \tau_{\text{reg}}$$

Layout focusing (Fig. 14), denoted $\widehat{\mathbf{focus}}\ (\widehat{\pi}, \widehat{\tau})$, similarly extracts the layout located at position $\widehat{\pi}$ within the parent layout $\widehat{\tau}$. It is undefined on splits. For instance, $\widehat{\mathbf{focus}}\ (.[7:5], \widehat{\tau}_{\text{Sw}}) = (.\text{Sw}.0 \text{ as } \widehat{\tau}_{\text{reg}})$.

$$\begin{array}{ll} \mathbf{focus}\ (\epsilon, x) = x & \mathbf{focus}\ (.i, \langle x_1, \ldots, x_n \rangle) = x_i \\ \mathbf{focus}\ (\pi, \_) = \_ & \mathbf{focus}\ (.K_i, K_1(x_1) + \ldots + K_n(x_n)) = x_i \end{array}$$

Fig. 13. Type Focusing.

$$\begin{array}{l} \widehat{\mathbf{focus}}\ (\epsilon, \widehat{\tau}) = \widehat{\tau} \qquad\qquad \widehat{\mathbf{focus}}\ (.i, \{\!\{\widehat{\tau}_1, \ldots, \widehat{\tau}_n\}\!\}) = \widehat{\tau}_i \\ \widehat{\mathbf{focus}}\ (.[o_k : \ell_k], \__\ell \ltimes_{1 \le i \le n} [o_i : \ell_i] : \widehat{\tau}_i) = \widehat{\tau}_k \\ \widehat{\mathbf{focus}}\ (.[o_k : \ell_k], \&_\ell\ (\widehat{\tau}) \ltimes_{1 \le i \le n} [o_i : \ell_i] : \widehat{\tau}_i) = \widehat{\tau}_k \\ \widehat{\mathbf{focus}}\ (.*, \&_\ell\ (\widehat{\tau}) \ltimes_{1 \le i \le n} [o_i : \ell_i] : \widehat{\tau}_i) = \widehat{\tau} \end{array}$$

Fig. 14. Layout Focusing.

Layout and Type Specialisation. "Specialisation", defined in Fig. 15 filters a type or a layout to exclude parts which are incompatible with a given provenance. Type specialisation, denoted $\tau \big/ p$, is a simple syntactic filter that discards irrelevant constructors. For instance,

$$\tau_{\text{RISC-V}} \big/ \text{Sw}(\_, \_, \_) = \text{Sw}(\tau_{\text{reg}}, \tau_{\text{reg}}, I_{12}).$$

Layout specialisation, denoted $\widehat{\tau} \big/ p$ (Fig. 16), is more complex: it removes all splits from $\widehat{\tau}$ (up to fragments) by filtering out branches whose provenance set excludes p. It returns a list of pairs of the form $(p' \mapsto \widehat{\tau}')$, where p' is a refined version of p and $\widehat{\tau}'$ is the restriction of $\widehat{\tau}$ to values that match p'. For instance, $\widehat{\tau}_{\text{RISC-V}} \big/ \text{Sw}(\_, \_, \_)$ returns a single pair $(\text{Sw}(\_, \_, \_) \mapsto \widehat{\tau}_{\text{Sw}})$, and specialisation according to the wildcard provenance lists all possible refinement pairs of a layout:

$$\widehat{\tau}_{\text{RISC-V}} \big/ \_ = \left\{ \begin{array}{c} (\text{Sw}(\_, \_, \_) \mapsto \widehat{\tau}_{\text{Sw}}), (\text{Add}(\_, \_, \_) \mapsto \widehat{\tau}_{\text{Add}}), \\ (\text{Addi}(\_, \_, \_) \mapsto \widehat{\tau}_{\text{Addi}}), (\text{Jal}(\_, \_) \mapsto \widehat{\tau}_{\text{Jal}}) \end{array} \right\}$$

$$\tau \big/ \_ = \tau \qquad\qquad \langle \tau_1, \ldots, \tau_n \rangle \;/\; \langle p_1, \ldots, p_n \rangle = \big\langle \tau_1 \big/ p_1, \ldots, \tau_n \big/ p_n \big\rangle$$

$$K_1(\tau_1) + \ldots + K_n(\tau_n) \big/ K(p) = K_i(\tau_i \big/ p)$$

Fig. 15. Type Specialisation

$$\tau \big/ \_ = \{ \_ \mapsto \tau \} \qquad\qquad (\pi \text{ as } \widehat{\tau}) \big/ p = \{ p \mapsto (\pi \text{ as } \widehat{\tau}) \}$$

$$\{\!\{ \widehat{\tau}_1, \ldots, \widehat{\tau}_n \}\!\} \big/ p = \left\{ p' \mapsto \{\!\{ \widehat{\tau}'_1, \ldots, \widehat{\tau}'_n \}\!\} \;\middle|\; \begin{array}{l} (p_i \mapsto \widehat{\tau}'_i) \in \widehat{\tau}_i \big/ p \\ \\ p_1 \cap \ldots \cap p_n = p' \end{array} \right\}$$

$$\text{split}\,(\widehat{\pi}) \begin{Bmatrix} c_1 \text{ from } P_1 \Rightarrow \widehat{\tau}_1 \\ \vdots \quad \text{from} \; \vdots \quad \Rightarrow \vdots \\ c_n \text{ from } P_n \Rightarrow \widehat{\tau}_n \end{Bmatrix} \big/ p = \left\{ \widehat{\tau}_i \big/ p \;\middle|\; \begin{array}{l} p' \in P_i \\ \\ p \subset p' \end{array} \right\}$$

Fig. 16. Layout Specialisation.

$$|t| = |\Delta(t)| \qquad |I_\ell| = \ell \qquad |\__\ell| = \ell \qquad |(c)_\ell| = \ell \qquad |\&_\ell\,(\widehat{\tau})| = \ell \qquad \left| \widehat{\tau} \bowtie_{0 \le i < n} [o_i : \ell_i] : \widehat{\tau}_i \right| = |\widehat{\tau}|$$

$$|\{\!\{ \widehat{\tau}_1, \ldots, \widehat{\tau}_n \}\!\}| = |\widehat{\tau}_1| + \ldots + |\widehat{\tau}_n| \qquad\qquad |(\pi \text{ as } \widehat{\tau})| = |\widehat{\tau}|$$

$$\left| \text{split}\,(\widehat{\pi}) \left\{ c_i \text{ from } P_i \Rightarrow \widehat{\tau}_i \;\middle|\; 1 \le i \le n \right\} \right| = \max_{1 \le i \le n} |\widehat{\tau}_i|$$

Fig. 17. Size of a memory layout in type environment $\Delta : \mathit{TyVars} \to \widehat{\mathit{Types}}$.

B RECURSIVE CONSTRUCTORS

We now detail the extension of our algorithm to handle such cases by emitting recursive constructor code. The idea is to replace Rebuild and Seek with Wrap(Rebuild) and Wrap(Seek) respectively. The Wrap function, defined in Algorithm 8, hashes arguments to keep track of which calls have already been performed. Each argument hash is associated with a function symbol, and any subsequent call on the same arguments returns a call to this function.

```
1  function Wrap(Fun):
2    H := empty
3    return λ (⟨s_in, τ_in, τ̂_in, p_in⟩, ⟨d_out, τ_out, τ̂_out⟩, π) . {
4      h ← (τ_in, τ̂_in, p_in, τ_out, τ̂_out, π)
5      if h ∈ dom(H) then
6        f ← H(h)
7        return call f(s_in, d_out); success
8      else
9        f, s, d ← fresh symbols
10       H(h) := Declared(f)
11       args'_in ← ⟨s, τ_in, τ̂_in, p_in⟩
12       args'_out ← ⟨d, τ_out, τ̂_out⟩
13       E_body ← Fun(args'_in, args'_out, π)
14       H(h) := Defined(f, λ(s, d).E_body)
15       return call f(s_in, d_out); success
16     end
17   }
```

Algorithm 8: Wrapper for emitting recursive code

C ASSORTED DEFINITIONS

C.1 Types and layouts

Agreement criteria define the relationship between a high-level type and its memory layout specification.

Definition C.1 (Agreement between types and layouts). Let τ a high-level type and $\widehat{\tau}$ a memory layout. We say that $\widehat{\tau}$ *represents* τ, or *agrees* with τ, and we write $\mathbf{agree}(\tau, \widehat{\tau})$, if either τ and $\widehat{\tau}$ are integer types of the same width (i.e., $\tau = \widehat{\tau} = I_\ell$) or the following conditions hold:

All fragments bind subterms to their valid representation. (Fragment Coherence)
For all $\widehat{\pi}$ such that $\widehat{\mathbf{focus}}\ (\widehat{\pi}, \widehat{\tau}) = (\pi \text{ as } \widehat{\tau}')$, $\tau' = \mathbf{focus}\ (\pi, \tau)$ is defined and $\widehat{\tau}'$ agrees with τ'.

Split branches are valid representations of high-level subtypes. (Branch Coherence)
For all $\widehat{\pi}$ such that $\widehat{\mathbf{focus}}\ (\widehat{\pi}, \widehat{\tau}) = \text{split}\ (\widehat{\pi}')\ \{c_i \text{ from } P_i \Rightarrow \widehat{\tau}_i\}$, for each i and each $p \in P_i$ such that p is a provenance of type τ, $\widehat{\tau}_i$ agrees with τ / p.

All data from the high-level type is represented within the memory layout. (Coverage)
For every π that leads to a single bit in τ (i.e., $\mathbf{focus}\ (\pi, \tau) = i1$), $\widehat{\tau}$ *covers* π: every memory type $\widehat{\tau}' \in \widehat{\tau} / \pi$ contains a fragment for a source position π_0 prefix of π. More precisely, there exist source and memory paths $\pi_0, \widehat{\pi}_0$ such that $\widehat{\mathbf{focus}}\ (\widehat{\pi}_0, \widehat{\tau}') = (\pi_0 \text{ as } \widehat{\tau}'')$ and either $\pi = \pi_0.\pi'$ or $\pi = \pi'.[o : \ell]$, $\pi_0 = \pi'.[o_0 : \ell_0]$ and $o_0 \le o \le o + \ell \le o_0 + \ell_0$.

Memory layouts can tell incompatible provenances apart. (Distinguishability)
For every pair of provenances p_0, p_1 of type τ such that p_0 and p_1 are incompatible (that is, there exists some position π such that $\mathbf{focus}\ (\pi, p_0)$ and $\mathbf{focus}\ (\pi, p_1)$ have different head constructors), $\widehat{\tau}$ *distinguishes* between p_0 and p_1. More precisely, for any $\widehat{\tau}_0 \in \widehat{\tau} / p_0$ and $\widehat{\tau}_1 \in \widehat{\tau} / p_1$, there exists a memory path $\widehat{\pi}$ such that $\widehat{\mathbf{focus}}\ (\widehat{\pi}, \widehat{\tau}_0) = (c_0)_\ell$, $\widehat{\mathbf{focus}}\ (\widehat{\pi}, \widehat{\tau}_1) = (c_1)_\ell$ and $c_0 \neq c_1$.

$$\frac{(t \mapsto \widehat{\tau}) \in \Delta \qquad \Delta, \Gamma \vdash e : \tau \text{ as } \widehat{\tau}}{\Delta, \Gamma \vdash e : \tau \text{ as } t} \qquad \frac{\mathbf{agree}(\tau, \widehat{\tau}) \qquad \Delta, \Gamma \vdash u : \tau}{\Delta, \Gamma \vdash (u : \tau \text{ as } \widehat{\tau}) : (\tau \text{ as } \widehat{\tau})}$$

$$\frac{\Delta, \Gamma \vdash e : \tau \text{ as } \widehat{\tau} \qquad \Delta, \Gamma \cup \{(x : \tau \text{ as } \widehat{\tau})\} \vdash e_0 : (\tau_0 \text{ as } \widehat{\tau_0})}{\Delta, \Gamma \vdash \text{let } x = e \text{ in } e_0 : (\tau_0 \text{ as } \widehat{\tau_0})}$$

$$\frac{\begin{array}{c}\mathbf{agree}(\tau, \widehat{\tau}) \\ \Delta, \Gamma \cup \{x : (\tau \text{ as } \widehat{\tau})\} \vdash e : (\tau' \text{ as } \widehat{\tau'}) \qquad \Delta, \Gamma \cup \{f : (\tau \text{ as } \widehat{\tau}) \to (\tau' \text{ as } \widehat{\tau'})\} \vdash e_0 : (\tau_0 \text{ as } \widehat{\tau_0})\end{array}}{\Delta, \Gamma \vdash \text{fun } f(x : \tau \text{ as } \widehat{\tau}) = e \text{ in } e_0 : (\tau_0 \text{ as } \widehat{\tau_0})}$$

$$\frac{(x : (\tau \text{ as } \widehat{\tau})) \in \Gamma}{\Delta, \Gamma \vdash x : (\tau \text{ as } \widehat{\tau})} \qquad \frac{(f : (\tau \text{ as } \widehat{\tau}) \to (\tau' \text{ as } \widehat{\tau'})) \in \Gamma \qquad (x : (\tau \text{ as } \widehat{\tau})) \in \Gamma}{\Delta, \Gamma \vdash f(x) : (\tau' \text{ as } \widehat{\tau'})}$$

$$\frac{x : \tau \text{ as } \widehat{\tau} \in \Gamma \qquad \Delta \vdash \text{pat}_i : \tau \qquad \Delta, \Gamma \vdash e_i : (\tau' \text{ as } \widehat{\tau'}) \qquad \forall p \in \mathbf{provs\_of}_\Delta(\tau), \exists i, \text{pat}_i \rhd p}{\Delta, \Gamma \vdash \text{match}(x)\{\text{pat}_1 \to e_1 \ \ldots \ \text{pat}_n \to e_n\} : (\tau' \text{ as } \widehat{\tau'})}$$

Fig. 18. Typing judgment for source expressions (grammar in Fig. 8): $\Delta, \Gamma \vdash e : \tau \text{ as } \widehat{\tau}$ with $\Delta : \mathit{TyVars} \to \mathit{Types} \cup \widehat{\mathit{Types}}$ and $\Gamma : \mathit{Vars} \to \mathit{Types} \times \widehat{\mathit{Types}}$

C.2 Memory values and stores

The notion of *memory valuexpressions*, whose grammar is depicted in Fig. 19, abstracts from previously defined memory contents (Fig. 9), and is in our formalism used to capture values halfway during the interpretation process.

Memory valuexpressions

$$
\begin{array}{rll}
\widehat{u} \in \widehat{\mathit{ValuExprs}} ::= & (u : \tau \text{ as } \widehat{\tau}) & \text{(unevaluated representation)} \\
\mid & \__\ell & \text{(opaque word)} \\
\mid & (c)_\ell & \text{(constant word)} \\
\mid & \&_\ell\,(\widehat{u}) & \text{(addressless pointer)} \\
\mid & \&_\ell\,(a) & \text{(pointer value)} \\
\mid & \widehat{u} \underset{0 \le i < n}{\bowtie} [o_i : \ell_i] : \widehat{u}_i & \text{(composite word)} \\
\mid & \{\!\{\widehat{u}_0, \ldots, \widehat{u}_{n-1}\}\!\} & \text{(struct)}
\end{array}
$$

Memory expressions

$$
\begin{array}{rll}
\widehat{e} \in \widehat{\mathit{Exprs}} ::= & e & \text{(source expression)} \\
\mid & \widehat{u} & \text{(memory valuexpression)} \\
\mid & \text{let } x : \tau \text{ as } \widehat{\tau} = \widehat{e} \text{ in } e & \text{(let-binding intermediate stage)}
\end{array}
$$

Fig. 19. Memory expressions and valuexpressions, representing intermediate stages between source expressions and memory values

The notion of shape of a memory value $\widehat{v}$ *according to a store* $\varsigma : \mathit{Addrs} \to \widehat{\mathit{Values}}$ (Fig. 20) is useful to characterise our equivalence notion (Fig. 21)

$$
\begin{array}{l}
\mathbf{shape\_of}_\varsigma\{ \\
\quad \begin{array}{ll}
\__\ell & \longrightarrow \__\ell \\
(c)_\ell & \longrightarrow (c)_\ell \\
\&_\ell\,(a) & \longrightarrow \&_\ell\left(\mathbf{shape\_of}_\varsigma(\widehat{v})\right) \quad \text{when } (a \mapsto \widehat{v}) \in \varsigma \\
\widehat{v} \underset{0 \le i < n}{\bowtie} [o_i : \ell_i] : \widehat{v}_i & \longrightarrow \mathbf{shape\_of}_\varsigma(\widehat{v}) \underset{0 \le i < n}{\bowtie} [o_i : \ell_i] : \mathbf{shape\_of}_\varsigma(\widehat{v}_i) \\
\{\!\{\widehat{v}_0, \ldots, \widehat{v}_{n-1}\}\!\} & \longrightarrow \left\{\!\!\left\{\mathbf{shape\_of}_\varsigma(\widehat{v}_0), \ldots, \mathbf{shape\_of}_\varsigma(\widehat{v}_{n-1})\right\}\!\!\right\}
\end{array} \\
\}
\end{array}
$$

Fig. 20. Shape of a memory value $\widehat{v}$ in a store ς: $\mathbf{shape\_of}_\varsigma(\widehat{v})$

$$\varsigma, \__\ell \sim \varsigma', \__\ell \qquad \varsigma, (c)_\ell \sim \varsigma', (c)_\ell \qquad \frac{(a \mapsto \widehat{v}) \in \varsigma \qquad (a' \mapsto \widehat{v}') \in \varsigma' \qquad \varsigma, \widehat{v} \sim \varsigma', \widehat{v}'}{\varsigma, \&_\ell\,(a) \sim \varsigma', \&_\ell\,(a')}$$

$$\frac{\varsigma, \widehat{v} \sim \varsigma', \widehat{v}' \qquad \varsigma, \widehat{v}_i \sim \varsigma', \widehat{v}'_i}{\varsigma, \widehat{v} \bowtie_{0 \le i < n} [o_i : \ell_i] : \widehat{v}_i \sim \varsigma', \widehat{v}' \bowtie_{0 \le i < n} [o_i : \ell_i] : \widehat{v}'_i} \qquad \frac{\varsigma, \widehat{v}_i \sim \varsigma', \widehat{v}'_i}{\varsigma, \{\!\{\widehat{v}_0, \ldots, \widehat{v}_{n-1}\}\!\} \sim \varsigma', \{\!\{\widehat{v}'_0, \ldots, \widehat{v}'_{n-1}\}\!\}}$$

Fig. 21. Equivalence between two memory values considered in their stores: $\varsigma, \widehat{v} \sim \varsigma', \widehat{v}'$ (this is equivalent to $\textbf{shape\_of}_{\varsigma}(\widehat{v}) = \textbf{shape\_of}_{\varsigma'}(\widehat{v}')$)

C.3 Semantics/Evaluation

In order to state the correctness of our compilation algorithms, we need to define a "memory-aware" small steps semantics for our source language, in Fig. 22 (toplevel rules) and Fig. 23 (rules to construct and evaluate values).

The judgment is denoted $\Gamma, \widehat{\sigma}, \varsigma, \widehat{e} \hookrightarrow \Gamma', \widehat{\sigma}', \varsigma', \widehat{e}'$, where:

- $\Gamma : \mathit{Vars} \to \mathit{Types} \times \widehat{\mathit{Types in shape}}$ is a typing environment (assigning both a type *and* a layout);
- $\widehat{\sigma} : \mathit{Vars} \to \widehat{\mathit{Values}}$ is a memory value environment (with the same domain as Γ);
- ς is a memory store used to interpret the contents of $\widehat{\sigma}$;
- $\widehat{e}$ is a *memory expression* that represents the memory value currently being built. it includes both source expressions and memory values, along with intermediate stages containing both concrete memory structures and unevaluated triples $(u : \tau \text{ as } \widehat{\tau})$.

Normal forms are states where $\widehat{e}$ is a memory value. In order to reduce a source expression e to a memory value $\widehat{v}$ (along with its store ς), $\hookrightarrow$ must handle both high-level constructs (e.g., let, match) and memory-level construction of values (i.e., represent some valuexpression u according to a given layout $\widehat{\tau}$).

$$\textsc{Variable}\ \frac{(x \mapsto \widehat{v}) \in \widehat{\sigma}}{\Delta, \Sigma \vdash \Gamma, \widehat{\sigma}, \varsigma, x \hookrightarrow \Gamma, \widehat{\sigma}, \varsigma, \widehat{v}} \qquad \textsc{FunApp}\ \frac{(x \mapsto \widehat{v}) \in \widehat{\sigma} \qquad (f \mapsto \lambda x'.e) \in \Sigma}{\Delta, \Sigma \vdash \Gamma, \widehat{\sigma}, \varsigma, f(x) \hookrightarrow \Gamma \cup \{x' : \Gamma(x)\}, \widehat{\sigma} \cup \{x' \mapsto \widehat{v}\}, \varsigma, e}$$

$$\textsc{LetStep}\ \frac{\Delta, \Sigma \vdash \Gamma, \widehat{\sigma}, \varsigma, \widehat{e} \hookrightarrow \Gamma', \widehat{\sigma}', \varsigma', \widehat{e}'}{\Delta, \Sigma \vdash \Gamma, \widehat{\sigma}, \varsigma, \mathsf{let}\ x : \tau\ \mathsf{as}\ \widehat{\tau} = \widehat{e}\ \mathsf{in}\ e \hookrightarrow \Gamma', \widehat{\sigma}', \varsigma', \mathsf{let}\ x : \tau\ \mathsf{as}\ \widehat{\tau} = \widehat{e}'\ \mathsf{in}\ e}$$

$$\textsc{LetBind}\ \Delta, \Sigma \vdash \Gamma, \widehat{\sigma}, \varsigma, \mathsf{let}\ x : \tau\ \mathsf{as}\ \widehat{\tau} = \widehat{v}\ \mathsf{in}\ e \hookrightarrow \Gamma \cup \{x : \tau\ \mathsf{as}\ \widehat{\tau}\}, \widehat{\sigma} \cup \{x \mapsto \widehat{v}\}, \varsigma, e$$

$$\textsc{Match}\ \frac{(x : \tau\ \mathsf{as}\ \widehat{\tau}) \in \Gamma \qquad (x \mapsto \widehat{v}) \in \widehat{\sigma} \qquad \textbf{eval\_match}\,(\tau, \widehat{\tau}, \widehat{v}, \varsigma, \{p_i \mid 1 \le i \le n\}) = i}{\Delta, \Sigma \vdash \Gamma, \widehat{\sigma}, \varsigma, \mathsf{match}(x)\{p_1 \to e_1\ \ldots\ p_n \to e_n\} \hookrightarrow \Gamma, \widehat{\sigma}, \varsigma, e_i}$$

Fig. 22. Evaluation judgment for source expressions, high-level rules: $\Gamma, \widehat{\sigma}, \varsigma, \widehat{e} \hookrightarrow \Gamma', \widehat{\sigma}', \varsigma', \widehat{e}'$. Δ is a global type variable environment and Σ a global function definition environment. Environments that are not relevant to a particular rule are displayed in gray. The Match rule relies on a memory-level pattern matching evaluation function **eval_match** which implements the approach presented in [Baudon et al. 2023] and returns the smallest i such that the i-th pattern matching branch matches, or $\bot$ if no branch matches.

It remains to define the semantics of our target representation (see grammar in Fig. 11), in Fig. 24.

$$\widehat{C}[\square] ::= \square \mid \&_\ell\left(\widehat{C}[\square]\right) \mid \widehat{C}[\square] \ltimes b : \widehat{u} \mid \widehat{u} \ltimes b : \widehat{C}[\square] \mid \left\{\!\!\left\{\widehat{u}, \ldots, \widehat{u}, \widehat{C}[\square], \widehat{u}, \ldots, \widehat{u}\right\}\!\!\right\} \mid \text{let } x : \tau \text{ as } \widehat{\tau} = \widehat{C}[\square] \text{ in } e$$

SubStep
$$\frac{\Delta, \Sigma \vdash \Gamma, \widehat{\sigma}, \varsigma, \widehat{u} \hookrightarrow \Gamma', \widehat{\sigma}', \varsigma', \widehat{u}'}{\Delta, \Sigma \vdash \Gamma, \widehat{\sigma}, \varsigma, \widehat{C}[\widehat{u}] \hookrightarrow \Gamma', \widehat{\sigma}', \varsigma', \widehat{C}[\widehat{u}']}$$

Alloc
$$\frac{a \notin \mathrm{dom}\,(\varsigma)}{\Delta, \Sigma \vdash \Gamma, \widehat{\sigma}, \varsigma, \&_\ell\,(\widehat{v}) \hookrightarrow \Gamma, \widehat{\sigma}, \varsigma \cup \{a \mapsto \widehat{v}\}, \&_\ell\,(a)}$$

TypeVar
$$\frac{(t \mapsto \widehat{\tau}) \in \Delta \qquad \Delta, \Sigma \vdash \Gamma, \widehat{\sigma}, \varsigma, (u : \tau \text{ as } \widehat{\tau}) \hookrightarrow \Gamma', \widehat{\sigma}', \varsigma', \widehat{e}}{\Delta, \Sigma \vdash \Gamma, \widehat{\sigma}, \varsigma, (u : \tau \text{ as } t) \hookrightarrow \Gamma', \widehat{\sigma}', \varsigma', \widehat{e}}$$

Split
$$\frac{\widehat{\tau} = \mathrm{split}\,(\ldots) \qquad P = \left\{x \mapsto \widehat{\mathbf{prov\_of}}_{\Delta,\varsigma}(\widehat{\tau}_x, \widehat{\sigma}(x)) \;\middle|\; (x : \tau_x \text{ as } \widehat{\tau}_x) \in \Gamma\right\} \qquad p = \mathbf{prov\_of}_\Delta(\tau, u)\,[.\pi \leftarrow \mathbf{focus}\,(\pi', P(x)) \mid \mathbf{focus}\,(\pi, u) = x.\pi'] \qquad \widehat{\tau}/p = \{(p, \widehat{\tau}')\}}{\Delta, \Sigma \vdash \Gamma, \widehat{\sigma}, \varsigma, (u : \tau \text{ as } \widehat{\tau}) \hookrightarrow \Gamma, \widehat{\sigma}, \varsigma, (u : \tau/p \text{ as } \widehat{\tau}')}$$

Fragment
$$\Delta \vdash \Gamma, \widehat{\sigma}, \varsigma, (u : \tau \text{ as } (\pi \text{ as } \widehat{\tau})) \hookrightarrow \Gamma, \widehat{\sigma}, \varsigma, (\mathbf{focus}\,(\pi, u) : \mathbf{focus}\,(\pi, \tau) \text{ as } \widehat{\tau})$$

Atom
$$\Delta \vdash \Gamma, \widehat{\sigma}, \varsigma, (c : \tau \text{ as } I_\ell) \hookrightarrow \Gamma, \widehat{\sigma}, \varsigma, (c)_\ell$$

Constant
$$\Delta \vdash \Gamma, \widehat{\sigma}, \varsigma, (u : \tau \text{ as } (c)_\ell) \hookrightarrow \Gamma, \widehat{\sigma}, \varsigma, (c)_\ell$$

Word
$$\Delta \vdash \Gamma, \widehat{\sigma}, \varsigma, (u : \tau \text{ as } \__\ell) \hookrightarrow \Gamma, \widehat{\sigma}, \varsigma, \__\ell$$

Pointer
$$\Delta \vdash \Gamma, \widehat{\sigma}, \varsigma, (u : \tau \text{ as } \&_\ell\,(\widehat{\tau})) \hookrightarrow \Gamma, \widehat{\sigma}, \varsigma, \&_\ell\,((u : \tau \text{ as } \widehat{\tau}))$$

BitRange
$$\Delta \vdash \Gamma, \widehat{\sigma}, \varsigma, (u : \tau \text{ as } \widehat{\tau} \ltimes b : \widehat{\tau}') \hookrightarrow \Gamma, \widehat{\sigma}, \varsigma, (u : \tau \text{ as } \widehat{\tau}) \ltimes b : (u : \tau \text{ as } \widehat{\tau}')$$

Struct
$$\Delta \vdash \Gamma, \widehat{\sigma}, \varsigma, (u : \tau \text{ as } \{\!\{\widehat{\tau}_1, \ldots, \widehat{\tau}_n\}\!\}) \hookrightarrow \Gamma, \widehat{\sigma}, \varsigma, \{\!\{(u : \tau \text{ as } \widehat{\tau}_1), \ldots, (u : \tau \text{ as } \widehat{\tau}_n)\}\!\}$$

Extract-like
$$\frac{(x : \tau_x \text{ as } \widehat{\tau}_x) \in \Gamma \qquad \widehat{\tau}_x = \widehat{\tau}}{\Delta, \Sigma \vdash \Gamma, \widehat{\sigma}, \varsigma, (x.\epsilon : \tau \text{ as } \widehat{\tau}) \hookrightarrow \Gamma, \widehat{\sigma}, \varsigma, \widehat{\sigma}(x)}$$

Seek-like
$$\frac{\begin{array}{c}(x : \tau_x \text{ as } \widehat{\tau}_x) \in \Gamma \qquad (x \mapsto \widehat{v}) \in \widehat{\sigma} \\ p = \widehat{\mathbf{prov\_of}}_{\Delta,\varsigma}(\widehat{\tau}_x, \widehat{v}) \qquad \widehat{\tau}_x/p = \{(p_b, \widehat{\tau}_b)\} \qquad (\widehat{\pi} \mapsto \pi_f \text{ as } \widehat{\tau}_f) \in \mathbf{Shatter}(\widehat{\tau}_b) \\ \pi = \pi_f.\pi' \qquad x_f \text{ fresh symbol} \qquad \tau_f = \mathbf{focus}\left(\pi_f, \tau_x\right) \qquad \widehat{v}_f = \widehat{\mathbf{focus}}_\varsigma\,(\widehat{\pi}, \widehat{v})\end{array}}{\Delta, \Sigma \vdash \Gamma, \widehat{\sigma}, \varsigma, (x.\pi : \tau \text{ as } \widehat{\tau}) \hookrightarrow \Gamma \cup \{x_f : (\tau_f \text{ as } \widehat{\tau}_f)\}, \widehat{\sigma} \cup \{x_f \mapsto \widehat{v}_f\}, \varsigma, (x_f.\pi' : \tau \text{ as } \widehat{\tau})}$$

Fig. 23. Evaluation judgment for source expressions, memory value construction. $\widehat{\mathbf{prov\_of}}(\widehat{\tau}, \widehat{v})$ explores all splits in $\widehat{\tau}$ and gathers the branches whose discriminant values are coherent with $\widehat{v}$.

$$\frac{y \notin \mathrm{dom}\,(\rho_{\mathrm{in}}) \cup \mathrm{dom}\,(\rho_{\mathrm{out}}) \qquad x \in \mathrm{dom}\,(\rho_{\mathrm{in}}) \qquad \rho_{\mathrm{in}}(x) = a.\widehat{\pi}_0}{\rho_{\mathrm{in}}, \rho_{\mathrm{out}}, \varsigma, \mathtt{let_{in}}\ y = x.\widehat{\pi}; \mathcal{E} \rightsquigarrow_T \rho_{\mathrm{in}} \cup \{y \mapsto a.\widehat{\pi}_0.\widehat{\pi}\}, \rho_{\mathrm{out}}, \varsigma, \mathcal{E}}$$

$$\frac{y \notin \mathrm{dom}\,(\rho_{\mathrm{in}}) \cup \mathrm{dom}\,(\rho_{\mathrm{out}}) \qquad x \in \mathrm{dom}\,(\rho_{\mathrm{out}}) \qquad \rho_{\mathrm{out}}(x) = a.\widehat{\pi}_0}{\rho_{\mathrm{in}}, \rho_{\mathrm{out}}, \varsigma, \mathtt{let_{out}}\ y = x.\widehat{\pi}; \mathcal{E} \rightsquigarrow_T \rho_{\mathrm{in}}, \rho_{\mathrm{out}} \cup \{y \mapsto a.\widehat{\pi}_0.\widehat{\pi}\}, \varsigma, \mathcal{E}}$$

$$\frac{x \notin \mathrm{dom}\,(\rho_{\mathrm{in}}) \cup \mathrm{dom}\,(\rho_{\mathrm{out}}) \qquad a \text{ fresh address}}{\rho_{\mathrm{in}}, \rho_{\mathrm{out}}, \varsigma, \mathtt{let_{out}}\ x = \mathtt{alloc}(\ell); \mathcal{E} \rightsquigarrow \rho_{\mathrm{in}}, \rho_{\mathrm{out}} \cup \{x \mapsto a.\epsilon\}, \varsigma \cup \{a \mapsto \__\ell\}, \mathcal{E}}$$

$$\frac{x \in \mathrm{dom}\,(\rho_{\mathrm{out}})}{\rho_{\mathrm{in}}, \rho_{\mathrm{out}}, \varsigma, x := c; \mathcal{E} \rightsquigarrow \rho_{\mathrm{in}}, \rho_{\mathrm{out}}, \varsigma[\rho_{\mathrm{out}}(x) \leftarrow c], \mathcal{E}}$$

$$\frac{(x \mapsto a_0.\widehat{\pi}) \in \rho_{\mathrm{out}} \qquad \left|\widehat{\mathbf{focus}}_\varsigma\,(\widehat{\pi}, \varsigma(a_0))\right| = \ell_0 \qquad a \text{ fresh address}}{\rho_{\mathrm{in}}, \rho_{\mathrm{out}}, \varsigma, x := \mathtt{\&alloc}(\ell); \mathcal{E} \rightsquigarrow \rho_{\mathrm{in}}, \rho_{\mathrm{out}}, \varsigma[a_0.\widehat{\pi} \leftarrow \&_{\ell_0}\,(a)] \cup \{a \mapsto \__\ell\}, \mathcal{E}}$$

$$\frac{x \in \mathrm{dom}\,(\rho_{\mathrm{in}}) \qquad y \in \mathrm{dom}\,(\rho_{\mathrm{out}}) \qquad \rho_{\mathrm{in}}(x) = a.\widehat{\pi} \qquad \widehat{\mathbf{focus}}_\varsigma\,(\widehat{\pi}, \varsigma(a)) = \widehat{v}}{\rho_{\mathrm{in}}, \rho_{\mathrm{out}}, \varsigma, y := x; \mathcal{E} \rightsquigarrow \rho_{\mathrm{in}}, \rho_{\mathrm{out}}, \varsigma[\rho_{\mathrm{out}}(y) \leftarrow \widehat{v}], \mathcal{E}}$$

$$\frac{x \in \mathrm{dom}\,(\rho_{\mathrm{out}})}{\rho_{\mathrm{in}}, \rho_{\mathrm{out}}, \varsigma, \mathtt{freeze}(x); \mathcal{E} \rightsquigarrow \rho_{\mathrm{in}} \cup \{x \mapsto \rho_{\mathrm{out}}(x)\}, \rho_{\mathrm{out}} \setminus x, \varsigma, \mathcal{E}}$$

$$\frac{x \in \mathrm{dom}\,(\rho_{\mathrm{out}}) \qquad \rho_{\mathrm{out}}(x) = a.\widehat{\pi} \qquad \widehat{\mathbf{focus}}_\varsigma\,(\widehat{\pi}, a) = \__\ell \qquad \ell = \ell_0 + \ell_1}{\rho_{\mathrm{in}}, \rho_{\mathrm{out}}, \varsigma, \mathtt{cast}\ x\ \mathtt{to}\ \__{\ell_0} \ltimes r : \__{\ell_1}; \mathcal{E} \rightsquigarrow \rho_{\mathrm{in}}, \rho_{\mathrm{out}}, \varsigma[a.\widehat{\pi} \leftarrow \__{\ell_0} \ltimes r : \__{\ell_1}], \mathcal{E}}$$

$$\frac{x \in \mathrm{dom}\,(\rho_{\mathrm{out}}) \qquad \rho_{\mathrm{out}}(x) = a.\widehat{\pi} \qquad \widehat{\mathbf{focus}}_\varsigma\,(\widehat{\pi}, a) = \__\ell \qquad \ell = \ell_1 + \cdots + \ell_n}{\rho_{\mathrm{in}}, \rho_{\mathrm{out}}, \varsigma, \mathtt{cast}\ x\ \mathtt{to}\ \left\{\!\!\left\{\__{\ell_1}, \ldots, \__{\ell_n}\right\}\!\!\right\}; \mathcal{E} \rightsquigarrow \rho_{\mathrm{in}}, \rho_{\mathrm{out}}, \varsigma[a.\widehat{\pi} \leftarrow \left\{\!\!\left\{\__{\ell_1}, \ldots, \__{\ell_n}\right\}\!\!\right\}], \mathcal{E}}$$

$$\frac{(x \mapsto a.\widehat{\pi}) \in \rho_{\mathrm{in}} \qquad (a \mapsto \widehat{v}) \in \varsigma \qquad \widehat{\mathbf{focus}}_\varsigma\,(\widehat{\pi}, \widehat{v}) = (c_i)_\ell}{\rho_{\mathrm{in}}, \rho_{\mathrm{out}}, \varsigma, \mathtt{switch}\,(x)\,\{c_1 \to \mathcal{E}_1\ \ldots\ c_n \to \mathcal{E}_n\} \rightsquigarrow_T \rho_{\mathrm{in}}, \rho_{\mathrm{out}}, \varsigma, \mathcal{E}_i}$$

Fig. 24. Evaluation judgment for target expressions: $\rho_{\mathrm{in}}, \rho_{\mathrm{out}}, \varsigma, \mathcal{E} \rightsquigarrow \rho'_{\mathrm{in}}, \rho'_{\mathrm{out}}, \varsigma', \mathcal{E}'$. Steps labelled with T indicate transitions that do not correspond to any $\rightsquigarrow$-step in the simulation used to prove correctness.