Rebuilding Algebraic Data Types from Mangled Memory Layouts

Gabriel Radanne, Thaïs Baudon, Laure Gonnord

Rebuilding Algebraic Data Types from Mangled Memory Layouts

Thaïs Baudon
Univ Lyon
Lyon, France
thais.baudon@ens-lyon.fr

Gabriel Radanne
Inria
France
gabriel.radanne@inria.fr

Laure Gonnord
UGA
Grenoble, France
laure.gonnord@grenoble-inp.fr

Abstract

Now integrated in mainstream languages, Algebraic Data Types (ADTs) have established themselves as a nice way to reason about data structures and their manipulations 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 the programmer 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, databases data-structures, or aggressively packed representations.

In this article, we propose a unified compilation procedure for ADTs constructors and destructors (i.e., pattern-matching) in the context of arbitrarily mangled memory layouts. We subsume existing compilation algorithms, and extend them to emit CFG-style programs with explicit memory allocation and full support for recursive types.

Keywords: Algebraic Data Types, Pattern Matching, Compilation, Data Layouts

1 Introduction

Algebraic Data Types 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 not bothered 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 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 Algebraic Data Types: 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 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 Algebraic Data Types, 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 Algebraic Data Types 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

Conference'17, July 2017, Washington, DC, USA

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|1:10|11|12:7]				rd	opcode	J-type

Figure 1. RISCV Core instruction format, excerpt.
"rs1,2" are source registers, "rd" a destination register. "imm[x:y]" means "y bits starting from x in the binary representation of imm". "imm[y|x]" means "bits y, then x 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

Figure 2. Instruction semantics and encoding, excerpt.

```
1  enum Reg { X0, X1, X2, X3, X4, X5, X6, X7, ... } // cut
2  // represented as a C-like enum on 5 bits
3
4  enum Op {
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  }
10 represented as split .[0:7] {// discriminant in the 7 lowest bits
11   | 0x23 from Sw =>
12     w32 with .[0:7]:(= 0x23) // opcode
13         with .[7:5]:(_.Sw.2.[0:5] as w5) // 5 lowest bits from imm
14         with .[12:3]:(= 2) // funct3
15         with .[15:5]:(_.Sw.0 as Reg32) // rs1
16         with .[20:5]:(_.Sw.1 as Reg32) // rs2
17         with .[25:7]:(_.Sw.2.[5:7] as w7) // 7 highest bits from imm
18   | // other cases cut for readability
```

Figure 3. Ribbit modelling (source types and memory layouts) for RISCV Instructions. iN (resp. wN) types are predefined as "int (resp. words) on N bits". Ranges are encoded from a left bound and a size.

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 *deconstruct* values. While innocuous at first, this severely limits highly mangled layouts where data needs to be deconstructed, reshaped, and rebuilt differently, as is the case for aggressive struct packing, flattening or AoS/SoA transformations. To better understand this limitation, let us study a real world example of an ADT with a complex layout: the RISC-V instruction set with its binary representation.

1.1 Real World ADTs: the RISC-V instruction set

To demonstrate the complexity of real-world memory layouts for ADTs, we consider a restricted version of the 32-bit RISC-V assembly language consisting of four instructions (`add`, `addi`, `sw`, and `jal`). We will use Ribbit's DSL to specify the layout as 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 depicted in Fig. 1, RISC-V 32-bit instructions have different formats w.r.t. their addressing mode. Further characteristics of our four instructions are depicted in Fig. 2.

Already, we see complications: in general, an instruction information (type, instruction name, involved registers, ...) is split 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 can not be readily extracted from the binary representation. For our particular (simple) subset : (i) the four instructions are distinguishable from their `opcode` *only*, bits 0 to 7. (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 on bit ranges 7-11 and 25-31. (iv) the 20-bits immediate value for the `jal` instruction can be recovered from bits 12 to 31 but we need to *rebuild* this immediate from four ranges of bits.

We demonstrate the modelling of RISC-V registers and instructions with ADTs in Fig. 3, using the Ribbit [Baudon et al. 2023] syntax. 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 word for "2"). Instructions (`Op` type) are encoded on 32 bits (`w32`). `split`, on Line 10, allow distinguishing the different cases using the 7 lowest bits (opcode). We only showcase the `Sw` case of the split. Line 12 specifies that the opcode is `0x23` (see Fig. 2). The immediate operand is split in two parts, which are encoded in bits 7 to 11 inclusive (Line 13) and 25 to 31 (Line 17) within the 32-bit word representing the full instruction.

1.2 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. 4 corresponds to `Sw(X1, X2, imm)`. This is where we hit the previously mentioned limitation: since `imm` is stored non consecutively , Baudon et al. [2023]'s algorithm is unable to identify the high and low bits and can't generate this code.

Similarly, Ribbit allow definitions of *pattern-matching* functions, such that the one in Fig. 5 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` representation while compiling the pattern and need to first find all the right pieces and combine them together. Baudon et al. [2023]'s algorithm is thus not capable of compiling the `is_compactable` function.

In this article, we extend their algorithm, and Ribbit itself, to handle such cases by *conjointly compiling constructors and destructors*. Our new compilation algorithm, when applied to the `is_compactable` function, generates code as a control flow graph shown in Fig. 6. This generated code:

1. *inspects* the internal representation of an input `Op` value to determine its head constructor (`Add`, `Addi`, `Jal` or `Sw`), as well as the nested register constructor in `Jal`;
2. *extracts* from this representation all subterms that are bound to variables in the matched pattern (e.g. parts of

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

Figure 4. Writes operations for `Sw(X1, X2, imm)`
Code which builds the appropriate representation in the root memory location o.

```
match_fn is_compactable : Op -> bool
  Add(rd, rs1, rs2) => rd == rs1 && rs1 != X0 && rs2 != X0,
  Addi(rd, rs, imm) => rd == rs && rs != X0 && imm[6:5] == 0,
  Jal(X1, imm) => imm[12:7] == 0,
  Sw(rs1, rs2, imm) => X7<rs1,rs2<X16 && imm[0:2] == imm[7:4] == 0,
  _ => false
```

Figure 5. Function determining whether a given 32-bit instruction can be compressed into a 16-bit one, in simplified Ribbit syntax. Conditions taken from [Waterman et al. 2019, chapter RISCV-C].

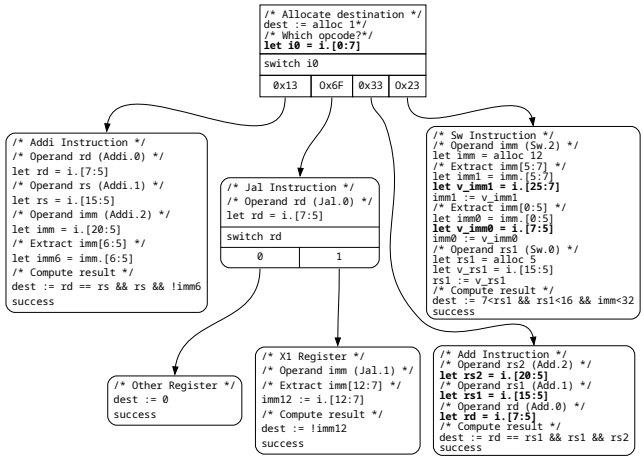

Figure 6. Generated code for `is_compactable`.
From the input `i`, it distinguish head constructors using the 7 lowest bits, then extracts subterms such as destination and source registers for `Add`, and the 12 bits `imm` for `Sw` (in bold). The result is in `dest`.

the immediate `imm` for `Jal`, the three registers operands `rd`, `rs1`, `rs2` for `Add`);

3. *allocates* and *initialises* memory to represent the value of the expression on the right-hand side.

We contribute a general procedure which compiles (potentially nested) constructor expressions, pattern matching, and *both* together, without introducing superfluous work, to a destination passing intermediary representation. This procedure emits precise memory allocation code and handles recursive types and recursive code emission.

Section 2 describes our input language based on types and memory layouts from Baudon et al. [2023]. Section 3 presents our target intermediate representation, in Destination Passing Style [Shaikhha et al. 2017]. Our approach relies on existing pattern matching compilation techniques, which we detail in Section 4. Our main contribution, detailed in Section 5, is a compilation algorithm for expressions constructing ADT values according to custom memory layouts.

2 Algebraic Data Types and Their Layouts

We now briefly formalise our input language. As in Section 1, we uses a two-tiered view: *algebraic data types* used for programming and *memory layouts* detailing how to represent them in memory. We then present a core programming language to manipulate such types. The bulk of the specification extends Baudon et al. [2023], with most of the limitations imposed on memory types lifted and a larger input language.

2.1 Algebraic Data Types

The grammar for Algebraic Data Types is presented in Fig. 7. We denote types using τ and type variables with $t \in \mathcal{V}_{\text{ty}}$. We denote all tuples with angle brackets, for instance $\langle i32, f64 \rangle$ for pairs of a 32-bit integers and a 64-bit float. Constructors of sums are marked with a capital letter, for instance $\text{Some}(t) + \text{None}$ is an option type. In examples, we use K as shortcut for $K(\langle\rangle)$. We use Γ to denote type environments, i.e., maps from type variables t to types τ. We also define two related constructs: provenances and paths. Provenances give partial information on what a value looks like: a type, a constructor, or anything (wildcard: _). Paths precisely indicate the position of a subterm in a given type or provenance, using accesses by tuple position $.i$ or by constructor $.K$.

Example 2.1 (Source type)**.** Our source type for RISC-V 32-bit instructions is a sum type with four constructors: $\tau_{\text{RISC-V}} = \text{Add}(\tau_{\text{reg}}, \tau_{\text{reg}}, \tau_{\text{reg}}) + \text{Addi}(\tau_{\text{reg}}, \tau_{\text{reg}}, i12) + \text{Jal}(\tau_{\text{reg}}, i20) + \text{Sw}(\tau_{\text{reg}}, \tau_{\text{reg}}, i12)$. The path .Add.0 designates the first argument of an `Add` instruction, namely its destination register. The provenance Sw(_, _, _) designates any value of type $\tau_{\text{RISC-V}}$ whose constructor is `Sw`.

2.2 Memory layouts

Each algebraic data type is associated with a *memory layout*, whose grammar is given in Fig. 8, which specifies how its values should be represented in memory. As a general convention, memory elements are distinguished with a hat. Memory layouts, denoted by $\widehat{\tau}$ (See Example 2.2) consist of concrete memory structures (words, pointers and structs/blocks) as well as constructs that refer back to the represented high level type (fragments refer to subterms, while splits create disjunctions between constructors). Fragments may refer to arbitrarily nested subterms; so-called *regular* layouts only contain fragments that refer to immediate subterms (i.e., the contents of a constructor for sums or a field for products). Memory paths, denoted $\widehat{\pi}$, indicate positions in layouts.

Example 2.2 (Memory layout)**.** The layout associated with the Ribbit example of Figure 3 is:

$$\widehat{\tau}_{\text{RISC-V}} = \text{Split}\,(.[0:7]) \left\{ \begin{array}{lllll} \texttt{0x33} & \text{from} & \text{Add} & \Rightarrow & \widehat{\tau}_{\text{Add}} \\ \texttt{0x23} & \text{from} & \text{Sw} & \Rightarrow & \widehat{\tau}_{\text{Sw}} \\ \ldots & \text{from} & \ldots & \Rightarrow & \ldots \end{array} \right\}$$

The split describes how to distinguish between constructors, by inspecting the 7 lowest bits (`0xXX` are fixed constants).

$$
\begin{aligned}
\tau \in \mathit{Types} ::= {} & t \in \mathcal{V}_{\text{ty}} && \text{(type variable)}\\
\mid {} & i\ell && \text{(primitive integer type)}\\
\mid {} & \langle \tau, \ldots, \tau \rangle && \text{(tuple/product type)}\\
\mid {} & K(\tau) + \ldots + K(\tau) && \text{(sum type)}\\
\Gamma : {} & \mathcal{V}_{\text{ty}} \to \mathit{Types} && \text{(type environment)}\\
p \in \mathit{Provs} ::= {} & \_ \mid \langle p, \ldots, p \rangle \mid K(p)\\
\pi \in \mathit{Paths} ::= {} & \epsilon \mid \pi.i \mid \pi.K
\end{aligned}
$$

Figure 7. Algebraic Data Types

$$
\begin{aligned}
\widehat{\tau} \in \widehat{\mathit{Types}} ::= {} & \widehat{t} \in \widehat{\mathcal{V}_{\text{ty}}} && \text{(variable)}\\
\mid {} & (\pi \text{ as } \widehat{\tau}) && \text{(fragment)}\\
\mid {} & (= \mathrm{c}) && \text{(fixed immediate)}\\
\mid {} & W_\ell \underset{1 \le i \le n}{\bowtie} [o_i : \ell_i] : \widehat{\tau}_i && \text{(word)}\\
\mid {} & \&_{\ell,a}\,(\widehat{\tau}) \underset{1 \le i \le n}{\bowtie} [o_i : \ell_i] : \widehat{\tau}_i && \text{(pointer)}\\
\mid {} & \{\!\{\widehat{\tau}_1, \ldots, \widehat{\tau}_n\}\!\} && \text{(struct)}\\
\mid {} & \text{Split}\,(\widehat{\pi}) \begin{Bmatrix} \mathrm{c}_1 & \text{from} & P_1 & \Rightarrow & \widehat{\tau}_1 \\ & & \ldots & & \\ \mathrm{c}_n & \text{from} & P_n & \Rightarrow & \widehat{\tau}_n \end{Bmatrix} && \text{(split)}\\
\widehat{\Gamma} : {} & \widehat{\mathcal{V}_{\text{ty}}} \to \widehat{\mathit{Types}} && \text{(memory type environment)}\\
\widehat{\pi} \in \widehat{\mathit{Paths}} ::= {} & \epsilon && \text{(empty path)}\\
\mid {} & \widehat{\pi}.[o:\ell] && \text{(bit range: } \ell \text{ bits from offset } o)\\
\mid {} & \widehat{\pi}.* && \text{(pointer dereferencing)}\\
\mid {} & \widehat{\pi}.i && \text{(struct field access)}
\end{aligned}
$$

Figure 8. Memory Layout

$$
\begin{aligned}
e \in \mathit{Exprs} ::= {} & \pi \in \mathit{Paths} && \text{(binding)}\\
\mid {} & z \in \mathbb{Z} \mid \ldots && \text{(constant)}\\
\mid {} & \langle e, \ldots, e \rangle && \text{(tuple)}\\
\mid {} & K(e) && \text{(constructor)}\\
m \in \mathit{Matches} ::= {} & \{p_1 \to e_1; \ldots; p_n \to e_n\}
\end{aligned}
$$

Figure 9. Program syntax

Layouts for individual instructions (e.g., Sw) are built from a W_{32} with additional constraints on subranges of bits:

$$
\begin{aligned}
\widehat{\tau}_{\text{Sw}} = {} & W_{32} \ltimes [0:7] : (= \texttt{0x23}) && \text{(opcode const)}\\
\ltimes {} & [7:5] : (.\text{Sw.0 as } \widehat{\tau}_{\text{reg}}) && \text{(base register)}\\
\ltimes {} & [12:3] : (= 2) \ltimes \ldots && \text{(func3 const, etc.)}
\end{aligned}
$$

The fragment construct $(.\text{Sw.0 as } \widehat{\tau}_{\text{reg}})$ expresses that bits 7 to 12 inclusive contain the representation of the first argument of Sw (base register) according to the $\widehat{\tau}_{\text{reg}}$ layout.

$$
\begin{aligned}
\mathcal{K} ::= {} & \texttt{fail} \mid \texttt{success} && \text{(return statements)}\\
\mid {} & \texttt{call}\ f(i,o)\ ;\ \mathcal{K} && \text{(function application)}\\
\mid {} & \texttt{let in}\ i = i.\widehat{\pi}\ ;\ \mathcal{K} && \text{(input location binding)}\\
\mid {} & \texttt{let out}\ o = o.\widehat{\pi}\ ;\ \mathcal{K} && \text{(output location binding)}\\
\mid {} & o := \text{rhs}\ ;\ \mathcal{K} && \text{(write to output location)}\\
\mid {} & \texttt{switch}\ i\ \{(\mathrm{c} \to \mathcal{K})*, (\_ \to \mathcal{K})?\} && \text{(switch)}\\
\text{rhs} ::= {} & \mathrm{c} && \text{(immediate value)}\\
\mid {} & i && \text{(input location contents)}\\
\mid {} & \texttt{alloc}\ \ell && \text{(pointer to } \ell \text{ newly allocated bits)}\\
\mathcal{F} ::= {} & \{f(i,o) \to \mathcal{K}\} && \text{(toplevel functions)}
\end{aligned}
$$

Figure 10. Target IR

2.3 Programs

We now define high-level programs that manipulate algebraic data types. Our general goal is to compile such programs to low-level programs that manipulate memory directly, with respect to the specified memory layout. For the purpose of this presentation, we only consider program parts which are directly related to algebraic data types: pattern matching, which destructs values, and simplified expressions, which construct values. Crucially, we consider both *in conjunction*, as shown in Fig. 9, with the usage of *matches*. A match m is composed of a list of *cases*; cases being themselves composed of a provenance on the left-hand side, and an expression on the right-hand side. Expressions are composed of tuples, constructors, and constants. Instead of variables, expressions refer to positions in the input value via *paths*.

Example 2.3 (Destination register binding)**.** The following match extracts the destination register of a RISC-V instruction if it exists, and returns X_0 otherwise:

$$
\text{get\_dest} = \begin{Bmatrix} \text{Add}(r,\_,\_) \to r & \text{Addi}(r,\_,\_) \to r \\ \text{Jal}(r,\_) \to r & \text{Sw}(\_,\_,\_) \to X_0 \end{Bmatrix}
$$

3 Target in Destination Passing Style

We now define our target intermediate representation in Fig. 10. The goal of this program representation is to make the following tasks explicit: switching on values, writing results to their appropriate memory location, and, crucially, allocating and initialising memory on the heap to properly represent the output expression. we depart from [Baudon et al. 2023] and define a new IR in Destination Passing Style [Shaikhha et al. 2017] (See Example 3.1). Our IR expresses traditional decision trees via a `switch` construct with a default branch marked by "_", along with `success` and `fail` return statements. For the remaining constructs, we will use the notion of *locations*: these are unaligned pointers which can be defined using memory paths and subpaths, and be passed as arguments to functions. They will be filled with the appropriate memory representation by the compiled code.

Input locations, ("i"), denote read-only memory representations of input values (at toplevel) or sub-input values (during computation). More precisely, root locations are provided before-hand as arguments. Input sub-locations are obtained by focusing an existing input location with a memory path, using the instruction `let in` $i' = i.\widehat{\pi}$; *Output locations*, ("o"), denote write-only memory locations that will be (eventually) allocated and computed during the execution of the target. These outputs (or sub-outputs obtained with `let out ... ;`) can be filled, using the `write` instruction, with several kinds of values: constants denoted c, the contents of an input location, or the address of newly allocated memory of a given size denoted `alloc` ℓ. Finally, our IR enables function calls thanks to the instruction `call` $f(i, o)$; . The list of functions is defined toplevel by the $\mathcal{F}$ environment. These functions will be denoted with a plain frame around them in the rest of the paper. Note that return statements, namely `success` and `fail`, do not return any value. A match will be compiled to a set of functions taking input and output locations as arguments, as shows Example 3.1. 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 3.1 (target IR for our example)**.** The program from Example 2.3, taking an input i of type $\tau_{\text{RISC-V}}$ and returning an output o of type τ_{reg}, will be compiled into:

```
fun aux(i, o) → {
  let in i' = i.[7:5] ; o := i' ; success
}
fun get_dest(i, o) → {
  let in i0 = i.[0:7] ;
  switch i0 {
    0x33 → call aux(i, o) ; success
    0x13 → call aux(i, o) ; success
    0x6F → call aux(i, o) ; success
    0x23 →      o := 0 ; success
  }
}
```

The compiled function get_dest considers its input i as a W_{32} value, from which it extracts its 7 lowest bits into a new variable i_0. From i_0's value it either directly write 0 as the result o (it has recognised `Sw`), or call the auxiliary function aux, which outputs to destination o 5 bits corresponding to the destination register of instructions `Add/Addi/Jal` (Fig. 1, `rd` is always located at the same place).

4 A Primer On Matching Compilation

Before presenting our full algorithm, we detail the required tooling. We first briefly summarise an existing compilation procedure for pattern matching, then define an operation to semantically explore a type conjoined with its layout.

4.1 From Patterns to Switch Trees

General procedures for pattern matching compilation [Maranget 2008; Sestoft 1996] 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 as the ones we consider, Baudon et al. [2023] provides a layout-aware compilation procedure to decision DAGs.

Our goal is to extend such a compilation procedure to properly handle binders and right-hand-side expressions. We will thus extend a traditional pattern matching compilation approach. Baudon et al. [2023] provides a CompileMatch procedure which we will use as a black box in the rest of this article, adapted to our formalism. CompileMatch takes as arguments a pattern match *without binders nor right-hand-sides*, i.e., a list of provenances, along with the type and layout of the input data. It can readily handle nested provenances. In the context of this article, we reuse this procedure to take as input a list of provenances paired with their right-hand-side target IR, and emit decision DAGs using the target IR defined in Section 3. We illustrate on an example.

Example 4.1 (Matching Compilation)**.** We consider the "matching part" of the functions defined in Example 2.3:

$$\textsc{CompileMatch}\left(i, \tau_{\text{RISC-V}}, \widehat{\tau}_{\text{RISC-V}}, \left\{\begin{array}{ll} \text{Jal}(\_,\_) & \Rightarrow \mathcal{K}_1 \\ \text{Sw}(\_,\_,\_) & \Rightarrow \mathcal{K}_2 \\ \text{Addi}(\_,\_,\_) & \Rightarrow \mathcal{K}_3 \\ \text{Add}(\_,\_,\_) & \Rightarrow \mathcal{K}_4 \end{array}\right\}\right) = \boxed{\begin{array}{l} \texttt{let in } i' = i.[0{:}7] \texttt{ ;} \\ \texttt{switch } i' \texttt{ \{} \\ \quad \texttt{0x6F} \rightarrow \mathcal{K}_1 \\ \quad \texttt{0x23} \rightarrow \mathcal{K}_2 \\ \quad \texttt{0x13} \rightarrow \mathcal{K}_3 \\ \quad \texttt{0x33} \rightarrow \mathcal{K}_4 \\ \texttt{\}} \end{array}}$$

This switch is the toplevel node of the code in Fig. 6.

4.2 Exploring Layouts with Focus and Specialise

To compile high-level patterns and 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 switches required to access those subterms, represented as fragments. This exploration will be driven by *provenances* since they form the common backbone of our high-level language constructs, including expressions and types, and already drive the existing CompileMatch procedure. Our goal is thus to define a function Explore which takes a provenance p, a type τ and a layout $\widehat{\tau}$, and returns the list of all accessible sub-elements in τ represented as $\widehat{\tau}$ which are "compatible" with p. Semantically, a branch characterises the values of type τ that match the provenance p. A branch is thus defined as quadruplet $(p_i, \tau_i, \widehat{\tau}_i, F_i)$ consisting of the provenance, type and layout refined for that specific branch, and of a list of fragments contained therein.

Example 4.2. We can explore $\widehat{\tau}_{\text{RISC-V}}$ with the provenance $\text{Sw}(\_)$: $\textsc{Explore}(\text{Sw}(\_), \tau_{\text{RISC-V}}, \widehat{\tau}_{\text{RISC-V}}) = \{(\text{Sw}(\_), \tau_{\text{Sw}}, \widehat{\tau}_{\text{Sw}}, F)\}$

where $\tau_{\text{Sw}} = \text{Sw}(\tau_{\text{reg}}, \tau_{\text{reg}}, i12)$

and $F = \left\{ \begin{array}{l} (.[7:5] \mapsto (\quad .\text{Sw.2}.[0:5] \text{ as } W_5)) \\ (.[15:5] \mapsto (\quad .\text{Sw.0 as } \widehat{\tau}_{\text{reg}})) \\ (.[20:5] \mapsto (\quad .\text{Sw.1 as } \widehat{\tau}_{\text{reg}})) \\ (.[25:7] \mapsto (\quad .\text{Sw.2}.[5:7] \text{ as } W_7)) \end{array} \right\}$.

Note that the type and layout of the (unique) branch F are *refined* according to its provenance Sw(_): τ_{Sw} and $\widehat{\tau}_{\text{Sw}}$ only capture values of the form Sw(_).

To precisely define Explore, we need two new operations for refining types and layouts: focusing and specialisation.

Focusing in Types and Layouts. "Focusing" 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 $\textbf{focus}(\pi, \theta)$ 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 $\tau_{\text{RISC-V}}$:

$$\textbf{focus}(.\text{Sw.0}, \tau_{\text{RISC-V}}) = \textbf{focus}\big(.\text{Sw.0}, \text{Sw}(\tau_{\text{reg}}, \tau_{\text{reg}}, i12)\big) = \tau_{\text{reg}}$$

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

Layout and Type Specialisation. "Specialisation" filters a type or a layout to exclude parts which are incompatible with a given provenance. Type specialisation, denoted τ/p, is a simple syntactic filter that discards irrelevant constructors. For instance, $\tau_{\text{RISC-V}}/\text{Sw}(\_,\_,\_) = \text{Sw}(\tau_{\text{reg}}, \tau_{\text{reg}}, i12)$. Layout specialisation, denoted $\widehat{\tau}/p$, 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, $\widehat{\tau}_{\text{RISC-V}}/\text{Sw}(\_,\_,\_)$ 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:

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

Explore. We are now ready to properly define Explore, in Algorithm 1. 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, use python-style generators using the "**yield**" keyword, and "for-each" style loops.

Using the specialisation $\widehat{\tau}_0/p_0$, we get all refinements pairs of $\widehat{\tau}_0$ compatible with p_0: each refinement pair is characterised by 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 τ_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.

```
1 function Explore(p_0, τ_0, τ̂_0):
2   for (p ↦ τ̂) ∈ τ̂_0/p_0 do
3     τ ← τ_0/p
4     F ← { π̂' ↦ τ̂_π̂' | focus^(π̂', τ̂) = (π' as τ̂') = τ̂_π̂' }
5     yield (p, τ, τ̂, F)
```

Algorithm 1: Explore

```
Data: ⟨i, τ_i, τ̂_i, p_i⟩ the input description
Data: o the output location
Data: π the path in the input to the desired value
Result: Code binding o to the memory value at
        position π in the input
1 function Extract(⟨i, τ_i, τ̂_i, p_i⟩, o, π):
2   if π = ϵ then
3     return [o := i ; success]
4   else
5     B ← for p_b, τ_b, τ̂_b, F_b ∈ Explore(p_i, τ_i, τ̂_i) do
6       if ∃(π̂_f ↦ (π_f as τ̂_f)) ∈ F_b, π_f ≤ π then
7         i' ← fresh symbol
8         π', τ'_i, p'_i ←
            focus(π_f, π), focus(π_f, τ_b), focus(π_f, p_b)
9         K ← [let in i' = i.π̂_f ;
               Extract(⟨i', τ'_i, τ̂_f, p'_i⟩, o, π')]
10      else K ← [fail]
11      yield (p_b, K)
12    return [CompileMatch (i, τ̂_i, B)]
```

Algorithm 2: Naive compilation procedure

4.3 A Naive Compilation Algorithm

Before diving into the full compilation algorithm, and as a general warm-up to compilation of pattern matching to our target IR, we showcase how to use Explore to easily implement the cases handled in Baudon et al. [2023]. The Extract procedure, defined in Algorithm 2, handles bound variables that exactly correspond to a single fragment within the memory layout, and therefore do not require converting between different layouts. For instance, the destination register at .Sw.0 in our running example directly corresponds to a single fragment in the layout $\widehat{\tau}_{\text{RISC-V}}$ (at position $.[0:7]$). This algorithm does not cover cases which require rebuilding a value from pieces, such as the offset at .Sw.2.

More precisely, Extract takes a description $\langle i, \tau_i, \widehat{\tau}_i, p_i \rangle$ of the input value, an output location o and a path π. It emits code to store in o the representation of the subterm located at π within the input value. In Line 2-3, if π is the empty path, then the input and output values are exactly the same and we simply copy the contents of i to o, then succeed. Otherwise, the emitted code must handle every possible branch of $\widehat{\tau}_i$: we collect them in B using Explore on Line 3. We will then emit

code that dynamically determines the appropriate branch by inspecting the input value using the pattern matching compilation algorithm CompileMatch on Line 12. For each branch of the input layout, we search for a fragment covering a prefix of π: this fragment necessarily contains the data at position π. If found, we obtain, on Line 8, focused path, types, and provenances for this fragment and bind its location $i.\widehat{\tau}_f$ on Line 9, and then recursively attempt to extract the desired value on Line 10. If no such fragment exists, then the output value is either not covered by this layout or broken into multiple pieces in separate fragments, we thus fail. In both cases, we emit a case for CompileMatch.

Example 4.3 (Extract – Algorithm 2)**.** Let us consider again the destination register of a Sw instruction at position .Sw.0. Let o a fresh output location and i an input location assumed to contain the representation of a Sw instruction. Extract $(\langle i, \tau_{\text{RISC-V}}, \widehat{\tau}_{\text{RISC-V}}, \text{Sw}(\_)\rangle, o, .\text{Sw}.0)$ starts by exploring $\widehat{\tau}_{\text{RISC-V}}$ as in Example 4.2, keeping only branches that match Sw and yielding a single branch $(\text{Sw}(\_), \tau_{\text{Sw}}, \widehat{\tau}_{\text{Sw}}, F)$ where $\tau_{\text{Sw}} = \text{Sw}(\tau_{\text{reg}}, \tau_{\text{reg}}, i12)$

and $F = \left\{ \begin{array}{l} (.[7{:}5] \mapsto (\quad .\text{Sw}.2.[0{:}5] \text{ as } W_5)) \\ (.[15{:}5] \mapsto (\quad .\text{Sw}.0 \text{ as } \widehat{\tau}_{\text{reg}})) \\ (.[20{:}5] \mapsto (\quad .\text{Sw}.1 \text{ as } \widehat{\tau}_{\text{reg}})) \\ (.[25{:}7] \mapsto (\quad .\text{Sw}.2.[5{:}7] \text{ as } W_7)) \end{array} \right\}$.

F contains the fragment $(.[15:5] \mapsto (\_.\text{Sw}.0 \text{ as } \widehat{\tau}_{\text{reg}}))$, which covers _.Sw.0. We can now focus on this fragment and proceed with the recursive call. Let i' a fresh symbol, $\tau' = \textbf{focus}(.\text{Sw}.0, \tau_{\text{Sw}}) = \tau_{\text{reg}}$, $p' = \textbf{focus}(.\text{Sw}.0, \text{Sw}) = \_$ and $\pi' = \textbf{focus}(.\text{Sw}.0, .\text{Sw}.0) = \epsilon$. We have the recursive call Extract $\left(\langle i', \tau', \widehat{\tau}_{\text{reg}}, p'\rangle, o, \pi'\right) = \boxed{o := i' \texttt{ ; success}}$ then $\mathcal{K} = \boxed{\texttt{let in } i' = i.[15{:}5] \texttt{ ; } o := i' \texttt{ ; success}}$. We yield the case $(\text{Sw}(\_), \mathcal{K})$ and can finally compute the full code with CompileMatch$(i, \tau_{\text{RISC-V}}, \widehat{\tau}_{\text{RISC-V}}, \{(\text{Sw}, \mathcal{K})\}) =$

$$\boxed{\begin{array}{l} \texttt{let in } i_0 = i.[0{:}7] \texttt{ ;} \\ \texttt{switch } i_0 \left\{ \begin{array}{l} \texttt{0x23} \rightarrow \texttt{let in } i' = i.[15{:}5] \texttt{ ;} \\ \qquad\qquad\quad o := i' \texttt{ ; success} \\ \_ \rightarrow \texttt{fail} \end{array} \right\} \end{array}}$$

Using all these tools, we were able to concisely express a not-so-simple procedure which Explores each branch of a layout, focuses on their constituents, then combines their compiled versions using CompileMatch. Similarly to Baudon et al. [2023]'s development, this procedure only handles paths corresponding to a whole fragment. Consequently, this procedure doesn't need to allocate, and always terminates. In the rest of this article, we will detail how to handle the full range of patterns and expressions.

5 Compilation of Constructor Expressions

Our main contribution is a general procedure, implemented into *Ribbit*, which can compile both patterns and expressions in a unified manner for arbitrary layouts. This procedure can thus compile a complete pattern matching branch with arbitrary bindings, or a standalone constructor expression.

This section presents this procedure in several steps: first, Section 5.1 presents a restricted version which handles reading from input locations and writing to output locations but leaves out memory allocation; then, Section 5.2 details how to allocate memory precisely at the right time; finally, Section 5.3 describes how to ensure termination of our algorithm and emit recursive code when necessary and demonstrates the procedure on such an example.

5.1 Seek And Rebuild

The Extract procedure presented in Section 4.3 handles simple cases where the wanted piece of data is a single fragment, corresponding to so-called "*regular*" layouts from Baudon et al. [2023]. In the general case, we want to rebuild arbitrary expressions and recover data from arbitrarily nested and scattered fragments, such as the immediate operand of the Sw instruction of our running example.

Our compilation algorithm consists of two mutually recursive procedures that emit code for a given pattern matching branch. The first emits code which Rebuilds (Algorithm 4) the necessary pieces to assemble a target expression e. Expressions consist of fixed parts (constructors and constants) and of variable parts from the input. We then need to emit code to Seek (Algorithm 3) such variable parts within the input identified by their position π. (i.e., if such a piece is wholly available, return it, otherwise, Rebuild it from smaller pieces, etc.) The main ideas behind these algorithms are:

- Alternatively explore and rebuild input and output values, using Seek (Algorithm 3) and Rebuild (Algorithm 4).
- For variables, which are identified by their position in the input, Seek tries to find the corresponding piece directly within the input value, similarly to Extract. Otherwise we use Rebuild to break it into smaller pieces, until we reach individual bits of numeric values, which are necessarily somewhere in the input (assuming a correct layout).
- Rebuild uses *fragments* from the specified layout to guide the search for smaller pieces, and fills the rest using constants gathered from the output layout.
- This initial version *doesn't allocate anything*. This will be addressed in the next section.

Let us now look at these algorithms in more detail. For each pattern matching branch $(p \rightarrow e)$ we aim to compile, we refer to the left-hand-side as the *input value*, identified by the tuple $\text{args}_{\text{in}} = \langle i_{\text{in}}, \tau_{\text{in}}, \widehat{\tau}_{\text{in}}, p_{\text{in}}\rangle$ composed of its input location, type, layout and provenance respectively. Initially, $p_{\text{in}} = p$. Similarly, we refer to the value computed by the right-hand side expression as the *output value* and identify it with the tuple $\text{args}_{\text{out}} = \langle o_{\text{out}}, \tau_{\text{out}}, \widehat{\tau}_{\text{out}}\rangle$ composed of its output location, type and layout respectively. Our goal is to emit code that writes to o_{out} the representation of the target value according to $\widehat{\tau}_{\text{out}}$, and reads all data needed for doing so from i_{in}, which contains the input value represented according to

Data: $\text{args}_{\text{in}} = \langle i_{\text{in}}, \tau_{\text{in}}, \widehat{\tau}_{\text{in}}, p_{\text{in}} \rangle$ the input description
Data: $\text{args}_{\text{out}} = \langle o_{\text{out}}, \tau_{\text{out}}, \widehat{\tau}_{\text{out}} \rangle$ the output description
Data: π the path in the input to the desired value
Result: Code binding the memory value at position π in the input

1 **function** SEEK($args_{in}, args_{out}, \pi$):
2 **if** $\pi = \epsilon \wedge \widehat{\tau}_{in} = \widehat{\tau}_{out}$ **then**
  // Input and output representations are the same, we return.
3 **return** $\boxed{o_{out} := i_{in} \ ; \ \texttt{success}}$
4 **else** // Otherwise, Explore all cases.
5 $B \leftarrow$ **for** $p_b, \tau_b, \widehat{\tau}_b, F_b \in \text{EXPLORE}(p_{in}, \tau_{in}, \widehat{\tau}_{in})$ **do**
  // Seek a fragment containing the piece of data at π.
6 **if** $\exists (\widehat{\pi}_f \mapsto (\pi_f \text{ as } \widehat{\tau}_f)) \in F_b \wedge \pi_f \preceq \pi$ **then**
  // Found one. We focus on it and search inside.
7 $\pi' \leftarrow \textbf{focus}(\pi_f, \pi)$
8 $i \leftarrow$ fresh symbol
9 $\text{args}'_{\text{in}} \leftarrow \langle i, \textbf{focus}(\pi_f, \tau_b), \widehat{\tau}_f, \textbf{focus}(\pi_f, p_b) \rangle$
10 $\mathcal{K} \leftarrow \boxed{\begin{array}{l}\texttt{let in } i = i_{\text{in}}.\widehat{\pi}_f \ ; \\ \text{SEEK}\left(\text{args}'_{\text{in}}, \text{args}_{\text{out}}, \pi'\right)\end{array}}$
11 **else** // Otherwise, Rebuild from smaller pieces.
12 $\text{args}'_{\text{in}} \leftarrow \langle i, \tau_b, \widehat{\tau}_b, p_b \rangle$
13 $\mathcal{K} \leftarrow \boxed{\text{REBUILD}(\text{args}'_{\text{in}}, \text{args}_{\text{out}}, \pi)}$
14 **yield** $(p_b, \mathcal{K})$
  // Assemble the code of these branches via a decision tree.
15 **return** $\boxed{\text{COMPILEMATCH}\ (i_{in}, \widehat{\tau}_{in}, B)}$

Algorithm 3: SEEK

$\widehat{\tau}_{\text{in}}$. Both algorithms follow the general shape demonstrated by Section 4.3 with a base case (the empty path for SEEK, and constants for REBUILD) followed by a call to EXPLORE and to COMPILEMATCH. SEEK is mostly similar to EXTRACT. Crucially, it only examines the input value. REBUILD is more complex, and aims to build two pieces of code: $\mathcal{K}_{\text{consts}}$ which populates o_{out} with appropriate constants, notably coming from constructors, and $\mathcal{K}_{\text{frags}}$ which fills it with pieces corresponding to fragments of the output layout.

Both algorithms need to maintain a precise description of the current case under scrutiny. Indeed, after a few recursive calls, we might be exploring deep in the input and output layouts. This description is represented by the provenances p_{in} and p_{out}. Naturally, both provenances share subparts, namely the places corresponding to variables in e. As we explore the output in REBUILD, we need to share the refined information between input and output provenances. This is the role of REMAP (Algorithm 5): given two provenances p_l and p_r and a map of shared positions between both sides, it creates a new provenance which is at least as precise as p_l, but contains shared information from p_r. It is used in both directions, first to create a more precise output provenance in Line 7, then to refine the input provenance again in Line 13. The initial output provenance is computed with the auxiliary

Data: $\text{args}_{\text{in}} = \langle i_{\text{in}}, \tau_{\text{in}}, \widehat{\tau}_{\text{in}}, p_{\text{in}} \rangle$ the input description
Data: $\text{args}_{\text{out}} = \langle o_{\text{out}}, \tau_{\text{out}}, \widehat{\tau}_{\text{out}} \rangle$ the output description
Data: e the desired constructor expression
Result: Code computed the memory value corresponding to expression e

1 **function** REBUILD($args_{in}, args_{out}, e$):
2 **if** $e = c \wedge \widehat{\tau}_{out} = W_\ell$ **then**
  // Target value is a constant encoded in an immediate type.
3 **return** $\boxed{o_{out} := c \ ; \ \texttt{success}}$
4 **else** // Otherwise, Explore all cases.
5 $\mathcal{P}_{\text{in}\to\text{out}} \leftarrow \{(\pi, \pi') \mid \textbf{focus}(\pi', e) = \pi\}$
6 $p_v \leftarrow \textbf{prov\_of}(\tau_{\text{out}}, e)$
7 $p_{\text{out}} \leftarrow \text{REMAP}\ (p_{\text{in}}, p_v, \mathcal{P}_{\text{in}\to\text{out}})$
8 $B \leftarrow$ **for** $p_b, \tau_b, \widehat{\tau}_b, F_b \in \text{EXPLORE}(p_{out}, \tau_{out}, \widehat{\tau}_{out})$ **do**
  // Fill in constant parts of the target memory type.
9 $\text{consts}_b \leftarrow \left\{(\widehat{\pi}, \text{c}) \;\middle|\; \widetilde{\textbf{focus}}(\widehat{\pi}, \widehat{\tau}_b) = (= \text{c})\right\}$
10 $\mathcal{K}_{\text{consts}} \leftarrow$ **for** $(\widehat{\pi}, c) \in consts_b$ **do**
11 $o \leftarrow$ fresh symbol
12 **yield** $\boxed{\texttt{let out } o = o_{out}.\widehat{\pi} \ ; \ o := c \ ; \ \texttt{success}}$
  // Rebuild target fragments from the input value, which we specialize for the current branch.
13 $p_{\text{in},b} \leftarrow \text{REMAP}(p_b, p_{\text{in}}, \text{INV}(\mathcal{P}_{\text{in}\to\text{out}}))$
14 $\text{args}'_{\text{in}} \leftarrow \langle s_{\text{in}}, \tau_{\text{in}}, \widehat{\tau}_{\text{in}}, p_{\text{in},i} \rangle$
15 $\mathcal{K}_{\text{frags}} \leftarrow$ **for** $(\widehat{\pi}_f \mapsto (\pi_f \text{ as } \widehat{\tau}_f)) \in F_b$ **do**
16 $o \leftarrow$ fresh symbol
17 $\text{args}'_{\text{out}} \leftarrow \langle o, \textbf{focus}(\pi_f, \tau_b), \widehat{\tau}_f \rangle$
18 **if** $\exists (\pi_{in}, \pi_{out}) \in \mathcal{P}_{in\to out}, \exists \pi, \pi_{out}.\pi = \pi_f$ **then**
  // If this fragment maps to a location within the input value, use it as a piece of the output value.
19 $\pi' \leftarrow \pi_{\text{in}}.\pi$
20 **yield** $\boxed{\begin{array}{l}\texttt{let out } o = o_{out}.\widehat{\pi}_f \ ; \\ \text{SEEK}(args'_{in}, args'_{out}, \pi')\end{array}}$
21 **else** // Otherwise, break it down further.
22 $e' \leftarrow \textbf{focus}(\pi_f, e)$
23 **yield** $\boxed{\begin{array}{l}\texttt{let out } o = o_{out}.\widehat{\pi}_f \ ; \\ \text{REBUILD}(args'_{in}, args'_{out}, e')\end{array}}$
24 **yield** $(p_{in,b}, \boxed{\mathcal{K}_{consts}; \mathcal{K}_{frags}})$
  // Assemble these branches into a decision tree.
25 **return** $\boxed{\text{COMPILEMATCH}\ (i_{in}, \widehat{\tau}_{in}, B)}$

Algorithm 4: REBUILD *without allocations*

1 **function** REMAP($p_l, p_r, \mathcal{P}$):
2 p such that p is more precise than p_l and
3 $\forall (\pi_l, \pi_r) \in \mathcal{P}, \forall \pi$ s.t. $\textbf{focus}(\pi_l.\pi, p_l) = \_$, then
4 $\textbf{focus}(\pi_l.\pi, p) = \textbf{focus}(\pi_r.\pi, p_r)$.

Algorithm 5: REMAP auxiliary operation

prov_of function, which is a simple syntactic translation from expressions to provenances.

$$\widehat{p} \in \widehat{Shapes} ::= ?_\ell \mid W_\ell \mid \&_\ell\,(\widehat{p}) \mid \{\!\{\widehat{p}_1, \ldots, \widehat{p}_n\}\!\}$$

Figure 11. Memory shapes

$\widehat{\textbf{shape\_of}}\{$

$$\begin{array}{ll}
\widehat{t} & \longrightarrow \widehat{\textbf{shape_of}}(\widehat{\Gamma}(\widehat{t})) \\
(\pi \text{ as } \widehat{\tau}) & \longrightarrow \widehat{\textbf{shape_of}}(\widehat{\tau}) \\
W_\ell \ltimes \ldots & \longrightarrow W_\ell \\
\&_{\ell,a}\,(\widehat{\tau}) \ltimes \ldots & \longrightarrow \&_\ell\left(\widehat{\textbf{shape_of}}(\widehat{\tau})\right) \\
\{\!\{\widehat{\tau}_1, \ldots, \widehat{\tau}_n\}\!\} & \longrightarrow \left\{\!\!\left\{\widehat{\textbf{shape_of}}(\widehat{\tau}_1), \ldots, \widehat{\textbf{shape_of}}(\widehat{\tau}_n)\right\}\!\!\right\} \\
\widehat{\tau} & \longrightarrow ?_{|\widehat{\tau}|}
\end{array}$$

$\}$

Figure 12. Translation from layouts to shapes

$\textsc{NewAllocs}_{\widehat{\pi}}\{$

$$\begin{array}{ll}
?_\ell, ?_\ell \mid ?_\ell, W_\ell & \to \varnothing \\
?_\ell, \&_\ell\,(\widehat{p}) & \to \{(\widehat{\pi}, |\widehat{p}|)\} \cup \textsc{NewAllocs}_{\widehat{\pi}.*}(?_{|\widehat{p}|}, \widehat{p}) \\
?_\ell, \{\!\{\widehat{p}_1, \ldots, \widehat{p}_n\}\!\} & \to \bigcup_{1 \le i \le n} \textsc{NewAllocs}_{\widehat{\pi}.i}\left(?_{|\widehat{p}_i|}, \widehat{p}_i\right) \\
W_\ell, W_\ell & \to \varnothing \\
\&_\ell\,(\widehat{p}), \&_\ell\,(\widehat{p}') & \to \textsc{NewAllocs}_{\widehat{\pi}.*}(\widehat{p}, \widehat{p}') \\
\{\!\{\widehat{p}_i \ldots\}\!\}, \{\!\{\widehat{p}'_i \ldots\}\!\} & \to \bigcup_{1 \le i \le n} \textsc{NewAllocs}_{\widehat{\pi}.i}(\widehat{p}_i, \widehat{p}'_i)
\end{array}$$

$\}$

Figure 13. Difference between shapes

5.2 Memory Allocation

The algorithm presented so far emits a target program that populates the provided output location to represent the desired output value, even in the case of "split" information (i.e. the access code in Figs. 4 and 6). However, we have not yet defined which memory locations are suitable for storing values of a given layout, and assumed that all memory path operations (dereference, struct access…) are legal and defined on all output locations. Naturally, this is not the case, and we might need to allocate new locations as we build the output value. This section focuses on the task of allocating such suitable memory to receive output values.

Memory shapes. We formalise the notion of a "suitable" memory location for a given layout with *memory shapes*, denoted $\widehat{p}$ and defined in Fig. 11. A shape describes the concrete layout of some value in memory, which can be either a fixed-size word, a fixed-size pointer to a known shape, or a struct aggregating several shapes together. The *opaque* shape $?_\ell$ is used for memory locations that have a known, fixed size, but whose precise shape is not known yet. We define the shape of a memory layout $\widehat{\tau}$ as $\widehat{\textbf{shape\_of}}(\widehat{\tau})$; we assert that any valid memory layout $\widehat{\tau}$ has a known size (denoted $|\widehat{\tau}|$) and use the largest branch size for split layouts, so that the shape of any memory layout is always defined.

Allocation procedure. The intuition behind our allocation procedure is as follows: for both Seek and Rebuild, we know that the output location o_{out} conforms with the shape $\widehat{\textbf{shape\_of}}(\widehat{\tau}_{\text{out}})$. This shape might be *opaque*, for instance if $\widehat{\tau}_{\text{out}}$ is a split. In Rebuild, we Explore the output type and layout, yielding a more precise layout $\widehat{\tau}_b$. This is precisely the place where we might need to allocate: if we discover that $\widehat{\tau}_b$ contains pointer to new structures, we should allocate them for future use. This leads us to define our allocation procedure GenAllocs (Algorithm 6) based on a notion of *difference of shapes* computed by NewAllocs (Fig. 13).

```
1 function GenAllocs(o, τ̂_old, τ̂_new):
2   p̂_old ← shape_of(τ̂_old)
3   p̂_new ← shape_of(τ̂_new)
4   for (π̂, ℓ) ∈ NewAllocs(p̂_old, p̂_new) do
5     o' ← fresh symbol
6     yield let dest o' = o.π̂ ;
            o' := alloc ℓ ;
            success
```

Algorithm 6: GenAllocs

NewAllocs takes two shapes, and collects pairs $(\widehat{\pi}, \ell)$ where the shape at position $\widehat{\pi}$ was previously unknown and is now a pointer to a shape of size ℓ. GenAllocs takes an output location o, an imprecise layout $\widehat{\tau}_{\text{old}}$ and a more precise layout $\widehat{\tau}_{\text{new}}$, and emits a piece of code $\mathcal{K}_{\text{allocs}}$ which allocates memory for each position reported by NewAllocs. $\mathcal{K}_{\text{allocs}}$ is then inserted in the code returned by Rebuild, on Line 24.

$\widehat{\textbf{shape\_of}}$, defined in Fig. 12, is a best-effort translation from memory layouts to shapes. It must be conservative (to ensure allocations do happen), but can be fairly imprecise when it comes to splits, as is the version we describe. However, a more precise definition is also possible, and even desirable. For instance, if all branches of a split are pointers to same-sized structures, we can report a good shape immediately, leading to an earlier allocation and code deduplication.

5.3 Recursive Constructors

Although the algorithm presented in Section 5.1 is sufficient to handle most situations, it does not necessarily terminates in the presence of recursive types and layouts. Let us consider it on an example.

Example 5.1 (Recursive rebuilding of linked lists)**.** Consider simply-linked lists of 32-bits integers $\tau = \mathrm{N} + \mathrm{C}(i32, \tau)$ with two possible layouts. $\widehat{\tau}_1$ is a traditional "pointers and blocks" layout with a pointer for each element. $\widehat{\tau}_2$ is 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:

$$
\begin{array}{l}
\widehat{\tau_2} = \text{Split}\ (.[0:2])\ \{ \\
\quad 2\ \text{from}\ \text{N} \Rightarrow W_{64} \\
\quad 1\ \text{from}\ \text{C}(\_,\text{N}) \Rightarrow W_{64} \ltimes [2:32] : (.\text{C.0 as}\ W_{32}) \\
\quad 0\ \text{from}\ \text{C}(\_,\text{C}(\_,\_)) \Rightarrow \\
\quad \&_{64,2}\ (\{\!\{(.\text{C.0 as}\ W_{32}), (.\text{C.1.C.0 as}\ W_{32}), (.\text{C.1.C.1 as}\ \widehat{\tau_2})\}\!\}) \\
\}
\end{array}
$$

It turns out that we can already ask Rebuild to emit appropriate conversion code: $\textsc{Rebuild}(\langle i, \tau, \widehat{\tau_1}, \_\rangle, \langle o, \tau, \widehat{\tau_2}\rangle, \epsilon)$. However, our current algorithm will not terminate, as we try to Rebuild each block in the list.

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 supplementary material. Intuitively, a call to Seek with arguments $args_{\text{in}} = \langle i_{\text{in}}, \tau_{\text{in}}, \widehat{\tau}_{\text{in}}, p_{\text{in}}\rangle$, $args_{\text{out}} = \langle o_{\text{out}}, \tau_{\text{out}}, \widehat{\tau}_{\text{out}}\rangle$ and π 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 $\tau_{\text{in}}, \widehat{\tau}_{\text{in}}, p_{\text{in}}, \tau_{\text{out}}, \widehat{\tau}_{\text{out}}$ and π or e. 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 $f(i, o)$. 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 5.2 (Linked lists, cont'd)**.** Using memoisation, the Rebuild call from our previous example terminates and emits recursive code. We explore $\widehat{\tau_2}$ and get three branches from its split. The first branch, corresponding to the provenance N, is immediate (we only have to write the tag constant). The second branch (C(_, N)) yields a layout with a single fragment $(.[2:32] \mapsto (.\text{C.0 as}\ W_{32}))$, which is immediately retrieved from the input. The third branch (C(_, C(_, _))), on the other hand, requires rebuilding the fragment $(.*.2 \mapsto (.\text{C.1.C.1 as}\ \widehat{\tau_2}))$, which is more involved. Indeed, $\widehat{\tau_1}$ represents the tail of a linked list with a pointer to the fragment $(.\text{C.1 as}\ \widehat{\tau_1})$. After two recursive Seek calls (we focus into $(.\text{C.1 as}\ \widehat{\tau_1})$ twice), we eventually attempt to Rebuild a piece of type τ represented as $\widehat{\tau_2}$ from the same piece represented as $\widehat{\tau_1}$, 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. 14.

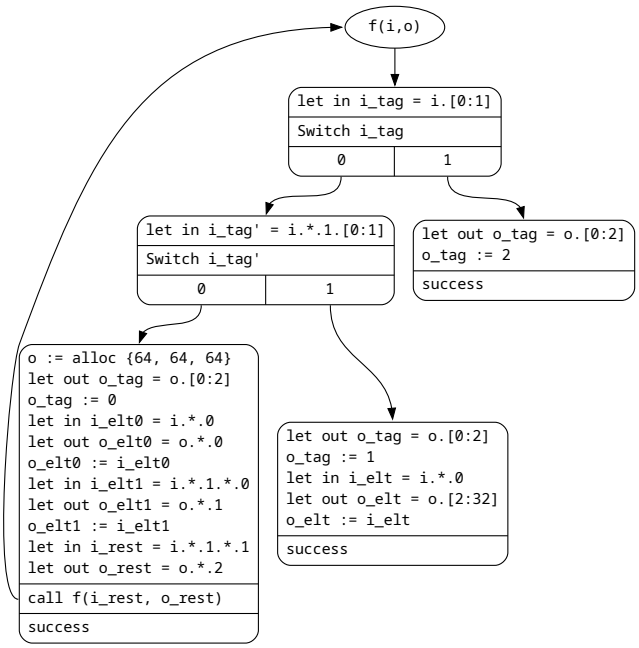

Figure 14. Generated code for rebuilding linked lists

6 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 4.3. 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 5.3. 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. "FP²: 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).

Supplementary material for the CC'24 submission:
Rebuilding Algebraic Data Types from Mangled Memory Layouts

A Focusing and Specialisation

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

Focusing in Types and Layouts. "Focusing" 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 $\tau_{\text{RISC-V}}$:

$$\mathbf{focus}(.\text{Sw}.0, \tau_{\text{RISC-V}}) = \mathbf{focus}\big(.\text{Sw}.0, \text{Sw}(\tau_{\text{reg}}, \tau_{\text{reg}}, i12)\big) = \tau_{\text{reg}}$$

Layout focusing, 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, $\widehat{\mathbf{focus}}(.[7:5], \widehat{\tau}_{\text{Sw}}) = (.\text{Sw}.0 \text{ as } \widehat{\tau}_{\text{reg}})$.

Layout and Type Specialisation. "Specialisation" filters a type or a layout to exclude parts which are incompatible with a given provenance. Type specialisation, denoted τ/p, is a simple syntactic filter that discards irrelevant constructors. For instance, $\tau_{\text{RISC-V}}/\text{Sw}(\_,\_,\_) = \text{Sw}(\tau_{\text{reg}}, \tau_{\text{reg}}, i12)$. Layout specialisation, denoted $\widehat{\tau}/p$, 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, $\widehat{\tau}_{\text{RISC-V}}/\text{Sw}(\_,\_,\_)$ 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:

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

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

Figure 15. Type Focusing

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

Figure 16. Layout Focusing

$$\tau/\_ = \tau \qquad \langle \tau_1, \ldots, \tau_n \rangle / \langle p_1, \ldots, p_n \rangle = \langle \tau_1/p_1, \ldots, \tau_n/p_n \rangle$$
$$K_1(\tau_1) + \ldots + K_n(\tau_n)/K(p) = K_i(\tau_i/p)$$

Figure 17. Type Specialisation

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

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

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

Figure 18. Layout Specialisation

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 7, 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(Rebuild):
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       K_body ← Rebuild(args'_in, args'_out, π)
14       H(h) := Defined(f, λ(s, d).K_body)
15       return call f(s_in, d_out) ; success
16     end
17   }
```

Algorithm 7: Wrapper for emitting recursive code

C Supplementary Examples

Example C.1. We model simple arithmetic expressions with the type $\tau_{\text{op}} = \text{Plus} + \text{Minus} + \text{Times}$ for operators and $\tau_{\text{arith}} =$

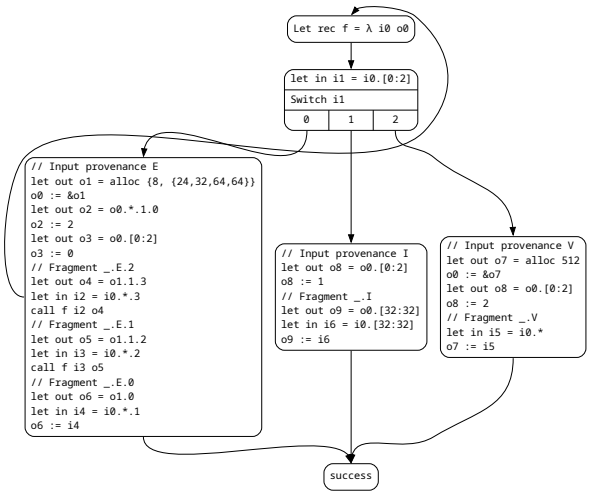

Figure 19. Generated code for converting arithmetic expressions from $\widehat{\tau}_{\text{naive}}$ to $\widehat{\tau}_{\text{optim}}$

$V(\text{str}) + I(i32) + E(\tau_{\text{op}}, \tau_{\text{arith}}, \tau_{\text{arith}})$ for expressions. $\widehat{\tau}_{\text{op}}$ is the immediate C-style enum representation on 4 bits.

We give two layouts for arithmetic expressions. First, a naive layout with pointers and blocks:

$$
\begin{aligned}
&\widehat{\tau}_{\text{naive}} = \text{Split}\,(.[0:2])\,\{\\
&\quad 0 \text{ from } E \;\Rightarrow\; \&_{64,2}\left(\left\{\!\!\left\{ \begin{array}{c} W_{56}; (.E.0 \text{ as } \widehat{\tau}_{\text{op}}) \\ (.E.1 \text{ as } \widehat{\tau}_{\text{naive}}) \\ (.E.2 \text{ as } \widehat{\tau}_{\text{naive}}) \end{array} \right\}\!\!\right\}\right) \ltimes [0:2] : (= 0)\\
&\quad 1 \text{ from } I \;\Rightarrow\; W_{64} \ltimes [0:2] : (= 1) \ltimes [32:32] : (.I \text{ as } W_{32})\\
&\quad 2 \text{ from } V \;\Rightarrow\; \&_{64,2}\,((.V \text{ as str})) \ltimes [0:2] : (= 2)\\
&\}
\end{aligned}
$$

and a somewhat optimised layout, which packs expressions with at least one integer operand into a smaller struct:

$$
\begin{aligned}
&\widehat{\tau}_{\text{optim}} = \text{Split}\,(.[0:2])\,\{\\
&\quad 0 \text{ from } E \;\Rightarrow\; \&_{64,2}\left(\left\{\!\!\left\{(\_.E.0 \text{ as } \widehat{\tau}_{\text{op}}), \widehat{\tau}_{E\text{op}}\right\}\!\!\right\}\right) \ltimes [0:2] : (= 0)\\
&\quad 1 \text{ from } I \;\Rightarrow\; W_{64} \ltimes [0:2] : (= 1) \ltimes [32:32] : (\_.I \text{ as } W_{32})\\
&\quad 2 \text{ from } V \;\Rightarrow\; \&_{64,2}\,((\_.V \text{ as str})) \ltimes [0:2] : (= 2)\\
&\}
\end{aligned}
$$

where

$$
\begin{aligned}
&\widehat{\tau}_{E\text{op}} = \text{Split}\,(.0)\,\{\\
&\quad 0 \text{ from } E(\_, I, \_) \;\Rightarrow\; \left\{\!\!\left\{ \begin{array}{c} W_{24} \ltimes . : (= 0) \\ (\_.E.1.I \text{ as } W_{32}) \\ (\_.E.2 \text{ as } \widehat{\tau}_{\text{optim}}) \end{array} \right\}\!\!\right\}\\
&\quad 1 \text{ from } E(\_, \_, I) \;\Rightarrow\; \left\{\!\!\left\{ \begin{array}{c} W_{24} \ltimes . : (= 1) \\ (\_.E.2.I \text{ as } W_{32}) \\ (\_.E.1 \text{ as } \widehat{\tau}_{\text{optim}}) \end{array} \right\}\!\!\right\}\\
&\quad 2 \text{ from } E(\_, \_, \_) \;\Rightarrow\; \left\{\!\!\left\{ \begin{array}{c} W_{24} \ltimes . : (= 2) \\ (\_.E.1 \text{ as } \widehat{\tau}_{\text{optim}}) \\ (\_.E.2 \text{ as } \widehat{\tau}_{\text{optim}}) \end{array} \right\}\!\!\right\}\\
&\}
\end{aligned}
$$