Compositional Flexible Memory Representations for Algebraic Data Types

Thaïs Baudon, Gabriel Radanne, Laure Gonnord

Compositional Flexible Memory Representations for Algebraic Data Types

Thaïs BAUDON, Laure GONNORD, Gabriel RADANNE

RESEARCH REPORT
N° 9495
Decembre 2022
Project-Teams CASH

ISSN 0249-6399 ISRN INRIA/RR--9495--FR+ENG

Compositional Flexible Memory Representations for Algebraic Data Types

Thaïs BAUDON*, Laure GONNORD†, Gabriel RADANNE‡

Project-Teams CASH

Research Report n° 9495 — Decembre 2022 — 36 pages

Abstract: Initially present only in functional languages such as OCaml and Haskell, Algebraic Data Types have now become pervasive in mainstream languages, providing nice data abstractions and an elegant way to express functions though *pattern matching*. Numerous approaches have been designed to compile rich pattern matching to cleverly designed, efficient decision trees. However, these approaches are specific to a choice of *memory representation* which must accommodate garbage collection and polymorphism.
ADTs now appear in languages more liberal in their memory representation. Notably, Rust is now introducing more and more memory optimisations. As memory representation and compilation are interdependent, it raises the question of pattern matching compilation for highly customised layouts.
We propose to ease the experimentation of new, custom layouts in the early stages of compiler development by providing specification tools and a complete synthesis chain to generate pattern matching compilation procedures.
In this article, we present a novel way to specify *compositional* memory layouts, for which we automatically synthesise elementary builders and accessors, yielding a correct representation-specific compilation algorithm. This approach is implemented in a prototype tool `ribbit`.

Key-words: Algebraic Data Types, Memory Layouts, Pattern Matching, Compilation

* ENS de Lyon, LIP (UMR CNRS/ENS Lyon/UCB Lyon1/INRIA),F-69000 Lyon, France
† Grenoble-INP/LCIS & LIP
‡ Inria, LIP (UMR CNRS/ENS Lyon/UCB Lyon1/INRIA),F-69000 Lyon, France

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Compositional Flexible Memory Representations for Algebraic Data Types

Résumé : Initialement présents dans les langages fonctionnels comme Ocaml et Haskell, les types algébriques sont maintenant de plus en plus présents dans les langages *mainstream* comme Rust. Ils permettent une abstraction commode des données et une façon élégante de décrire des fonctions via le filtrage de motifs (*pattern-matching*).

Ce rapport de recherche présente un framework facilitant la création et l'experimentation de nouvelle représentation mémoire en fournissant un outil de specification pour les représentations, et une chaine de synthèse completer pour générer les procedures de compilation attenante. La correction de l'algorithme est prouvée sous une hypothèse de *validité* de la représentation. L'algorithme est implémenté dans l'outil `ribbit`.

Mots-clés : HPC, types algébriques, filtrage, compilation

Contents

1 Introduction

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

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

Despite these promises, Algebraic Data Types were initially only present in functional programming languages such as OCaml and Haskell. Recently, they have gained a foothold in more mainstream languages such as Typescript, Scala, Rust and even soon Java. They are however still lacking in high-performance lower-level languages. One difficulty for language designers wishing to add pattern matching to their language is that compiling a rich pattern matching language to efficient code is a non-trivial task, which is not commonly available in shared compiler frameworks such as LLVM. Indeed, such frameworks only provide optimisations for C-like switches on integers (or integer-like enumerations). Additionally, existing works on pattern matching Maranget (2008); Wadler (1987); Sestoft (1996) provide very efficient compilation schemes, but are geared towards memory representations found in GC-managed functional languages such as OCaml and Haskell: uniform representations with liberal usage of boxing. Highly non-uniform data representations such as the ones found in C++ do not easily fit.

More generally, the descriptive nature of ADTs should enable compilers to aggressively optimise the representation of terms. The simplest example is the `Option` type, whose values are either `Some value` or `None`. An easy way to represent such type is to box the value in the `Some` constructor below a pointer. However, if that value is an integer ranging from 0 to 10, we can represent it unboxed and use 11 for `None`. This optimisation is regularly done by programmers manually in for instance C++, at the price of error-prone manipulations. More complex optimisations on nested and rich data types are even more error-prone.

While these transformation are not generally easy, compilers have all the information at hand to perform them automatically on the whole program. This trove of potential optimisations has been brushed upon in recent versions of Rust, but remains largely unexplored. One reason for this lack of exploration is that ensuring the correctness of such optimising transformation is difficult. Indeed, the choice of memory representation of values in a language generally has far reaching consequences in the whole compiler, the language runtime but also it's foreign function interface and garbage collector. Compiler designers are thus legitimately careful about changes in the memory representation, where mistakes could lead to complex compilation bugs.

Unfortunately, there is no ultimate memory representation: each language has very specific demands concerning its memory representation. Some languages requires great uniformity, for instance to allow for a tracing GC, other aim for the most compact representation. This prevent the development of a single highly efficient and formally proven memory representation that could be used in many languages.

We propose to address these difficulties from the perspective of a compiler designer that desires to explore various "memory optimisations" for monomorphic types and their performance regarding the pattern-matching procedure:

- We introduce a notion of *compositional* memory representation for Algebraic Data Types and define a novel specification for the compiler developer to provide such a representation.
- From this specification, and for each type of the input program, we automatically synthesise a function from values to their memory representation, along with a state-of-the-art optimising pattern matching compilation procedure tailored to the specified memory representation.
- Our synthesis procedure automatically checks that the provided specification is correct, and enforces the validity of the generated representation and the associated pattern matching compilation algorithm.
- Our framework is implemented in a tool `ribbit`.

```
//Colors for the Red-Black Trees
enum Color { Red, Black }

//Red-Black Trees of content T
enum RBT<T> {
  Node (Color, T, &RBT<T>, &RBT<T>),
  Empty,
}

pub fn cardinal<T>(v : RBT<T>) -> u32 {
  match v {
    Tree::Node(_, _, l, r) => 1+cardinal(*l)+cardinal(*r),
    Tree::Empty => 0
  }}
```

Figure 1: Red-Black trees and cardinal operation in Rust

```
fn cardinal(_1:RBT) = switch(_1){
    | 0 -> 0
    | 1 -> let _5 = cardinal(*((_1 as Node).2))
           let _6 = cardinal(*((_1 as Node).3))
           1 + _6 + _5
}
```

(a) Simplified Rust "MIR" output

```
cardinal (param/290) =
  (if (== param/290 1)
    0
    (+ (+ 1 (apply* cardinal (field2 param/290))
      (apply* cardinal (field3 param/290))))
  )
```

(b) Simplified OCaml "Lambda" output

Figure 2: Intermediate representation of the `cardinal` function in Rust and OCaml

2 Memory Representation of ADTs

To explore in more detail the memory representation of algebraic data types, we now investigate two languages which implements ADTs: Rust and OCaml. While these languages have some common lineage, they have a very different attitude towards code emission: OCaml is a GC-managed language which factors predictability and regularity. Rust on the other hand favours performance and absolute control over low-level details. These differences result in drastically different choices in memory representation. We now focus on a concrete example of algebraic datatype which requires both expressivity and performance: Red Black Trees.

2.1 Red-Black Trees in OCaml and Rust

A Rust version of *Red-Black Trees* is depicted in Fig. 1. We first define `Color`, which is either the constant `Red` or `Black`. The type definition of `RBT<T>` expresses trees as recursive data structures of content `T`. It has two cases: `Empty`, and `Node` which contains a colour, a value, and two pointers (denoted `&(..)`) to its left and right subtrees. For instance, `Node(Red, 1515, &Node(Black, 0, &Empty,&Empty), &Empty)` is a tree of type `RBT<u64>`.

The `cardinal` function takes a tree and returns its cardinal using *pattern matching*. If its argument is of the "shape" of the left-hand side of the rule then the value of the right-hand side (body) is evaluated. In Rust, pattern matching is introduced by the keyword `match` and alternatives are depicted under the form of a list of the form $p \Rightarrow b$ where p is a *pattern* and b its body. Moreover, patterns can be nested, and the body can use named subterms. In our example, `Empty` yields a cardinal of 0 and `Node(_,_,l,r)` yields a cardinal of `1+cardinal(l)+cardinal(r)`.

We now look at how Rust and OCaml compile the `cardinal` function. Both compilers provide an intermediate representation (IR) in which the pattern matching is represented as a decision tree which manipulates the underlying memory representation. We showcase simplified versions of the parts of interest in Fig. 2. OCaml's Lambda IR represents matching using the built-in primitive `field` to access subterms inside the

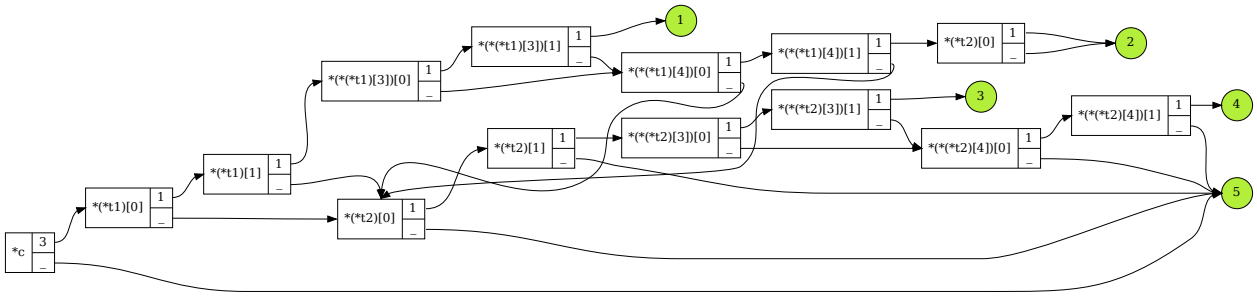

Figure 3: Output of our `ribbit` compiler for the balancing code of Fig. 1 in the OCaml memory representation
This output is a decision tree whose root is on the left. Decision (square) nodes compute values from their input (low-level representation), then branch depending on this value. Round nodes denote the final returned value of the procedure (the numbering of the matched branch).

representation. It uses the `if` primitive to discriminate between cases. Rust's MIR[1] uses slightly lower level operations such as dereferencing and field accesses, and a `switch` construct on memory words.

In both cases, `Red`, `Black` and `Empty` are represented as unboxed integers. The similarity stops there. In OCaml, sum types are represented uniformly: argument-less constructors are unboxed integers, and constructors with arguments are pointers to a block, which always starts with a tag and some fields. In the `Node` case, the block contains a tag and four fields. This totals 6 memory words. In Rust, `Node` is directly a struct containing only four words. Indeed, Rust recognises that the tag (which differentiate `Node` and `Empty`) and the colour can be both packed in a single 64bits field. The resulting type uses 4 memory words.

2.2 Pattern Matching Optimising Compilation

Red-Black trees famously rely on a fairly complex balancing step, which redistributes the colours depending on the internal invariant of the data structure. Thanks to nested patterns, this step can be expressed very compactly using the following matching:

```
match c, v, t1, t2 {
    Black, z, &Node(Red, y, &Node(Red, x, a, b), c), d
  | Black, z, &Node(Red, x, a, &Node(Red, y, b, c)), d
  | Black, x, a, &Node(Red, z, &Node(Red, y, b, c), d)
  | Black, x, a, &Node(Red, y, b, &Node(Red, z, c, d))
  => Node(Red, y, &Node(Black,x,a,b), &Node(Black,z,c,d)),
     a, b, c, d => Node (a, b, c, d),
}
```

This pattern matching inspects four arguments at the same time: the current colour `c`, the current value `v` and the sub-trees `t1` and `t2`. Since this pattern matching is at the core of a performance-sensitive data structure, we naturally want it to be as efficient as possible. This is why many pattern matching implementations come with clever heuristics and techniques to output optimised decision trees Kosarev et al. (2020); Maranget (2008); Sestoft (1996). The resulting code is highly non-trivial, as can be seen in Fig. 3.

2.3 Defining a Memory Representation

So far, we have kept a prudish veil on how one actually defines the memory representation of values in a language like Rust or OCaml. This is largely because it is composed of an uncountable collection of decision spread around the whole design of the compiler and its runtime: it affects the compilation of expressions, creation of values, field accesses, foreign function interfaces, runtime and garbage collection code and, what concerns us most in this article, pattern matching compilation. All these elements must naturally be kept synchronised, at the risk of causing delicate compiler bugs. Even in a language like Rust, which favour optimised memory layouts, incremental improvements to the representation of values has progressed slowly due to the far-reaching nature of such changes to the compiler.

[1] the Middle Intermediate Representation (MIR) is a control flow graph. We reconstruct some of its logic here for ease of reading.

In this article, we present a framework to easily define the memory representation of a language. Our framework automatically checks, using constraint solving, that the proposed representation is well-formed – for instance, that constructors can effectively be differentiated – and synthesise two functions: one to compile pattern matches and one to turn values into their memory representation.

Our framework is based on two main insights.

1. A reasonable memory layout of values should be **composable**: for a value v, its subvalues can be found in its representation. For instance `Node(Red, 1515, &Node(Black, 0, &Empty,&Empty), &Empty)` contains the subvalue `Node(Black, 0, &Empty,&Empty)` in its representation.
2. To yield itself to pattern matching, the memory representation of values should be **distinguishable**: one should be able to determine if an `RBT` value is a `Node` or an `Empty`.

To specify such a memory representation, called *compositional*, one should provide two operations for each type: 1. where to place subterms in the memory representation, 2. how to distinguish values. For simplicity, we only show the definition for the `RBT<u64>` type. A full compiler would provide such information for all types.

2.3.1 Composability

For each type, we require the position of its subterm. Concretely, this is a mapping from source values with holes to memory values with holes:

$$\begin{aligned}
\mathit{Node}(c : \mathit{Color}, _, _, _) &\to \{\!\{ \llcorner c' \lrcorner_{64}, _, _, _ \}\!\} \text{ with } c'[0,1] = 2 * c + 1 \\
\mathit{Node}(_, i : u64, _, _) &\to \{\!\{ _, \llcorner i \lrcorner_{64}, _, _ \}\!\} \\
\mathit{Node}(_, _, l, _) &\to \{\!\{ _, _, \llcorner l \lrcorner_{*}, _ \}\!\} \\
\mathit{Node}(_, _, _, r) &\to \{\!\{ _, _, _, \llcorner r \lrcorner_{*} \}\!\} \\
\&t &\to \mathrm{Ptr}_{64,3} \llcorner t \lrcorner_{*}
\end{aligned}$$

There are many things to note here. $\{\!\{\ \}\!\}$ indicates a structure: a sequence of memory word of various sizes. $\llcorner \_ \lrcorner_*$ is an optionally sized position in the representation. Some fields are directly sized: the integer content i for instance is explicitly sized as 64 bit. The sizes of l and r, the two subtree pointers, are left unspecified and will be inferred. Numerical representation can be tweaked as we move to the memory representation: for instance, the colour c is stored in the first two bits of the header word, appropriately shifted. We also indicate that the pointer is 64-bits wide, and that 3-bits are always 0 for alignment.

To enforce composability, we enforce that the "source" patterns on the left hand side should be *shallow* – without any subpatterns. Note that there are no mapping for *Empty*, since it doesn't have subvalues.

2.3.2 Distinguishability

To distinguish values, we require a collection of *discriminants* that express where in memory to look, and which value would be associated to each constructor. We only need one discriminant for `RBT`:

$$\{\!\{ \llcorner h \lrcorner_{64}, \ldots \}\!\} \text{ with } tag = h[0,0] \Rightarrow \left\{ \begin{array}{l} \mathit{Empty} \mapsto tag = 0 \\ \mathit{Node} \mapsto tag = 1 \end{array} \right\}$$

This reads as follow: "Let tag be the lowest bit of the first field of the given `RBT` value, if the constructor of the value is *Empty* it has $tag = 0$, if it is *Node* is has $tag = 1$".

2.3.3 Synthesis

These two information are in fact completely sufficient to fully determine the memory representation of values. Our synthesis proceeds as follow:

1. We first derives a "general shape" of values. Here, this indicate that RBT memory values are of the shape $\{\!\{\mathrm{Word}_{64}\}\!\}$ in the *Empty* case – a struct with a single 64bits integer – and $\{\!\{\mathrm{Word}_{64}, \mathrm{Word}_{64}, \mathrm{Ptr}_{64,3} RBT, \mathrm{Ptr}_{64,3} RBT\}\!\}$ in the *Node* case – a struct with four 64bits words.

2. Based on this general shape, the missing information is inferred using constraint solving. For instance, it will determine that the word in the *Empty* case will always be 0, to distinguish with the *Node* case. This step also checks the correctness of the representation.
3. Thanks to this last step, the memory representation is fully defined. The framework synthesis a $\textsc{Repr}^\bullet$ functions which returns the memory value of any given value. For instance $\textsc{Repr}^\bullet_{RBT<u64>}(\mathit{Node}(\mathit{Red}, 1515, \mathit{Empty}, \mathit{Empty}))$ $\{\!\{2, 1515, \mathrm{Ptr}_{64,3}\,\{\!\{0\}\!\}, \mathrm{Ptr}_{64,3}\,\{\!\{0\}\!\}\}\!\}$
4. Finally, our framework synthesise a compilation function from pattern to decision trees.

Our synthesis framework has been implemented in a tool called `ribbit`. It provides the full chain from specification of the memory representation (for now, as OCaml code), to constraint solving (using the SMT solver Z3) to pattern matching compilation. We showcase the compilation of the balancing function provided above in Fig. 3.

In the rest of this article, we formally define a language with pattern matching (Section 3), propose an input specification for *compositional representations* (Section 4) and details our synthesis procedure (Section 5). We finally sketch some extensions to our framework (Section 6).

3 Patterns and Pattern Matching

In this section, we present a language of patterns as well as a *generic* algorithm to compile pattern matching to decision trees.

3.1 Patterns, Syntax and Semantics

The Pat language, described in Fig. 4, roughly follows the syntax of ML-style languages, restricted to simple types and patterns. A matching, denoted m, is composed of a list of patterns. The bodies of the clauses are left unspecified here and can be composed of any expression language. Patterns, denoted p, are composed of constructor patterns along with variables (x), wildcards ($\_$) and "or"-patterns ($p_1 \mid p_2$). Type expressions, denoted τ, are composed of sums, written $\sum_{0\leq i<n} K_i(t_{i,0}, \ldots, t_{i,n_i-1})$ or $K_0(\tau_{0,0}, \ldots, \tau_{0,n_0-1}) + \ldots + K_{n-1}(\tau_{n-1,0}, \ldots, \tau_{n-1,n_{n-1}-1})$, and of primitive types (essentially fixed-width signed or unsigned integers). Type environments, denoted Γ, contains both variable and type names, both associated to type expressions. Values, denoted v, follow a subset of the pattern grammar. Finally, value environments, denoted σ, associate variables to their values.

For ease of presentation, we assume the following:

- Types are monomorphic: all code has been specialised.
- All types are named with unique identifiers, whose definitions are stored in the type environment Γ. This allows us to segment types and break recursions. For instance the definition $\tau = A + B(i32)$ yields the type naming environment $\Gamma = \{t \mapsto A + B(t'); t' \mapsto i32\}$. Recursive types result in a cycle in the type environment.
- Patterns are *exhaustive*: all possible cases are handled.

Dynamic Semantics The semantics of patterns take the form of a judgement $p \rhd v \to \sigma$ meaning that pattern p matches value v and binds the variables bound in σ. We also define the matching judgement $\mathrm{Match}\,\{p_1 \mid \cdots \mid p_n\} \rhd v \to i, \sigma$ which additionally returns the index of the matched branch. In a full-fledged language, it would then trigger the evaluation of the body of the branch in question. The semantic rules are standard and shown in Appendix A.

Static Semantics The typing judgement, denoted $\Gamma \vdash p : t \to \Gamma'$, indicates that pattern p is of type t (defined in Γ) and returns a new environment Γ' containing the new variables bound in p. It follows the standard rules for typing patterns, shown in Appendix A.

Example 1 (Running example: definition)**.** As a running example for the rest of the paper, we consider the type t defined in the following type naming environment:

Patterns

$$
\begin{aligned}
p ::= {} & \_ && \text{(Wildcard)}\\
\mid {} & x && \text{(Variable)}\\
\mid {} & (p_1 \mid p_2) && \text{(Disjunction)}\\
\mid {} & K(p_1, \ldots, p_n) && \text{(Constructor pattern)}
\end{aligned}
$$

Matching

$$
m ::= \text{Match}\,\{p_1 \mid \cdots \mid p_n\} \qquad \text{(Matching)}
$$

Types

$$
\begin{aligned}
t ::= {} & T && \text{(Primitive type)}\\
\mid {} & \textstyle\sum_n K_i(t_0, \ldots, t_{n_i - 1}) && \text{(Sum type)}\\
\Gamma ::= {} & \{\overline{x_i \mapsto \tau_i}; \overline{t_j \mapsto \tau_j}\} && \text{(Type env.)}
\end{aligned}
$$

Values

$$
\begin{aligned}
v ::= {} & z \in \mathbb{Z} && \text{(Integer constant)}\\
\mid {} & K(v_0, \ldots, v_{n-1}) && \text{(Constructor value)}\\
\sigma ::= {} & \{x_0 \mapsto v_0; \ldots; x_{n-1} \mapsto v_{n-1}\} && \text{(Value env.)}
\end{aligned}
$$

Figure 4: Pat, our simplified language of patterns and types

$$
\Gamma = \begin{Bmatrix} t \mapsto \mathit{None} + \mathit{Some}(t_{\text{ABC}}); & \\ t_{\text{ABC}} \mapsto A + B + C(t_{\text{int}}); & t_{\text{int}} \mapsto \texttt{u32} \end{Bmatrix}
$$

We define the following matching problem:

$$
m_0 = \text{Match} \begin{Bmatrix} \mid (p_0)\ \mathit{None} \mid \mathit{Some}(A) \\ \mid (p_1)\ \mathit{Some}(B) \\ \mid (p_2)\ \mathit{Some}(C(n)) \end{Bmatrix}
$$

- The *pattern* p_0 matches either value *None* or $\mathit{Some}(A)$. We write $m_0 \rhd \mathit{None} \to 0, \emptyset$ and $m_0 \rhd \mathit{Some}(A) \to 0, \emptyset$.
- The third pattern p_2 matches any value of type t of the form $\mathit{Some}(C(n))$, where n is a value of type t_{int}, and binds its value to the given name. For instance, $m_0 \rhd \mathit{Some}(C(42)) \to 2, \{n \mapsto 42\}$

Additionally, these patterns are well-typed:

$$
\Gamma \vdash p_0 : t \to \Gamma \qquad\qquad \Gamma \vdash p_2 : t \to \Gamma \cup \{n : \texttt{u32}\}
$$

♦

3.2 Templated Compilation of Pattern Matching

We now quickly rephrase Maranget (2008)'s state-of-the-art pattern matching compilation algorithm to decision trees, illustrated in Example 3. Decision trees $\mathcal{T} \in \mathit{Trees}$, whose grammar is described in Fig. 5, formalise the description of *nested switch cases* that we illustrated in Fig. 3.

The function $\textsc{Compile}_t : \textsc{Pat} \to \mathit{Trees}$ takes as input a pattern matching problem for values of a given type t and outputs an equivalent decision tree for *memory representations* of such values. In the rest of this article, we assume that Γ is always provided implicitly.

Decision trees Decision trees are composed of nodes of the form $\mathsf{switch}\ (e)\ \{\ \mathcal{C}\ \}$, consisting of an expression e computing the memory value under scrutiny and of a list of cases $\mathcal{C}$. A decision tree leaf is of the form

Expressions

$$
\begin{array}{rll}
e ::= & \Delta & \text{(Main input)} \\
| & e \oplus z \quad \text{where } \oplus \in \{\wedge, \vee, +, \leq, \neq, \dots\} & \text{(Bitwise and arithmetic operations)} \\
| & * e & \text{(Pointer dereferencing)} \\
| & e.i & (i\text{-th block field access}) \\
| & \dots & \text{(Memory-dependent operations)}
\end{array}
$$

Switch cases

$$
\begin{array}{rll}
\mathcal{C} ::= & z \mapsto \mathcal{T} \ \cdot \ \mathcal{C} & \text{(Regular case)} \\
| & \top \mapsto \mathcal{T} & \text{(Default case)} \\
| & \emptyset & \text{(No default case)}
\end{array}
$$

Decision trees

$$
\begin{array}{rll}
\mathcal{T} \in & \textit{Trees} & \\
\mathcal{T} ::= & \mathsf{switch}\,(e)\,\{\ \mathcal{C}\ \} & \text{(Decision node)} \\
| & \mathsf{success}(j, \{\overline{x_i \mapsto e_i}\}) & (\text{Branch } j \text{ with bindings}) \\
| & \mathsf{unreachable} & \text{(Unreachable leaf)}
\end{array}
$$

Figure 5: Decision trees and their expressions

$\mathsf{success}(j, \sigma)$, which successfully returns a branch index j and a set of binding expressions σ. Since we only consider exhaustive patterns, there are no failure cases.

At toplevel, decision trees take as input the *main input* of the whole matching, denoted Δ. The scrutinee of a switch is an expression manipulating memory values (thus an expression over Δ). These expressions are partially target-dependent but composed at least of dereferencing, field access and simple bitwise and arithmetic operations. The output bindings σ are also target-dependent.

The dynamic semantics of these decision trees are similar to those of pattern matching, thus we omit their description and provide a simple example.

Example 2 (Decision tree)**.** The following decision tree matches an option type in the OCaml representation:

$$
\mathsf{switch}\,(\Delta\ \&\ 1)\begin{Bmatrix} 1 \mapsto & \mathsf{success}(0, \emptyset) \\ 0 \mapsto & \mathsf{success}(1, \{n \mapsto *\Delta.1\}) \end{Bmatrix}
$$

This first determines whether the least significant bit of Δ is 1 or 0. This determines whether a value is *None* (the integer 1) or a *Some* (a pointer). If *None*, we succeed with branch 0. If *Some*, we succeed with branch 1 and a *binding environment* mapping n to the value inside *Some*, as computed by $*\Delta.1$. ♦

Pattern matching compilation algorithm Our algorithm $\textsc{Compile}_{t_0}\,(\mathcal{P})$ compiles a pattern matching problem to an equivalent decision tree using the now classic approach of *pattern matrices*. Its arguments are:

- the type of the main input value (at toplevel) t_0;
- a *pattern matrix* $\mathcal{P}$ in which each row encodes a case of the matching problem and and each column a term to be scrutinised (allowing us, for instance, to split product patterns and choose which field to inspect first). At the end of each row, we also record its index and a binding environment. The column headers indicate which parts of the main input this column is matching against, as a subterm of the toplevel value.

$\textsc{Compile}_t$ proceeds by recursively emitting switch nodes (depicted as branching nodes in our graphical representation) and reducing the pattern matrix until exhaustion. It has five cases, depending on the shape of $\mathcal{P}$:

empty If $\mathcal{P}$ is empty, the algorithm fails, as exhaustive patterns should have a valid pattern for every case.
wildcard If the first row consists only of wildcards, the matching can only succeed as this branch accepts all values.

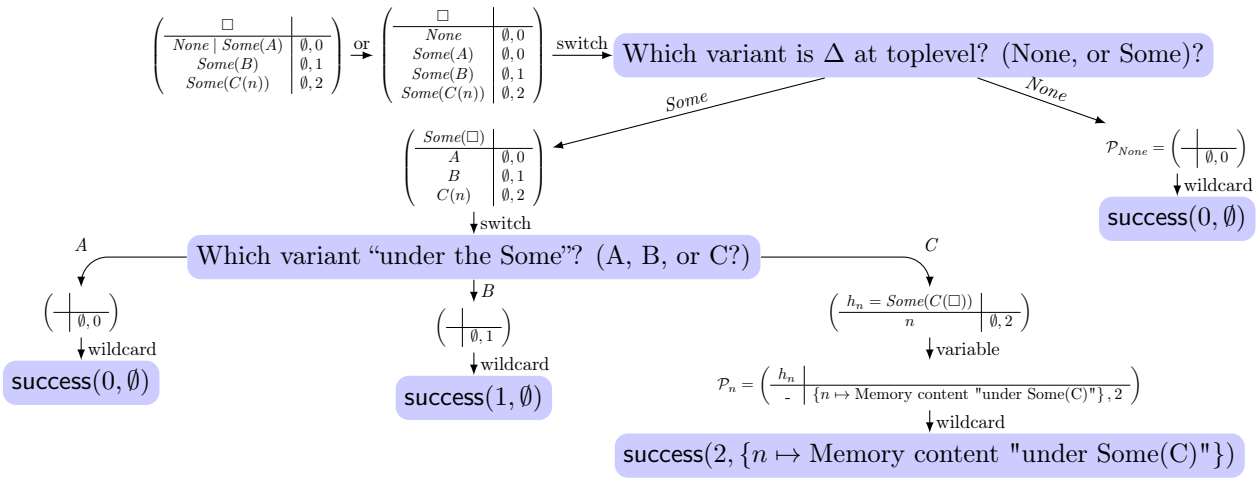

Figure 6: Execution of $\textsc{Compile}_t(m_0)$ on the running example. Blue coloured parts depict the final decision tree (that can be used to match any value of type t); other parts depict pattern matrices used during the compilation procedure.

variable If there exists a variable pattern p_i^ℓ in some row ℓ (and the previous rows contain no variable patterns), we introduce p_i^ℓ into this row's binding environment.
or If there exists an "or"-pattern $p_i^\ell = p_1 \mid p_2$ in some row ℓ (and the previous rows contain no "or"-patterns), we split it into two new rows.
switch Otherwise, an actual switch node must be generated. The aim of this switch node is to inspect the head constructor of the value contained in one of the inputs, compare it to the patterns of its column in $\mathcal{P}$, then branch to its associated subtree that performs the remaining computations (on other columns and nested values).

Example 3 (Running example: $\textsc{Compile}_{t_\Delta}$ function). Fig. 6 depicts the execution of $\textsc{Compile}_t(m_0)$ on the running example (Example 1). The produced decision tree can then be used to "match" the memory representation of any value v of type t. It first applies the **or** rule, then the **switch** rule to decide whether the head constructor of v is *None* or *Some*. In the former case, it immediately stops, returning 0. In the latter case, it continues through the left branch of the tree while getting access to the subvalue "under *Some*", for which it should decide whether it is A, B or a C. Finally, if this subvalue is a C, it should get access to the subvalue "under C". Note that the detailed operation of how to discriminate between constructors such as A, B and C is still undefined at this point. This is indeed highly dependent on precise details of the chosen internal memory layout. ♦

Representation dependence While Maranget (2008)'s algorithm produces good decision trees, it was originally specific to a given memory representation (the OCaml one). In order to allow for custom representations, we must provide additional representation-specific functions to:

- extract the representation of a subvalue from the memory representation of its parent value ($\textsc{Unwrap}_t^\bullet$);
- retrieve the head constructor of a source value from its memory representation ($\textsc{InspectAndSplit}_t^\bullet$).

Unfortunately, specifying these functions for each type and constructor is particularly complex. In particular, ensuring the correctness of such functions when written by hand in compiler code is delicate. Instead, we propose to *synthesise* such representation-specific functions for any custom memory representation, based on a higher-level description of the memory layout of values. In the following sections, we show how to automatically derive the *full compilation algorithm for any type*, including representation-specific functions, from a user-provided *high-level description* of the desired memory representation.

$$\widehat{\tau} \in \widehat{\mathit{Types}} ::= \_ \mid \mathrm{Word}_\ell \mid \mathrm{Ptr}_{\ell,a}\widehat{\tau} \mid \left\{\!\!\left\{\overline{\widehat{\tau}}\right\}\!\!\right\} \mid \bigsqcup \widehat{\tau}$$

$$\widehat{v} \in \widehat{\mathit{Values}} ::= \_ \mid \mathrm{Word}_\ell\,(z) \mid \mathrm{Ptr}_{\ell,a}\widehat{v} \mid \left\{\!\!\left\{\overline{\widehat{v}}\right\}\!\!\right\} \mid \bigsqcup \widehat{v}$$

$$\begin{aligned}
\mathrm{ops} \in \mathit{WordOps} ::= {} & \blacksquare && \text{(Identity)}\\
\mid {} & \mathrm{ops}[i,j] && \text{(Bits } i \text{ to } j\text{, inclusive)}\\
\mid {} & \mathrm{ops} \oplus z \quad \text{where } \oplus \in \{\wedge, \vee, +, \leq, \neq, \dots\} && \text{(Bitwise and arithmetic operations)}\\
\widehat{h} \in \widehat{\mathit{Contexts}} ::= {} & \llcorner \underline{\mathrm{ops}} \lrcorner_* && \text{(Unsized memory hole)}\\
\mid {} & \llcorner \underline{\mathrm{ops}} \lrcorner_\ell && \text{(Sized memory hole)}\\
\mid {} & \mathrm{Ptr}_{\ell,a}\widehat{h} && \text{(Pointer)}\\
\mid {} & \left\{\!\!\left\{_, \dots, \underset{i}{\widehat{h}}, \dots, _\right\}\!\!\right\} && (i\text{-th field in a block})
\end{aligned}$$

Figure 7: Memory values, types and contexts

4 Input Specification for Memory Layouts

A layout, or *memory representation*, specifies how to encode values of any source type in memory, so as to be able to encode each typed value in a fully deterministic way. This section formally explains how to provide an input specification for our synthesis algorithm.

4.1 Notations: Source and Memory Operations

In this section, we define similar notions on the source language Pat and on memory contents. From now on, hats will denote memory-related notions: $h \in \mathit{Contexts}$ denotes a *source* context while $\widehat{h} \in \widehat{\mathit{Contexts}}$ denotes a *memory* context.

Source contexts, subterms of a type Source contexts are used to deconstruct values and types. A constructor context is a constructor pattern in which all subpatterns are wildcards, except one which is another context. All contexts end with a single *hole* $\square$. The grammar for source contexts ($\mathit{Contexts}$) is defined below:

$$\begin{aligned}
h ::= {} & \square && \text{(Hole)}\\
\mid {} & K(_, \dots, _, h, _, \dots, _) && \text{(Constructor context)}
\end{aligned}$$

We can now define the *subterms* of a sum type expression as a set of contexts where subterms are located:

$$\mathit{SubTerms}(\sum_n K_i(\overline{t_{i,j}}^{n_i})) = \left\{ K_i\left(\_, \dots, \underset{j}{\square}, \dots, \_\right) \right\}_{\substack{0 \leq i < n \\ 0 \leq j < n_i}}$$

We also write $\mathit{SubTerms}(t)$ instead of $\mathit{SubTerms}(\Gamma(t))$ when Γ is implicit from the context.

Example 4 (Running example: subterms)**.** From the definition of types in Example 1, we obtain:

$$\mathit{SubTerms}(t) = \{\mathit{Some}(\square)\} \qquad \mathit{SubTerms}(t_{\mathrm{ABC}}) = \{C(\square)\}$$

Constant constructors such as *None* do not yield any subterm context. ♦

Memory Values and Types In order to work on the *internal representation* of values, we consider an abstraction of memory, similar to LLVM IR's types, shown in Fig. 7. A *memory value* is always typed and is either a fixed-size word encoding an integer $z \in \mathbb{Z}$, written $\mathrm{Word}_\ell\,(z)$ and typed Word_ℓ, a fixed-size pointer to another memory value $\widehat{v}$ of type $\widehat{\tau}$ with a fixed number a of address alignment bits (filled with 0s), written $\mathrm{Ptr}_{\ell,a}\widehat{v}$ and typed $\mathrm{Ptr}_{\ell,a}\widehat{\tau}$, or a contiguous block containing a finite number of heterogenous memory values (fields), written $\{\!\{\widehat{v}_0, \ldots, \widehat{v}_{n-1}\}\!\}$ and typed $\{\!\{\widehat{\tau}_0, \ldots, \widehat{\tau}_{n-1}\}\!\}$. We also define a (finite) *disjoint union* of memory values (resp. types) to denote the possibility of several concrete memory values (resp. types) at a given location, along with a wildcard $\_$ to denote an unspecified memory value (resp. type). The typing rules for the judgment $\vdash \widehat{v} : \widehat{\tau}$ are straightforward and omitted for the sake of brevity.

Memory Contexts A *memory context*, defined in Fig. 7, indicates a position within a memory value and an optional transformation of the word at this position. It consists of a coarse-grained hierarchy of *structural* elements – namely pointers and blocks – ending with an optionally sized "hole" $\__\ell$ and of finer-grained operations on the word designated by the hole. The hole of a memory context applied to a word (resp. pointer) value should match the size of this word (resp. the number of alignment bits). The hole of a memory context applied to a block must be unsized and contain the identity operation (i.e., $\llcorner\blacksquare\lrcorner_*$).

Example 5 (Running example: memory contexts)**.** Consider t_{ABC} from our running example (Example 1). Its memory representation in OCaml (which we will refine later in Section 4.3) is of type $\widehat{\tau} = \mathrm{Word}_{64} \sqcup \mathrm{Ptr}_{64,3}\,\{\!\{\mathrm{Word}_{64}, \mathrm{Word}_{64}\}\!\}$. The OCaml representation maps the concrete source values A, B and $C(42)$ to the following memory values:

$$A \mapsto \mathrm{Word}_{64}\,(1) \qquad B \mapsto \mathrm{Word}_{64}\,(3) \qquad C(42) \mapsto \mathrm{Ptr}_{64,3}\,\{\!\{\mathrm{Word}_{64}\,(2)\,, \mathrm{Word}_{64}\,(85)\}\!\}$$

The two constant constructors A and B are mapped to odd constant values. The C constructor is mapped to a 64-bit-wide, 3-bit-aligned pointer (thus an even number), pointing to a block containing its tag 2 and a value. The integer value 42 is located in the second field of this block. As we will see in more detail in Section 4.3, integers are encoded through a 1-bit left shift and an increment ($42 \times 2 + 1 = 85$). ♦

4.2 Input Ingredients for Compositional Layouts

Let t a source type such that $\Gamma(t) = \sum_n K_i(\overline{t_{i,j}}^{n_i})$. In order to synthetise a full (compositional) memory representation, the user should provide:

- $\textsc{Position}^\bullet_t : \mathit{SubTerms}(t) \to \widehat{\mathit{Contexts}}$
  defines how the representation of each subvalue is composed into the representation of its parent value;
- $\textsc{Discrs}^\bullet_t \subset \widehat{\mathit{Contexts}} \times (\{K_i\}_n \to \mathbb{Z})$
  is a set of *discriminants*. A discriminant consists of a memory context indicating where to look, and a *partial* mapping from constructors to integer values that can be found at this position. The discriminants should be defined such that, for each distinct pair of constructors, at least one discriminant is defined for both constructors and maps them to distinct integer values, i.e.:

$$\forall 0 \leq i < j < n,\ \exists(\widehat{h}, \mathit{split}) \in \textsc{Discrs}^\bullet_t,\ \mathit{split}(K_i) \neq \mathit{split}(K_j)$$

4.3 Example: OCaml Layout Specification

We now demonstrate how to specify the OCaml memory representation of values Minsky and Madhavapeddy (2021) in our framework. All computation details can be found in Appendix D.

Informal specification OCaml memory values are either unboxed immediates or pointers to blocks, both one word wide. OCaml blocks are contiguous word-aligned structs. Every block starts with a *header* word containing various metadata including a *tag* indicating the constructor of the underlying value. For simplicity, we ignore the remaining metadata here. An important optimisation is that argument-less constructors are represented as unboxed integers, rather than empty blocks. The integer value is the index of the constructor. In OCaml, *None* has the same representation as the integer value 0 (i.e., $\mathrm{Word}_{64}\,(1)$), without any boxing.

To distinguish pointers from unboxed values at runtime, the least significant bit of every memory value is used as a *pointer flag*. Concretely, a memory value whose lowest bit is set to 1 contains an unboxed constant in its remaining 63 bits, while a lowest bit set to 0 indicates that the memory value is a pointer. Finally, addresses are word-aligned, guaranteeing that their 3 last bits are always 0 (for 64-bit-wide words).

Formal OCaml layout specification Let t such that $\Gamma(t) = \sum_n K_i\left(\overline{t_{i,j}}^{n_i}\right)$.

The $\textsc{Position}^\bullet$ function gives the position of a subvalue in the "pointer-to-block" memory structure:

$$\textsc{Position}^\bullet_t\left(K_i\left(\_,\ldots,\underset{j}{\square},\ldots,\_\right)\right) = \mathrm{Ptr}_{64,3}\left\{\!\!\left\{\_,\ldots,\underset{j+1}{\blacksquare_*},\ldots,\_\right\}\!\!\right\}$$

The three discriminants express how to distinguish between constructors:

- $D_{\geq 0}$ distinguishes between unit and non-unit constructors by inspecting the lowest bit, which is always 1 (resp. 0) for unit (resp. non-unit) constructors:

$$D_{\geq 0} = \left(\underline{\blacksquare[0,0]}_*; \left\{K_i \mapsto \begin{cases} 1 & \text{if } n_i = 0 \\ 0 & \text{otherwise} \end{cases} \mid 0 \leq i < n\right\}\right)$$

- $D_{=0}$ distinguishes individual unit constructors:

$$D_{=0} = (\blacksquare_{64}; \{K_i \mapsto 2 \times i + 1 \mid n_i = 0\})$$

  In this case, the value must be an unboxed integer, which we can inspect right away.
- $D_{>0}$ distinguishes individual non-unit constructors:

$$D_{>0} = (\mathrm{Ptr}_{64,3}\left\{\!\!\left\{\blacksquare_{64},\ldots\right\}\!\!\right\}; \{K_i \mapsto i \mid n_i > 0\})$$

  In this case, the value must be a pointer to a block.

We then have $\textsc{Discrs}^\bullet_t = \{D_{\geq 0}, D_{=0}, D_{>0}\}$.

Example 6 (Running example: t_{ABC} ingredients)**.** Consider t_{ABC} in our running example. As we saw in Example 5, the unit constructors A and B are distinguishable from C by inspecting the lowest bit of the representation (which acts as a pointer flag); we also define a discriminant to distinguish A from B and another that solely affects C (so as to obtain the expected memory structure).

$$\begin{aligned}\textsc{Discrs}^\bullet_{t_{\mathrm{ABC}}} = \Bigg\{ &\left(\underline{\blacksquare[0,0]}_{64}, \begin{cases} A \mapsto 1 & C \mapsto 0 \\ B \mapsto 1 \end{cases}\right), \\ &\left(\blacksquare_{64}, \left\{A \mapsto 1 \quad B \mapsto 3\right\}\right), \\ &\left(\mathrm{Ptr}_{64,3}\left\{\!\!\left\{\blacksquare_{64},\ldots\right\}\!\!\right\}, \{C \mapsto 2\}\right)\Bigg\}\end{aligned}$$

We also define the composition of $\textsc{Repr}^\bullet_{t_{\mathrm{int}}}(n)$ within $\textsc{Repr}^\bullet_{t_{\mathrm{ABC}}}(C(n))$:

$$\textsc{Position}^\bullet_{t_{\mathrm{ABC}}}(C(\square)) = \mathrm{Ptr}_{64,3}\left\{\!\!\left\{\_,\blacksquare_*,\ldots\right\}\!\!\right\}$$

5 Synthesis of Memory Representations

We now give our complete synthesis procedure, whose overall pipeline is shown in Fig. 8, which takes as input the specification functions of Section 4.2.

It proceeds by generating partial intermediate representations of a type t for each *tag*, which can be either a constructor or a wildcard $\top$. It then completes these intermediate representations using constraint solving. This is done in three steps:

1. *Structural interpolation* (Section 5.1) determines the memory type $\textsc{ReprTy}^\bullet_t(tag)$ of the representation of any value of type t whose constructor fits *tag*, as well as a *memory pattern* $\textsc{PreRepr}^\bullet_t(tag)$ which encodes all statically known representation elements. This step proceeds by collecting all necessary structural information from the input specification and computing an "intersection" of all collected constraints. If this step succeeds, its result is a memory pattern with some remaining unsolved word contents.

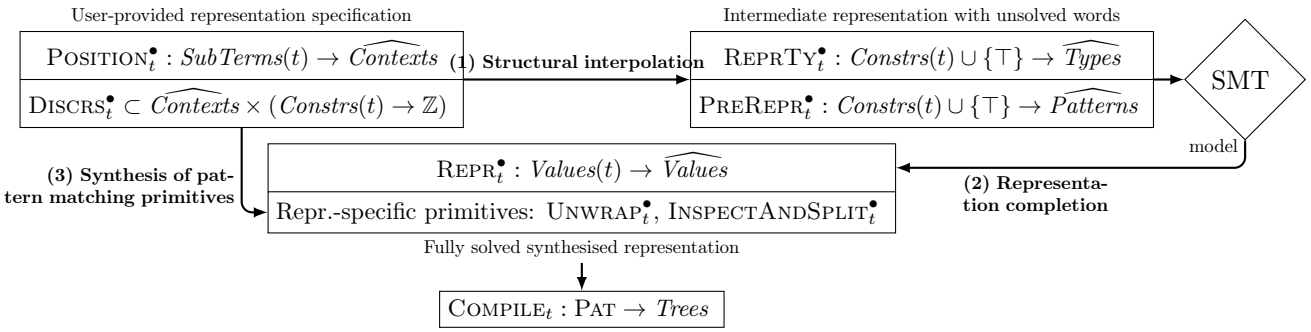

Figure 8: Representation synthesis pipeline

$$\phi ::= \top \mid (\mathrm{ops} = x_h) \mid (\mathrm{ops} = z) \mid \phi \wedge \phi$$
$$\widehat{p} \in \widehat{\mathit{Patterns}} ::= t@h \mid \mathrm{Word}_\ell(\phi) \mid \mathrm{Ptr}_{\ell,a}\widehat{p} \mid \left\{\!\!\left\{\widehat{\widehat{p}?}\right\}\!\!\right\} \mid \bigsqcup \widehat{p}$$

Figure 9: Memory patterns and word formulas

2. *Representation completion* (Section 5.2) fully determines the contents of each word. After ensuring that the intersection of all patterns is non-empty, there still remain under-specified parts. The goal of this step is to synthesise a *unique memory value*. We translate unsolved word descriptions to SMT formulas on fixed-size bitvectors. Solving these formulas yields a model that assigns a concrete value or a subterm symbol to each word, which finishes the complete synthesis of the $\textsc{Repr}_t^\bullet$ function. If a formula is unsatisfiable, it means that we cannot obtain a suitable representation.
3. *Synthesis of pattern matching primitives* (Section 5.3) produces the primitives needed by our templated $\textsc{Compile}_t$ algorithm. More precisely, for each source type t, we generate a function $\textsc{Unwrap}_t^\bullet$ that extracts the representation of a subvalue from the representation of its parent value and $\textsc{InspectAndSplit}_t^\bullet$, which determines the constructor of a source value from its representation.

Each of these three steps may fail, indicating that the input specification is *invalid*. The rest of the section follows the synthesis pipeline. Since the representation of a type is dependent on the representation of its subtypes, we process the types of an environment Γ in the reverse prefix order.

5.1 Structural Interpolation

$\textsc{PreRepr}_t^\bullet(\mathit{tag})$ is representation template in which structural elements are fully determined but word contents remain unsolved. $\textsc{ReprTy}_t^\bullet(\mathit{tag})$ gives the memory type for a given tag. To generate them, we use *memory patterns*, shown in Fig. 9. Memory patterns are memory value skeletons in which word contents are specified by a formula ϕ. A formula is either true ($\top$) or a conjunction of clauses of the form $(\mathrm{ops}_1 = x_1) \wedge \cdots \wedge (\mathrm{ops}_m = x_m)$, where x_k are concrete integers or subterm symbols and ops_k are sequences of operations applied to the word contents $\blacksquare$. Memory patterns also feature an application case $t@h$, which stands for the (unsolved) memory representation of the subterm of type t at position h. The remaining pattern cases are similar to the previously defined memory values and types.

Memory patterns are equipped with *union* and *intersection* operations, defined in Appendix B, which allow to build them up incrementally. For instance, the following rule expresses that a pointer pattern is compatible with any word pattern whose width "fits" in its address alignment bits; the specified word contents must then be zero.

$$\frac{\ell' \leq a \qquad (0)_{\ell'} \text{ satisfies } \phi}{\mathrm{Ptr}_{\ell,a}\widehat{p} \cap \mathrm{Word}_{\ell'}(\phi) = \mathrm{Ptr}_{\ell,a}\widehat{p}}$$

Memory patterns also have an *instantiation* operation, noted $\widehat{h}[\widehat{p}]$ which builds a pattern from $\widehat{h}$, where the hole is replaced by $\widehat{p}$. This is also defined in Appendix B.

Example 7 (Memory patterns)**.** $\text{Word}_{64}\,(\blacksquare[0,0]=1)$ denotes any 64-bit-wide word whose lowest bit is 1. In our running example, the memory pattern $t_{\text{int}}@C(\square)$ matches memory values representing a value of type t_{int}. Ultimately, it will represent $\textsc{Repr}^\bullet_{t_{\text{int}}}(v)$ where v is the subvalue "under C". ♦

5.1.1 $\textsc{PreRepr}^\bullet_t$ and $\textsc{ReprTy}^\bullet_t$

Let t a type and *tag* one of its constructors, or $\top$. $\textsc{PreRepr}^\bullet_t(tag)$ is the intersection of all constraints pertaining to this memory layout: composability constraints, denoted $\widehat{p}^c_t$ and distinguishability constraints, denoted $\widehat{p}^d_t$.

$$\textsc{PreRepr}^\bullet_t(tag) = \widehat{p}^d_t(tag) \cap \bigcap_{h \in SubTerms(t,tag)} \widehat{p}^c_t(h)$$

This memory pattern intersection may be empty, in which case representation generation fails and the input specification is deemed invalid since it provided incompatible memory patterns. Composability constraints are defined by iterating over subterms. Crucially, this yields a natural base case for constructors without subterms. $\textsc{ReprTy}^\bullet_t(tag)$ is a restriction of $\textsc{PreRepr}^\bullet_t(tag)$, where formulas are erased, yielding a memory type. We will now define each of these constraints.

5.1.2 Distinguishability Constraints

Distinguishability constraints $\widehat{p}^d_t(tag)$ encode that, for any given sum type t, the head constructor of any memory value can be identified by inspecting some predefined memory locations. Concretely, we accumulate all constraints on a given constructor, and then merge all constructor-specific constraints.

Let t such that $\Gamma(t) = \sum_n K_i(\overline{t_{i,j}}^{n_i})$ and K_i one of its constructors. We collect constraints by iterating over each discriminant and its constructors, collecting coarse-grained contexts, word operations, widths and values. For unsized memory holes (i.e., $*$), we use the maximal value width.

$$\mathcal{C} =_{\text{def}} \left\{ \begin{array}{l} (\widehat{h}, \ell, \text{ops}, z) \text{ where} \\ \quad (\widehat{h}', \mathit{split}) \in \textsc{Discrs}^\bullet_t \\ \quad (K_i \mapsto z) \in \mathit{split} \\ \quad \widehat{h}[\underline{\text{ops}}_{\downarrow \ell'}] = \widehat{h}' \\ \quad \ell = (\text{if } (\ell' \text{ is } *) \text{ then } \max_{(K \mapsto z) \in \mathit{split}} \texttt{width}(z) \text{ else } \ell') \end{array} \right\}$$

We now define the *specific* distinguishability pattern $\widehat{p}^d_t(K_i)$ that applies to a given constructor K_i by instanciating contexts, intersecting the resulting patterns and taking the conjunction of word formulas:

$$\widehat{p}^d_t(K_i) = \bigcap_{(\widehat{h}_i, \ell_i, \text{ops}_i, z_i) \in \mathcal{C}} \widehat{h}_i[\text{Word}_{\ell_i}\,(\text{ops}_i = z_i)]$$

The *generic* distinguishability pattern of t captures the structural characteristics common to all constructors:

$$\widehat{p}^d_t(\top) = \bigcup_{K \in Constrs(t)} \widehat{p}^d_t(K)$$

5.1.3 Composability Constraints

Composability constraints $\widehat{p}^c_t(h)$ encode that the memory representation of a value that fits h must contain the memory representation of its subvalue at $\textsc{Position}^\bullet_t(h)$. Our goal is to collect representation constraints for each subterm position h, then merge them all.

Let t' be the subtype of t at the hole in h. Inductively, we have defined the *generic representation type* of t': $\widehat{\tau}' = \textsc{ReprTy}^\bullet_{t'}(\top)$. Let $\widehat{h}[\underline{\text{ops}}_{\downarrow \ell_0}] = \textsc{Position}^\bullet_t(h)$. For the representation to be consistent, the memory hole at $\widehat{h}$ must have type $\widehat{\tau}'$. There are two cases:

- The memory value at position $\widehat{h}$ is a proper ℓ-bit wide word. We thus have $\widehat{\tau}' = \text{Word}_\ell$ and $\ell_0 = \ell$ or $\ell_0 = *$. To link the various constraints, we introduce x_h, a variable that contains the integer contents at position h. We output the constraint $\widehat{p}^c_t(h) = \widehat{h}[\text{Word}_\ell\,(\text{ops} = x_h)]$.

- The memory value at position $\widehat{h}$ is a not a word (for instance, it is a block). We necessarily have $\llcorner \underline{\text{ops}} \lrcorner_{\ell_0} = \llcorner \blacksquare \lrcorner_*$. Since the stored memory value is not a word, we simply recall the constraints pertaining to the position h with its type t' using an application pattern: $\widehat{p}^c_t(h) = \widehat{h}[t'@h]$.

Example 8 (Running example: constraints)**.** Consider the type t_{ABC} from our running example and the input ingredients of Example 6:

- t_{ABC} is a sum type with three variants. For the sake of brevity, we only give $\widehat{p}^d_{t_{\text{ABC}}}(C)$, which is an intersection of two memory patterns since two discriminants involve C:

$$\widehat{p}^d_{t_{\text{ABC}}}(C) = \text{Ptr}_{64,3}\left\{\!\!\left\{\text{Word}_{64}\left(\blacksquare = 2\right), \ldots\right\}\!\!\right\} \cap \text{Word}_1\left(\blacksquare[0,0] = 0\right)$$

  This pattern matches memory values that have characteristics of both a 64-bit-wide, 3-bit-aligned pointer to a block with at least one field, whose first field is a 64-bit-wide word whose contents $\blacksquare$ satisfy $\blacksquare = 2$, and of a word or address of at least one bit whose lowest bit is 0.
- t_{ABC} has one subterm $C(\square)$. Its composability pattern is:

$$\begin{aligned}\widehat{p}^c_{t_{\text{ABC}}}(C(\square)) =& \text{Ptr}_{64,3}\left\{\!\!\left\{\_, \llcorner \blacksquare \lrcorner_*, \ldots\right\}\!\!\right\}\left[\text{Word}_{64}\left(\blacksquare = x_{C(\square)}\right)\right] \\ =& \text{Ptr}_{64,3}\left\{\!\!\left\{\_, \text{Word}_{64}\left(\blacksquare = x_{C(\square)}\right)\right\}\!\!\right\}\end{aligned}$$

  which expresses that the representation of $C(n)$ must be a pointer to a block with at least two fields, whose second field is a word containing an (unknown) integer value y such that $y = x_{C(\square)}$, where $x_{C(\square)}$ denotes the integer contents of $\text{Repr}^\bullet_{t_{\text{int}}}(n)$.

♦

5.2 Representation Completion

We have now collected all constraints on our representation into a memory pattern that contains formulas. The next step of representation synthesis consists in *solving* each word formula. More precisely, given $\text{PreRepr}^\bullet_t(K)$ for some constructor K of t, we consider each memory context $\widehat{h}$ such that the memory pattern $\text{Word}_{\ell_{\widehat{h}}}\left(\phi_{\widehat{h}}\right)$ is at position $\widehat{h}$ in the memory pattern $\text{PreRepr}^\bullet_t(K)$.

Let $x_{h_1}, \ldots, x_{h_m}$ be all the subvalue representation variables that appear in $\phi_{\widehat{h}}$. Let $\ell_1, \ldots, \ell_m$ the widths of these subvalues' representations (i.e., $\text{ReprTy}^\bullet_{t_k} = \text{Word}_{\ell_k}$). Our goal is to synthesise a function that, given the subvalues' representations, builds the word of the parent value (which we denoted $\blacksquare$ in our formula). Let $y_{\widehat{h}}$ this function from bitvectors of widths $\ell_1, \ldots, \ell_m$ to an $\ell_{\widehat{h}}$-bit-wide bitvector. Let $\phi'_{\widehat{h}} = \phi_{\widehat{h}}[\blacksquare \to y_{\widehat{h}}(x_{h_1}, \ldots, x_{h_m})]$ the formula with $\blacksquare$ substituted with its (symbolic) value. Solving $\phi'_{\widehat{h}}$ then consists in finding a function interpretation for y that satisfies it, denoted $f_{\widehat{h}}$. If $\phi'_{\widehat{h}}$ is unsatisfiable, the input specification is deemed invalid.

We solve each such formula $\phi'_{\widehat{h}}$ in $\text{PreRepr}^\bullet_t(K)$ and obtain suitable $f_{\widehat{h}}$ interpretations. This gives us fully set values for each memory word, allowing us to define the representation function for source values of constructor K. In order to define the full representation function, we perform the same procedure for each constructor K_i of t. We finally set:

$$\begin{aligned}\text{Repr}^\bullet_t(K_i(\overline{v_j})) = \text{PreRepr}^\bullet_t(K_i)\left[\text{Word}_{\ell_{\widehat{h}}}\left(\phi_{\widehat{h}}\right) \to \text{Word}_{\ell_{\widehat{h}}}\left(e_{\widehat{h}}\right)\right] \\ \text{where } e_{\widehat{h}} = f_{\widehat{h}}\left(\text{Repr}^\bullet_{t_1}(v_{h_1}), \ldots, \text{Repr}^\bullet_{t_m}(v_{h_m})\right)\end{aligned}$$

Example 9 (Running example: completion)**.** We admit the following representation function for the primitive type t_{int}: $\text{Repr}^\bullet_{t_{\text{int}}}(n) = \text{Word}_{64}\left(2 \times n + 1\right)$ (see Section 6). Let us now build $\text{Repr}^\bullet_{t_{\text{ABC}}}(K)$ for each constructor $K \in \{A, B, C\}$:

- $K = A$. Collecting constraints yields this unsolved representation:

$$\begin{aligned}\text{PreRepr}^\bullet_{t_{\text{ABC}}}(A) &= \text{Word}_{64}\left(\blacksquare[0,0] = 1\right) \cap \text{Word}_{64}\left(\blacksquare = 1\right) \\ &= \text{Word}_{64}\left(\blacksquare[0,0] = 1 \wedge \blacksquare = 1\right)\end{aligned}$$

  from which we extract the formula $\phi = (y[0,0] = 1 \wedge y = 1)$ whose unknown is $y \in \{0,1\}^{64}$ (a bitvector of size 64). The model $y = \text{Word}_{64}\left(1\right)$ satisfies ϕ, hence $\text{Repr}^\bullet_{t_{\text{ABC}}}(A) = \text{Word}_{64}\left(1\right)$. Similarly, we obtain $\text{Repr}^\bullet_{t_{\text{ABC}}}(B) = \text{Word}_{64}\left(3\right)$.

- $K = C$. After simplification, the constraints for C (from Example 8) give the following unsolved representation, in which the 64-bit-wide bitvector SMT variable $x_{C(\square)}$ symbolises the representation of the subvalue "under C":

$$\textsc{PreRepr}^{\bullet}_{t_{\text{ABC}}}(C) = \text{Ptr}_{64,3}\left\{\!\!\left\{ \begin{array}{l} \text{Word}_{64}\left(\blacksquare = 2\right), \\ \text{Word}_{64}\left(\blacksquare = x_{C(\square)}\right) \end{array} \right\}\!\!\right\}$$

  The first word description leads to a constant value of 2, the second one is solved by $y = x_{C(\square)}$ and we finally get:

$$\textsc{Repr}^{\bullet}_{t_{\text{ABC}}}(C(n)) = \text{Ptr}_{64,3}\left\{\!\!\left\{\text{Word}_{64}\left(2\right), \text{Word}_{64}\left(\textsc{Repr}^{\bullet}_{t_{int}}(n)\right)\right\}\!\!\right\}$$

♦

5.3 Synthesis of Pattern Matching Primitives

We have previously asserted that the specification yield a consistent memory layout and synthesised a representation function to build memory values? We finally synthesise the pattern matching primitives introduced in Section 3.2.

$\textsc{Unwrap}^{\bullet}_t$: *SubTerms*$(t) \to$ *Exprs*
$\textsc{Unwrap}^{\bullet}_t(h)$ builds an expression to access the representation of the subvalue at position h in a value of type t. The input ingredient $\textsc{Position}^{\bullet}_t(h)$ returns a memory context $\widehat{h}$ providing this information. To obtain $\textsc{Unwrap}^{\bullet}_t(h)$, we translate $\widehat{h}$ to a *target expression*, as defined in Fig. 5. For instance, if $\textsc{Position}^{\bullet}_t(h) = \left\{\!\!\left\{\_, \ldots, \underset{i}{\square}, \ldots, \_\right\}\!\!\right\}$, then $\textsc{Unwrap}^{\bullet}_t(h) = \Delta.i$.

$\textsc{InspectAndSplit}^{\bullet}_t$: (*Constrs*$(t) \to$ *Trees*$) \to$ *Trees*
$\textsc{InspectAndSplit}^{\bullet}_t(\mathit{branch})$ builds a decision tree given a function *branch* from t constructors to decision trees. This primitive is more complex to build, and requires careful exploration of t discriminants to emit the minimal amount of switch nodes. We proceed by iteratively generating intermediate decision trees for decreasing sets of possible constructors, starting from all constructors of t. We then glue these intermediate decision trees with switch nodes.

More precisely, we define the decision tree $\mathcal{T}_{\mathcal{K}}$ by induction on the current set of constructors $\mathcal{K}$:

- If $\mathcal{K} = \{K\}$ contains a single constructor,
  we return $\mathcal{T}_{\mathcal{K}} = \mathit{branch}(K)$.
- Otherwise, let us consider H the set of discriminants which effectively distinguish values in $\mathcal{K}$:

$$H = \left\{ \widehat{h} \;\middle|\; \begin{array}{l} (\widehat{h}, \mathit{split}) \in \textsc{Discrs}^{\bullet}_t \\ \exists K, K' \in \mathcal{K} \cap \mathit{split}, \mathit{split}(K) \neq \mathit{split}(K') \end{array} \right\}$$

  H might contain several discriminants. We choose the "shallowest" one, i.e. that doesn't look far into the representation, and name it $\widehat{h}_0$. Thanks to the precondition on $\textsc{Discrs}^{\bullet}$, we show (in Appendix C) that this shallowest discriminant always exists. By our distinguishability constraints, $\widehat{h}_0$ is a valid context for all values whose constructor is in $\mathcal{K}$.
  We can now build our switch node. Let e the target expression returning the value located at $\widehat{h}_0$ (built similarly to $\textsc{Unwrap}^{\bullet}$). By definition, the possible values of e are $V = \{z \mid K \in \mathcal{K} \cap \mathit{split}, \mathit{split}(K) = z\}$. For each z_i in V, let $\mathcal{K}_{z_i} = \{K \in \mathcal{K} \mid \mathit{split}(K) = z_i \vee K \notin \mathit{split}\}$ the constructors compatible with this result. Let $\mathcal{K}_{\top} = \{K \in \mathcal{K} \mid K \notin \mathit{split}\}$ the remaining constructors. We finally obtain:

$$\mathcal{T}_{\mathcal{K}} = \mathsf{switch}\,(e) \begin{cases} z_0 \mapsto \mathcal{T}_{\mathcal{K}_{z_0}} \\ \vdots \\ z_{m-1} \mapsto \mathcal{T}_{\mathcal{K}_{z_{m-1}}} \\ \top \mapsto \mathcal{T}_{\mathcal{K}_{\top}} \end{cases} \quad \text{where } z_i \in V$$

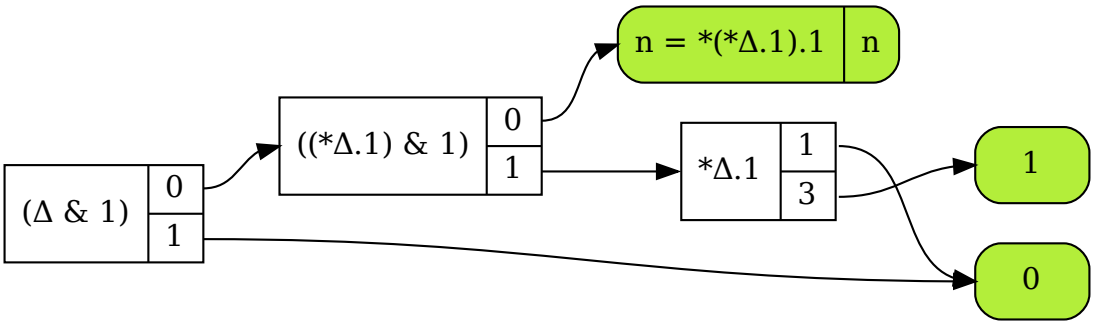

Figure 10: Output of `ribbit` for our running example $\textsc{Compile}_t(m_0)$ with the OCaml representation.

Appendix C shows the detailed algorithm and its proof.

Finally, our *representation-parametrised* $\textsc{Compile}_t$ function presented in Section 3.2 is complete and can compile any pattern matching on t values to an equivalent decision tree. We state its correctness w.r.t. the memory layout:

Theorem 1. *Let Γ a type environment; $t \in \Gamma$, m a matching such that $\Gamma \vdash m : t \to \Gamma'$ and $\mathcal{T} = \textsc{Compile}_t(m)$. Then for any value v of type t such that $m \rhd v \to i, \sigma$, the evaluation of $\mathcal{T}$ on $\textsc{Repr}^\bullet_t(v)$ succeeds with $i, \{x \mapsto \textsc{Repr}^\bullet_{\Gamma(x)}\sigma(x)\}$.*

Example 10 (Running example: $\textsc{InspectAndSplit}^\bullet$ generation)**.** Let us build $\mathcal{T}_{A,B,C}$. The shallowest discriminant (See Example 6) is $D_{AB,C}$ that inspects the lowest bit of the representation, thus $e = \Delta \;\&\; 1$ that has values 0 (for C) or 1 (for A or B). The recursion for C gives $branch(C)$. The recursion using $D_{A,B}$ is similar. We finally obtain the helper function:

$$\textsc{InspectAndSplit}^\bullet_{t_{\text{ABC}}}(branch) =_{\text{def}}$$
$$\mathsf{switch}\,(\Delta \;\&\; 1)\begin{cases} 0 \mapsto & branch(C) \\ 1 \mapsto & \mathsf{switch}\,(\Delta)\begin{cases} 1 \mapsto & branch(A) \\ 3 \mapsto & branch(B) \end{cases} \end{cases}$$

After generating all helper functions, a final call to $\textsc{Compile}_t(m_0)$ produces the final decision tree depicted in Fig. 10. Notice that switch nodes now use concrete expressions operating on memory values to extract subvalues' representations and distinguish between constructors.

For space reasons, we only gave a partial development of our running example in the OCaml representation. For the (far too extensive) details see Appendix D. ⧫

6 Extensions

So far, we only showed the basic pattern constructs: sum types. Modern pattern languages have many additional constructs. We now sketch several extensions of our framework. These extensions are implemented in our prototype.

Reference and product types References and product are degenerate cases of sum types with only one case. Our framework trivially extends to these constructs, with the simplification that $\textsc{Discrs}^\bullet$ is always empty for these types (as there is nothing to be distinguished).

Primitive types Users may wish to pattern match on primitive types. At first glance, primitive types are similar to sum types where each constant is a unit constructor. Unfortunately, the number of distinct values (2^{64} for native integers) makes such definition of $\textsc{Discrs}^\bullet$ too large and the solving step intractable. Furthermore, primitives types are not always finite (big integers).

To alleviate this issue, we extend the notion of $\textsc{Discrs}^\bullet$ to work on *sets of values* which should be treated the same, as if we had several sum type constructors with a single field containing the individual value. Formally, for each T a primitive type, we require an additional ingredient $\textsc{ValSets}^\bullet_T$ which defines a partition of the values of T. We then require discriminants to work on these sets of values: $\textsc{Discrs}^\bullet_T \subset \widehat{\mathit{Contexts}} \times (\textsc{ValSets}^\bullet_T \to \mathbb{Z})$. The rest of our synthesis procedure readily extends to these additional specifications.

Bit-stealing A more interesting extension is an optimisation known as *bit-stealing* in which address alignment bits are used to store additional integer values. For instance, our RBT example can be represented even more compactly by using aligned pointers and storing the color in the lowest bit. Our framework already support leveraging pointer alignment: the OCaml representation exploits the fact that the lowest bit of pointers is always 0. To bit-steal, we extend pointers appropriately: $\mathrm{Ptr}_{\ell,a}(z) \twoheadrightarrow \widehat{v}$ is a ℓ-wide pointer to $\widehat{v}$ with z in the a lowest-bits. Pattern intersection and solving are easily adapted.

7 Related Works

7.1 Memory Representation and ADTs

Memory representation in functional polymorphic garbage-collected languages was identified quickly as an important area for performance improvements. Peterson (1989) proposes techniques to avoid tagging, while Leroy (1992); Jones and Launchbury (1991) suggest ways to unbox values. Our work encourages new development in this area, by allowing to combine these works with efficient pattern matching compilation. Leroy (1990) presents a calculus which can mix a uniform polymorphic representation and monomorphic optimised representation, which we could use to make several representations cohabit. Colin et al. (2018) details how to extend our source language to handle recursive types in the presence of unboxing. Many of these works are implemented in some capacity in OCaml and Haskell.

Iannetta et al. (2021); Koparkar et al. (2021) propose drastically different representations for Algebraic Data Types, where almost everything is flattened, allowing excellent cache behaviour and parallelism. Our work would augment these approaches with efficient decision tree generation.

7.2 Pattern Matching Compilation

Pattern languages for algebraic data types were first introduced by the HOPE language Burstall et al. (1980). Its general form has been adopted mostly as-is in mainstream languages with rich static typing such as Haskell, OCaml, F#, Scala or Rust, but also more recently in more general languages such as Python and soon Java. This diversity of host languages, with their very varied compilation techniques and memory representations, make our framework all the more relevant.

We focused on the core of pattern matching language, with some minor extensions like disjunctive patterns. Other extensions include ranges, guards, matching of polymorphic variants Garrigue (1998), and exception patterns (in recent OCaml versions). These are orthogonal to our work.

As for pattern-matching compilation, many works exists since the first introduction of pattern matrices in the context of the LML language Augustsson (1985). Fessant and Maranget (2001) first proposed optimisation for backtracking automata, introducing the "row and column" approach to split the pattern matrix. Their technique is currently used in OCaml. This approach was later refined by Maranget (2008) to produce good decision trees, which we base our work on. It delivers excellent performance, while being reasonable to compute in practice. Most approaches to improve the current state-of-art algorithms rely on heuristics for the choice of column to split A study of heuristics is done in Scott and Ramsey (2000). Both conclude that the choice of heuristic only has minor performance consequences in most cases, but can matter for very particular matches. We believe the choice of memory representation has a much bigger impact on performance.

8 Conclusion

We have presented a specification and synthesis framework for *composable* memory representation, suitable for Algebraic Data Types implementation and compilation. Our framework not only synthesise optimising compilation procedure, but also checks the valid of the provided specification. As a case study, we used our method to recover a complete description for the OCaml representation and associated pattern-matching compilation. To our knowledge, this is the first generic specification of memory representation linked with Algebraic Data Types compilation purpose. We have also implemented our technique in a prototype tool called `ribbit` and shown its output on concrete examples.

Our technique paves the way towards the formalisation and description of new optimisation techniques for memory representation. In recent versions, Rust has been introducing more and more complex memory representation optimisations, which are so far unspecified and could benefit from our synthesis framework. Furthermore, we believe there is a trove of optimisations yet to be explored when it comes to the memory representation of Algebraic Data Types. Some promising leads are to apply super-optimisation to individual performance-sensitive data structures, or to allow programmers to specify whether types should be optimised for space, cache behaviour, or even sharing. We hope this work serves as a stepping stone for these further optimisations.

References

Lennart Augustsson. 1985. Compiling Pattern Matching. In *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings (Lecture Notes in Computer Science, Vol. 201)*, Jean-Pierre Jouannaud (Ed.). Springer, 368–381. https://doi.org/10.1007/3-540-15975-4_48

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A Source patterns

In this appendix, we give an *exhaustive* presentation of our source pattern language, including its complete formal semantics. It includes all the extensions described in Section 6. This extends Section 3.1.

We first recall the complete grammar in Fig. 11.

Dynamic Semantics Fig. 12 defines the dynamic operational semantics of patterns. $p \rhd v \to \sigma$ stands for "p matches value v and binds the variables in σ". Most of the rules are straightforward, with the following peculiarities:

- The bound value environment returned by the matching is populated by variables, through the VAR rule.
- We enforce that there is no shadowing: variables must be bound only once, as asserted by the side-condition in the CONSTRUCTOR rules.
- Alternatives are left-leaning: we first try to match with the left branch (ALTL rule) before trying the right branch (ALTR rule).

We also define the matching judgement in rule MATCHING: Match $\{p_1 \mid \cdots \mid p_n\} \rhd v \to i, \sigma$ which behaves as the normal judgement, but additionally returns the index of the branch which was matched. In a full language, it would then trigger the evaluation of the body of the branch in question.

Typing Fig. 13 defines the typing judgement $\Gamma \vdash p : t \to \Gamma'$: "pattern p has type t and binds variables whose types are defined in environment Γ'". Typing follows the semantics closely:

- As before, the bound typing environment Γ' is populated by variables through the VAR rule.
- We enforce that there is no shadowing in the TUPLE and CONSTRUCTOR rules.
- Bound environments must be identical in all branches, as enforced by the ALT rule.

B Memory patterns

Memory patterns are memory value skeletons in which possible values of address alignment bits and word contents are constrained by SMT formulae. They are notably used for structural interpolation, in Section 5.1. In this section, we flesh out their auxiliary operations, notably union, intersection and instanciation.

We first recall their grammar in Fig. 14. $t@h$ denotes the application of $\text{PREREPR}^\bullet_t$ to the subvalue at position h. Note the extension of pointer pattern to accommodate for bit-stealing. Other cases closely follow the grammar of memory values: fixed-width words whose contents are specified by an SMT formula ϕ, fixed-width and fixed-alignment pointers (to another memory pattern) whose address alignment bits are specified by an SMT formula ϕ, blocks containing a finite number of fields (which are either memory patterns or the wildcard $\_$) and disjoint unions of memory patterns. ϕ is either $\top$ (always true) or a conjunction of clauses $\bigwedge_i(\text{ops}_i = x_i)$ where each x_i is either a concrete integer value or a subterm symbol and each ops_i is a sequence of operations applied to the word contents.

Union and intersection Memory patterns are equipped with intersection and union operations.

The intersection of two memory patterns, defined in Fig. 15, is empty if they are incompatible (that is, if they specify different memory structures) or another memory pattern that captures both patterns' structural information and word contents otherwise. Intersection performs width extension: the intersection of two words is a word of the same width as the widest initial word. Also note that $(t@h)$ only intersects with memory patterns that do not constrain memory values more than $\text{PREREPR}^\bullet_t(\top)$ – indeed, $(t@h)$ denotes any memory value that matches $\text{PREREPR}^\bullet_t(\top)$.

The union of two memory patterns, defined in Fig. 16, merges their common memory structures and adds disjoint unions between incompatible structures and different SMT formulae. The resulting memory pattern captures all information that applies to any memory value that matches either of the initial patterns.

Instantiation *Memory context instantiation*, defined in Fig. 17, formalizes the substitutions in memory patterns. $\widehat{h}[\widehat{p}]$ is a memory pattern whose structure is that of the memory context $\widehat{h}$, where the holes has been replaced by $\widehat{p}$ enhanced with the inner word operations.

C Decision tree generation algorithm for InspectAndSplit•

In Section 5.3, we give a definition of $\textsc{InspectAndSplit}_t^\bullet$ using a recursive algorithm to build decision trees. This definition uses the concept of *shallowest* context, without defining it. In this appendix, we flesh out this definition and formalize and prove the algorithm to build decision trees.

C.1 Memory context relations

We first define a *partial prefix order* $\prec$ on memory contexts in Fig. 18. Intuitively, if $\widehat{h} \prec \widehat{h}'$, then any memory value that is compatible with $\widehat{h}'$ is also compatible with $\widehat{h}$. Accordingly, we assert that any transformation on words is a prefix of pointer dereferencing, as the former is compatible with any word or pointer of appropriate width whereas the latter only applies to pointers.

Two memory contexts may be "compatible" in that they both apply to at least one common memory value, without one being a prefix of the other. We formalise this relation in Fig. 19.

We also define a "semi-ordered compatibility" relation $\rtimes$ from which we derive a notion of *minimum memory context*:

$$\widehat{h} \rtimes \widehat{h}' \iff \widehat{h} = \widehat{h}' \vee \widehat{h} \prec \widehat{h}' \vee \widehat{h} \bowtie \widehat{h}' \qquad\qquad \min \widehat{H} = \left\{ \widehat{h} \in \widehat{H} \mid \forall \widehat{h}' \in \widehat{H}, \widehat{h} \rtimes \widehat{h}' \right\}$$

Finally, the following results are needed to prove the existence of at least one minimal relevant discriminant:

Lemma 1. *Let $\widehat{H}$ a non-empty, finite set of memory contexts and $\widehat{P}$ a set of memory patterns instantiating all contexts of $\widehat{H}$ (i.e., $\forall \widehat{h} \in \widehat{H}, \exists \widehat{p} \in \widehat{P}, \exists \widehat{p}', \widehat{p} = \widehat{h}[\widehat{p}']$). Then we have*

$$\bigcap_{\widehat{p} \in \widehat{P}} \widehat{p} \neq \emptyset \Rightarrow \min \widehat{H} \neq \emptyset$$

Proof. We first show that for any memory contexts $\widehat{h}$ and $\widehat{h}'$ and any memory patterns $\widehat{p}$ and $\widehat{p}'$ such that $\widehat{h}[\widehat{p}] \cap \widehat{h}'[\widehat{p}'] \neq \emptyset$, we have $\widehat{h} \rtimes \widehat{h}'$ or $\widehat{h}' \rtimes \widehat{h}$. This result is immediate by induction on pattern intersection rules.

Therefore, for all $\widehat{h}, \widehat{h}' \in \widehat{H}$, we have $\widehat{h} \rtimes \widehat{h}'$ or $\widehat{h}' \rtimes \widehat{h}$. Since $\widehat{H}$ is a finite ordered set, there exists at least one $\widehat{h} \in \widehat{H}$ such that $\forall \widehat{h}' \in \widehat{H}, \widehat{h}' \not\prec \widehat{h}$, that is $\forall \widehat{h}' \in \widehat{H}, \widehat{h}' \rtimes \widehat{h}$, which proves the existence of at least one minimal memory context of $\widehat{H}$. ◊

Lemma 2. *Let $\widehat{h}_0, \widehat{h}_1, \widehat{h}_2$ and $\widehat{p}, \widehat{p}'$ such that $\widehat{h}_0 \rtimes \widehat{h}_1$ and $\widehat{h}_1[\widehat{p}] \cap \widehat{h}_2[\widehat{p}'] \neq \emptyset$. Then we have $\widehat{h}_0 \rtimes \widehat{h}_2$.*

Proof. By induction on memory pattern intersection. ◊

C.2 Decision tree generation algorithm

Recall the definition of $\textsc{InspectAndSplit}_t^\bullet(\mathit{branch})$ where t is a sum type and *branch* a function from t constructors to decision trees: $\textsc{InspectAndSplit}_t^\bullet(\mathit{branch}) = \mathcal{T}_{\mathit{Constrs}(t)}$, where $\mathcal{T}_\mathcal{K}$ is defined as follows for a set of constructors $\mathcal{K}$:

- if $\mathcal{K}$ is empty, it is undefined and we omit any decision tree branch leading to $\mathcal{T}_\emptyset$;
- if $\mathcal{K}$ contains one constructor K, we branch to its associated continuation with $\mathcal{T}_{\{K\}} = \mathit{branch}(K)$;
- otherwise, we pick a *minimal relevant discriminant* $(\widehat{h}, \mathit{split}) \in \textsc{Discrs}_t^\bullet$ such that

$$\widehat{h} \in \min \left\{ \widehat{h} \;\middle|\; \begin{array}{l} (\widehat{h}, \mathit{split}) \in \textsc{Discrs}_t^\bullet \\ \exists K, K' \in \mathcal{K} \cap \mathit{split}, \mathit{split}(K) \neq \mathit{split}(K') \end{array} \right\}$$

Let e the target expression returning the value located at $\widehat{h}_0$ (built similarly to $\textsc{Unwrap}^\bullet$). We build the following switch node:

$$\mathcal{T}_\mathcal{K} = \mathsf{switch}\,(e) \left\{ c \mapsto \mathcal{T}_{\mathcal{K}(c)} \;\middle|\; \begin{array}{l} c \in \mathbb{Z} \cup \{\top\} \\ \mathcal{K}_c \neq \emptyset \end{array} \right\}$$

with $\mathcal{K}_z = \{K \in \mathcal{K} \mid (K, z) \in \mathit{split} \vee K \notin \mathit{split}\}$ for $z \in \mathbb{Z}$ and $\mathcal{K}_\top = \{K \in \mathcal{K} \mid K \notin \mathit{split}\}$.

The correctness of this algorithm relies on the two following results.

Lemma 3 (Strictly decreasing constructor set)**.** *Let $\mathcal{K} \subseteq \mathit{Constrs}(t)$ with $|\mathcal{K}| > 1$. For any source value v whose head constructor K is in $\mathcal{K}$, the execution of $\mathcal{T}_{\mathcal{K}}$ on $\textsc{Repr}_t^\bullet(v)$ branches to a recursively defined decision tree $\mathcal{T}_{\mathcal{K}'}$ with $K \in \mathcal{K}' \subsetneq \mathcal{K}$.*

Proof. Let $(\widehat{h}, \mathit{split})$ the minimal relevant discriminant picked for this step. Let v a source value whose head constructor K is in $\mathcal{K}$. Let us first remark that in any case, the execution of $\mathcal{T}_{\mathcal{K}}$ on $\textsc{Repr}_t^\bullet(v)$ branches to $\mathcal{T}_{\mathcal{K}(c)}$ for some $c \in \mathbb{Z} \cup \{\top\}$.

We first show that each such set $\mathcal{K}(c)$ is a strict subset of $\mathcal{K}$. By definition of $(\widehat{h}, \mathit{split})$, there are at least two distinct constructors K and K' in $\mathcal{K} \cap \mathit{split}$ such that $\mathit{split}(K) \neq \mathit{split}(K')$. It follows that at least one of these constructors is absent from each $\mathcal{K}(c)$ for $c \in \mathbb{Z} \cup \{\top\}$, hence $\mathcal{K}(c) \subsetneq \mathcal{K}$.

We now show that we always have $K \in \mathcal{K}(c)$. This is immediate if $K \notin \mathit{split}$. Otherwise, we have $K \in \mathit{split}$ and we show that the switch node branches to $\mathcal{T}_{\mathcal{K}(\mathit{split}(K))}$: upon evaluation on the input $\textsc{Repr}_t^\bullet(v)$, the switch expression e yields the memory value located at $\widehat{h}$. By construction, $\textsc{Repr}_t^\bullet$ is such that this memory value is a word whose contents are $\mathit{split}(K)$. Indeed, for all $K \in \mathit{split}$, $\textsc{PreRepr}_t^\bullet(K)$ was built from an intersection of memory patterns including $\widehat{p} = \widehat{h}' \left[\mathrm{Word}_\ell \left(\mathrm{ops} = \mathit{split}(K)\right)\right]$ where $\widehat{h}'[\underline{\mathrm{ops}}_{\ell'}] = \widehat{h}$ (ℓ and ℓ' are irrelevant here). This implies that the memory value located at $\widehat{h}'$ in $\textsc{Repr}_t^\bullet(v)$ exists and is a word whose contents ■ are such that $\mathrm{ops} = \mathit{split}(K)$, that is, the memory value located at $\widehat{h}$ in $\textsc{Repr}_t^\bullet(v)$ exists and is a word whose contents are equal to $\mathit{split}(K)$. ◊

Lemma 4 (Existence of at least one minimal relevant discriminant)**.** *Assume that the representation is valid, that is, $\textsc{Repr}_t^\bullet(K)$ exists for each constructor K of t. Then for all $\mathcal{K} \subseteq \mathit{Constrs}(t)$ such that $|\mathcal{K}| > 1$, we have*

$$\min \left\{ \widehat{h} \;\middle|\; \begin{array}{l} (\widehat{h}, \mathit{split}) \in \textsc{Discrs}_t^\bullet \\ \exists K, K' \in \mathcal{K} \cap \mathit{split}, \mathit{split}(K) \neq \mathit{split}(K') \end{array} \right\} \neq \emptyset$$

Proof. According to the distinguishability condition on $\textsc{Discrs}_t^\bullet$, there exists a discriminant $(\widehat{h}, \mathit{split}) \in \textsc{Discrs}_t^\bullet$ such that $K, K' \in \mathit{split}$ and $\mathit{split}(K) \neq \mathit{split}(K')$ for any constructor $K' \neq K$, therefore $\widehat{H}$ is non-empty.

For any constructor K of type t, let

$$\widehat{H}_K = \widehat{H} \cap \left\{ \widehat{h} \;\middle|\; \begin{array}{l} (\widehat{h}, \mathit{split}) \in \textsc{Discrs}_t^\bullet \\ K_i \in \mathit{split} \end{array} \right\}$$

Recall the definition of the distinguishability pattern of any constructor K. Owing to representation validity, we have

$$\widehat{p}_t^d(K) = \bigcap_{(\widehat{h}_i, \ell_i, \mathrm{ops}_i, z_i) \in \mathcal{C}} \widehat{h}_i \left[\mathrm{Word}_{\ell_i} \left(\mathrm{ops}_i = z_i\right)\right] \neq \emptyset$$

where

$$\mathcal{C} = \left\{ \begin{array}{l} (\widehat{h}, \ell, \mathrm{ops}, z) \text{ where} \\ \quad (\widehat{h}', \mathit{split}) \in \textsc{Discrs}_t^\bullet \\ \quad (K \mapsto z) \in \mathit{split} \\ \quad \widehat{h}[\underline{\mathrm{ops}}_{\ell'}] = \widehat{h}' \\ \quad \ell = (\text{if } (\ell' \text{ is } *) \text{ then } \max\limits_{(K' \mapsto z') \in \mathit{split}} \texttt{width}(z) \text{ else } \ell') \end{array} \right\}$$

Notice that for any relevant discriminant $(\widehat{h}, \mathit{split}) \in \textsc{Discrs}_t^\bullet$ such that $K \in \mathit{split}$, there exists $(\widehat{h}_i, \ell_i, \mathrm{ops}_i, z_i) \in$

$\mathcal{C}$ such that $\widehat{h} = \widehat{h}_i[\underline{\lfloor \text{ops}_i \rfloor_{\ell'_i}}]$; we can write

$$\begin{aligned}
&\widehat{h}_i[\text{Word}_{\ell_i}\,(\text{ops}_i = z_i)] \\
&= \widehat{h}_i\left[\underline{\lfloor \text{ops}_i \rfloor_{\ell'_i}}[\text{Word}_{\ell_i}\,(\blacksquare = z_i)]\right] \\
&= \left(\widehat{h}_i[\underline{\lfloor \text{ops}_i \rfloor_{\ell'_i}}]\right)[\text{Word}_{\ell_i}\,(\blacksquare = z_i)] \\
&= \widehat{h}[\text{Word}_{\ell_i}\,(\blacksquare = z_i)]
\end{aligned}$$

That is, each considered memory context is instantiated once in this non-empty pattern intersection. We can thus apply Lemma 1 to show that for all $K \in \mathcal{K}$, $\min \widehat{H}_K \neq \emptyset$.

We must now prove that $\min \widehat{H} = \min\left\{\min \widehat{H}_K\right\}_{K\in\mathcal{K}}$ is not empty. Let $K, K' \in \mathcal{K}$ such that $K \neq K'$. Using the same distinguishability condition argument as before, we have $\widehat{H}_K \cap \widehat{H}_{K'} \neq \emptyset$ and apply Lemma 2 on both constructors' distinguishability patterns to show that any minimal context for either constructor is compatible with all minimal contexts for the other constructor, and thus $\min\left(\min \widehat{H}_K \cup \min \widehat{H}_{K'}\right) \neq \emptyset$. $\Diamond$

Using these two results, we show that $\textsc{InspectAndSplit}^\bullet_t$ descends through switch nodes that successively restrict the set of potential constructors, ending with the decision tree associated with the specific identified constructor.

We can thus assert the correctness of our pattern matching compilation scheme:

Theorem 2. *Let Γ a type environment; let $t \in \Gamma$ a type. Let m a matching such that $\Gamma \vdash m : t \to \Gamma'$ and $\mathcal{T} = \textsc{Compile}_t(m)$. Then for any value v of type t such that $m \rhd v \to i, \sigma$, the evaluation of $\mathcal{T}$ on $\textsc{Repr}^\bullet_t(v)$ succeeds with $i, \{x \mapsto \textsc{Repr}^\bullet_{\Gamma(x)}\sigma(x)\}$.*

D Full example: OCaml representation

Recall the type environment from our running example:

$$\begin{aligned}
\Gamma = \{\ & t \mapsto \mathit{None} + \mathit{Some}(t_{\text{ABC}}); \\
& t_{\text{ABC}} \mapsto A + B + C(t_{\text{int}}); \\
& t_{\text{int}} \mapsto u32\}
\end{aligned}$$

In this section, we detail the complete representation synthesis pipeline and pattern matching compilation for the OCaml representation, *with all extensions enabled*. (Note that the bit-stealing extension is not actually used in this representation, which is why all addresses' alignment bits are set to 0.)

D.1 Input specification

D.2 Primitive type t_{int}

In OCaml, all integer values follow the same memory layout. Let $S = [0; 2^{32} - 1]$ the set of $u32$ values. We encode any integer value $n \in S$ on 64 bits, then transform it through a one-bit left shift and an increment to set its pointer flag (lowest bit) to 1. As S is the only defined value set and no other representation constraints apply, no discriminant is needed for t_{int}. We thus define the following ingredients:

$$\textsc{ValSets}^\bullet_{t_{\text{int}}} = \{S\} \qquad \textsc{Discrs}^\bullet_{t_{\text{int}}} = \emptyset \qquad \textsc{Position}^\bullet_{t_{\text{int}}}(S(\square)) = \underline{\lfloor (\blacksquare - 1)/2[0, 31] \rfloor_{64}}$$

D.3 Sum types t_{ABC} and t

Sum types ingredients are similar across individual types. The main distinction done is between unit and non-unit constructors, which are represented by 64-bit-wide words and 64-bit-wide pointers to blocks, respectively.

We define the following ingredients for t_{ABC}:

$$\textsc{Position}^{\bullet}_{t_{\text{ABC}}}(C(\square)) = \text{Ptr}_{64,3}\left\{\!\!\left\{ \_, \llcorner\blacksquare\lrcorner_{*}, \ldots \right\}\!\!\right\}$$

$$\textsc{Discrs}^{\bullet}_{t_{\text{ABC}}} = \left\{ \begin{array}{l} D_{AB,C} = \left(\underline{\llcorner 0[\blacksquare, 0] \lrcorner}_{*}, \left\{ \begin{array}{ll} A \mapsto 1 & B \mapsto 1 \\ C \mapsto 0 & \end{array} \right\} \right), \\ D_{A,B} = \left(\llcorner\blacksquare\lrcorner_{64}, \left\{ A \mapsto 1 \quad B \mapsto 3 \right\} \right), \\ D_{C} = \left(\text{Ptr}_{64,3}\left\{\!\!\left\{ \llcorner\blacksquare\lrcorner_{64}, \ldots \right\}\!\!\right\}, \{C \mapsto 2\} \right) \end{array} \right\}$$

We define the following ingredients for t:

$$\textsc{Position}^{\bullet}_{t}(\mathit{Some}(\square)) = \text{Ptr}_{64,3}\left\{\!\!\left\{ \_, \llcorner\blacksquare\lrcorner_{*}, \ldots \right\}\!\!\right\}$$

$$\textsc{Discrs}^{\bullet}_{t} = \left\{ \begin{array}{l} D_{\mathit{None},\mathit{Some}} = \left(\underline{\llcorner 0[\blacksquare, 0] \lrcorner}_{*}, \left\{ \begin{array}{l} \mathit{None} \mapsto 1 \\ \mathit{Some} \mapsto 0 \end{array} \right\} \right), \\ D_{\mathit{None}} = \left(\llcorner\blacksquare\lrcorner_{64}, \{\mathit{None} \mapsto 1\} \right), \\ D_{\mathit{Some}} = \left(\text{Ptr}_{64,3}\left\{\!\!\left\{ \llcorner\blacksquare\lrcorner_{64}, \ldots \right\}\!\!\right\}, \{\mathit{Some} \mapsto 1\} \right) \end{array} \right\}$$

D.4 Structural interpolation

Example 11 (Building the unsolved representation of a primitive type: t_{int})**.** t_{int} only has one value set S. No discriminant applies: by convention, we define $\widehat{p}^{d}_{t_{\text{int}}}(S) = \_$. We derive its only composability pattern from $\textsc{Position}^{\bullet}_{t_{\text{int}}}(S(\square))$ to build its unsolved representation:

$$\begin{array}{rl} \textsc{PreRepr}^{\bullet}_{t_{\text{int}}}(\top) & = \textsc{PreRepr}^{\bullet}_{t_{\text{int}}}(S) \\ & = \widehat{p}^{c}_{t_{\text{int}}}(S) = \text{Word}_{64}\left(((\blacksquare - 1)/2)[0,31] = x_{\square}\right) \\ \textsc{ReprTy}^{\bullet}_{t_{\text{int}}}(\top) & = \textsc{ReprTy}^{\bullet}_{t_{\text{int}}}(S) = \text{Word}_{64} \end{array}$$

♦

Example 12 (Composing word representations: t_{ABC})**.** We first compute the unsolved representations of both unit constructors A and B, which solely consists of their distinguishability pattern. Two discriminants apply: $D_{A,B}$ and $D_{AB,C}$. We decompose and recombine $D_{AB,C}$ into the memory pattern $\text{Word}_1\,(\blacksquare[0,0] = 1)$ and get

$$\begin{array}{rl} \textsc{PreRepr}^{\bullet}_{t_{\text{ABC}}}(A) = \widehat{p}^{d}_{t_{\text{ABC}}}(A) & \\ & = \text{Word}_{64}\,(\blacksquare = 1) \cap \text{Word}_1\,(\blacksquare[0,0] = 1) \\ & = \text{Word}_{64}\,(\blacksquare = 1 \wedge \blacksquare[0,0] = 1) \\ \textsc{PreRepr}^{\bullet}_{t_{\text{ABC}}}(B) = \widehat{p}^{d}_{t_{\text{ABC}}}(B) & \\ & = \text{Word}_{64}\,(\blacksquare = 3) \cap \text{Word}_1\,(\blacksquare[0,0] = 1) \\ & = \text{Word}_{64}\,(\blacksquare = 3 \wedge \blacksquare[0,0] = 1) \end{array}$$

We therefore have $\textsc{ReprTy}^{\bullet}_{t_{\text{ABC}}}(A) = \textsc{ReprTy}^{\bullet}_{t_{\text{ABC}}}(B) = \text{Word}_{64}$.

For the non-unit constructor C, we need both a composability and a distinguishability memory pattern. Two discriminants apply: D_C and $D_{AB,C}$, hence

$$\begin{array}{rl} \widehat{p}^{d}_{t_{\text{ABC}}}(C) = & \text{Ptr}_{64,3}(\top) \twoheadrightarrow \left\{\!\!\left\{ \text{Word}_{64}\,(\blacksquare = 2), \ldots \right\}\!\!\right\} \\ & \cap\ \text{Word}_1\,(\blacksquare[0,0] = 0) \\ & = \text{Ptr}_{64,3}(\top \wedge \blacksquare[0,0] = 0) \twoheadrightarrow \left\{\!\!\left\{ \text{Word}_{64}\,(\blacksquare = 2), \ldots \right\}\!\!\right\} \end{array}$$

We derive the composability pattern from $\textsc{Position}^{\bullet}_{t_{\text{ABC}}}(C(\square))$ and $\textsc{ReprTy}^{\bullet}_{t_{\text{int}}}(\top) = \textsc{ReprTy}^{\bullet}_{t_{\text{int}}}(S) = \text{Word}_{64}$:

$$\widehat{p}^{c}_{t_{\text{ABC}}}(C(\square)) = \text{Ptr}_{64,3}(\top) \twoheadrightarrow \left\{\!\!\left\{ \_, \text{Word}_{64}\left(\blacksquare = x_{C(\square)}\right), \ldots \right\}\!\!\right\}$$

Note that the representation of the subvalue "under C" has been integrated as an SMT variable $x_{C(\square)}$. Hence

$$\begin{aligned}\textsc{PreRepr}^{\bullet}_{t_{\text{ABC}}}(C) &= \widehat{p}^{d}_{t_{\text{ABC}}}(C) \cap \widehat{p}^{c}_{t_{\text{ABC}}}(C(\square)) \\ &= \begin{array}{l}\text{Ptr}_{64,3}(\top \wedge \blacksquare[0,0] = 0 \wedge \top) \\ \multimap \left\{\!\!\left\{ \text{Word}_{64}\left(\blacksquare = 2\right), \text{Word}_{64}\left(\blacksquare = x_{C(\square)}\right) \right\}\!\!\right\}\end{array}\end{aligned}$$

and $\textsc{ReprTy}^{\bullet}_{t_{\text{ABC}}}(C) = \text{Ptr}_{64,3}\left\{\!\!\left\{\text{Word}_{64}, \text{Word}_{64}\right\}\!\!\right\}$.

We finally compute the union of these partial representations to get the generic unsolved representation and its memory type:

$$\begin{aligned}&\textsc{PreRepr}^{\bullet}_{t_{\text{ABC}}}(\top) \\ &= \textsc{PreRepr}^{\bullet}_{t_{\text{ABC}}}(A) \cup \textsc{PreRepr}^{\bullet}_{t_{\text{ABC}}}(B) \cup \textsc{PreRepr}^{\bullet}_{t_{\text{ABC}}}(C) \\ &= \textsc{PreRepr}^{\bullet}_{t_{\text{ABC}}}(A) \sqcup \textsc{PreRepr}^{\bullet}_{t_{\text{ABC}}}(B) \sqcup \textsc{PreRepr}^{\bullet}_{t_{\text{ABC}}}(C) \\ &\textsc{ReprTy}^{\bullet}_{t_{\text{ABC}}}(\top) \\ &= \textsc{ReprTy}^{\bullet}_{t_{\text{ABC}}}(A) \cup \textsc{ReprTy}^{\bullet}_{t_{\text{ABC}}}(B) \cup \textsc{ReprTy}^{\bullet}_{t_{\text{ABC}}}(C) \\ &= \text{Word}_{64} \cup \text{Word}_{64} \cup \text{Ptr}_{64,3}\left\{\!\!\left\{\text{Word}_{64}, \text{Word}_{64}\right\}\!\!\right\} \\ &= \text{Word}_{64} \sqcup \text{Ptr}_{64,3}\left\{\!\!\left\{\text{Word}_{64}, \text{Word}_{64}\right\}\!\!\right\}\end{aligned}$$

⧫

Example 13 (Composing non-word representations: t)**.** The unsolved representation of the unit constructor *None* is very similar to that of A and B from the previous example: two discriminants D_{None} and $D_{None,Some}$ apply and we get

$$\begin{aligned}\textsc{PreRepr}^{\bullet}_{t}(\mathit{None}) &= \widehat{p}^{d}_{t}(\mathit{None}) \\ &= \text{Word}_{64}\left(\blacksquare = 1\right) \cap \text{Word}_{1}\left(\blacksquare[0,0] = 1\right) \\ &= \text{Word}_{64}\left(\blacksquare = 1 \wedge \blacksquare[0,0] = 1\right) \\ \textsc{ReprTy}^{\bullet}_{t}(\mathit{None}) &= \text{Word}_{64}\end{aligned}$$

The unsolved representation of the non-unit constructor *Some* is the intersection of its only composability pattern and of its distinguishability pattern, involving two discriminants D_{Some} and $D_{None,Some}$.

$$\begin{aligned}\widehat{p}^{d}_{t}(\mathit{Some}) &= \text{Ptr}_{64,3}(\top) \multimap \left\{\!\!\left\{\text{Word}_{64}\left(\blacksquare = 1\right), \ldots\right\}\!\!\right\} \\ &\quad \cap \text{Word}_{1}\left(\blacksquare[0,0] = 0\right) \\ &= \text{Ptr}_{64,3}(\top \wedge \blacksquare[0,0] = 0) \multimap \left\{\!\!\left\{\text{Word}_{64}\left(\blacksquare = 1\right), \ldots\right\}\!\!\right\}\end{aligned}$$

Since $\textsc{ReprTy}^{\bullet}_{t_{\text{ABC}}}(\top)$ is not a word type, we integrate the subvalue representation as an application pattern rather than an SMT variable:

$$\widehat{p}^{c}_{t}(\mathit{Some}(\square)) = \text{Ptr}_{64,3}(\top) \multimap \left\{\!\!\left\{\_, t_{\text{ABC}}@\mathit{Some}(\square), \ldots\right\}\!\!\right\}$$

and we get

$$\begin{aligned}&\textsc{PreRepr}^{\bullet}_{t}(\mathit{Some}) = \widehat{p}^{d}_{t}(\mathit{Some}) \cap \widehat{p}^{c}_{t}(\mathit{Some}(\square)) \\ &= \text{Ptr}_{64,3}(\top \wedge \blacksquare[0,0] = 0) \multimap \left\{\!\!\left\{\text{Word}_{64}\left(\blacksquare = 1\right), \ldots\right\}\!\!\right\} \\ &\cap \text{Ptr}_{64,3}(\top) \multimap \left\{\!\!\left\{\_, t_{\text{ABC}}@\mathit{Some}(\square), \ldots\right\}\!\!\right\} \\ &= \begin{array}{l}\text{Ptr}_{64,3}(\top \wedge \blacksquare[0,0] = 0 \wedge \top) \\ \multimap \left\{\!\!\left\{\text{Word}_{64}\left(\blacksquare = 1\right), t_{\text{ABC}}@\mathit{Some}(\square)\right\}\!\!\right\}\end{array} \\ &\textsc{ReprTy}^{\bullet}_{t}(\mathit{Some}) = \text{Ptr}_{64,3}\left\{\!\!\left\{\text{Word}_{64}, \textsc{ReprTy}^{\bullet}_{t_{\text{ABC}}}(\top)\right\}\!\!\right\}\end{aligned}$$

And we can finally build the generic unsolved representation and memory type:

$$\begin{aligned}
\textsc{PreRepr}^\bullet_t(\top) &= \textsc{PreRepr}^\bullet_t(\mathit{None}) \sqcup \textsc{PreRepr}^\bullet_t(\mathit{Some}) \\
&= \mathrm{Word}_{64}\left(\blacksquare = 1 \wedge \blacksquare[0,0] = 1\right) \\
&\quad \sqcup \begin{array}{l} \mathrm{Ptr}_{64,3}(\top \wedge \blacksquare[0,0] = 0 \wedge \top) \\ \twoheadrightarrow \left\{\!\!\left\{ \mathrm{Word}_{64}\left(\blacksquare = 1\right), t_{\mathrm{ABC}} @ \mathit{Some}(\square) \right\}\!\!\right\} \end{array} \\
\textsc{ReprTy}^\bullet_t(\top) &= \textsc{ReprTy}^\bullet_t(\mathit{None}) \sqcup \textsc{ReprTy}^\bullet_t(\mathit{Some}) \\
&= \mathrm{Word}_{64} \sqcup \mathrm{Ptr}_{64,3}\left\{\!\!\left\{ \mathrm{Word}_{64}, \textsc{ReprTy}^\bullet_{t_{\mathrm{ABC}}}(\top) \right\}\!\!\right\}
\end{aligned}$$

♦

D.5 Representation completion

Example 14 (Representation completion of a primitive type: t_{int})**.** We complete the unsolved representation of the only value set of t_{int}. $\textsc{PreRepr}^\bullet_{t_{\mathrm{int}}}(S) = \mathrm{Word}_{64}\left(\phi\right)$ contains one formula at the position $\llcorner\blacksquare\lrcorner_*$ that we express as the following atomic SMT problem:

$$\begin{aligned}
&(\phi)\ y_{\llcorner\blacksquare\lrcorner_*} : \{0,1\}^{32} \to \{0,1\}^{64} \\
&\text{ such that } \forall x_\square : \{0,1\}^{32}, (y_{\llcorner\blacksquare\lrcorner_*}(x_\square) - 1)/2[0,31] = x_\square
\end{aligned}$$

where the variable $x_\square$ symbolises the source integer value as a bitvector of width 32. We immediately solve it with the following model: $y_{\llcorner\blacksquare\lrcorner_*}(x_\square) = (2 \times x_\square + 1)_{64}$, hence $\textsc{Repr}^\bullet_{t_{\mathrm{int}}}(n) = \mathrm{Word}_{64}\left(2 \times n + 1\right)$. ♦

Example 15 (Representation completion of t_{ABC})**.** Recall the unsolved representation patterns from the previous section. $\textsc{PreRepr}^\bullet_{t_{\mathrm{ABC}}}(A) = \mathrm{Word}_{64}\left(\phi\right)$ contains one formula at the position $\llcorner\blacksquare\lrcorner_*$ that we express as the following atomic SMT problem:

$$(\phi)\ y_{\llcorner\blacksquare\lrcorner_*} : \{0,1\}^{64} \text{ such that } y_{\llcorner\blacksquare\lrcorner_*} = 1 \wedge y_{\llcorner\blacksquare\lrcorner_*}[0,0] = 1$$

which we solve with the model $y_{\llcorner\blacksquare\lrcorner_*} = (1)_{64}$, which satisfies ϕ, hence $\textsc{Repr}^\bullet_{t_{\mathrm{ABC}}}(A) = \mathrm{Word}_{64}\left(1\right)$.

We carry out the same steps to solve $\textsc{PreRepr}^\bullet_{t_{\mathrm{ABC}}}(B)$: $y_{\llcorner\blacksquare\lrcorner_*} = 3$ satisfies $y_{\llcorner\blacksquare\lrcorner_*} = 3 \wedge y_{\llcorner\blacksquare\lrcorner_*}[0,0] = 1$ and thus $\textsc{Repr}^\bullet_{t_{\mathrm{ABC}}}(B) = \mathrm{Word}_{64}\left(3\right)$.

Solving $\textsc{PreRepr}^\bullet_{t_{\mathrm{ABC}}}(C)$ is slightly more involved: it contains three formulas at positions $\widehat{h}_0 = \mathrm{Ptr}_{64,3}\left\{\!\!\left\{\llcorner\blacksquare\lrcorner_*, \ldots\right\}\!\!\right\}$, $\widehat{h}_1 = \mathrm{Ptr}_{64,3}\left\{\!\!\left\{\_, \llcorner\blacksquare\lrcorner_*, \ldots\right\}\!\!\right\}$ and $\llcorner\blacksquare\lrcorner_*$, which we express through the following atomic SMT problems:

$$\begin{aligned}
&(\phi_0)\ y_{\widehat{h}_0} \in \{0,1\}^{64} \text{ such that } y_{\widehat{h}_0} = 2 \\
&(\phi_1)\ y_{\widehat{h}_1} \in \{0,1\}^{64} \to \{0,1\}^{64} \\
&\qquad \text{such that } \forall x_{C(\square)} : \{0,1\}^{64}, y_{\widehat{h}_1}(x_{C(\square)}) = x_{C(\square)} \\
&(\phi)\ y_{\llcorner\blacksquare\lrcorner_*} \in \{0,1\}^3 \text{ such that } \top \wedge y_{\llcorner\blacksquare\lrcorner_*}[0,0] = 0 \wedge \top
\end{aligned}$$

We immediately solve these with the following models:

$$\begin{aligned}
&y_{\widehat{h}_0} = (2)_{64} \text{ satisfies } \phi_0 \\
&y_{\widehat{h}_1}(x_{C(\square)}) = x_{C(\square)} \text{ satisfies } \phi_1 \\
&y_{\llcorner\blacksquare\lrcorner_*} = (0)_3 \text{ satisfies } \phi
\end{aligned}$$

and thus get, for any value v of type t_{int}:

$$\textsc{Repr}^\bullet_{t_{\mathrm{ABC}}}(C(v)) = \begin{array}{l} \mathrm{Ptr}_{64,3}(0) \\ \twoheadrightarrow \left\{\!\!\left\{ \mathrm{Word}_{64}\left(2\right), \mathrm{Word}_{64}\left(\textsc{Repr}^\bullet_{t_{\mathrm{int}}}(v)\right) \right\}\!\!\right\} \end{array}$$

♦

Example 16 (Representation completion of t)**.** Recall the unsolved representation patterns from the previous section. $\textsc{PreRepr}_t^\bullet(\mathit{None}) = \mathrm{Word}_{64}(\phi)$ contains one formula at the position $\llcorner\blacksquare\lrcorner_*$ that we express as the following atomic SMT problem:

$$(\phi)\ y_{\llcorner\blacksquare\lrcorner_*} : \{0,1\}^{64} \text{ such that } y_{\llcorner\blacksquare\lrcorner_*} = 1 \wedge y_{\llcorner\blacksquare\lrcorner_*}[0,0] = 1$$

The model $y_{\llcorner\blacksquare\lrcorner_*} = (1)_{64}$, satisfies ϕ, hence $\textsc{Repr}_t^\bullet(\mathit{None}) = \mathrm{Word}_{64}(1)$.

$\textsc{PreRepr}_t^\bullet(\mathit{Some})$ contains three formulas at positions $\widehat{h}_0 = \mathrm{Ptr}_{64,3}\{\!\{\llcorner\blacksquare\lrcorner_*, \ldots\}\!\}$, $\widehat{h}_1 = \mathrm{Ptr}_{64,3}\{\!\{\_, \llcorner\blacksquare\lrcorner_*, \ldots\}\!\}$ and $\llcorner\blacksquare\lrcorner_*$ that we express as the following atomic SMT problems:

$$\begin{aligned}
&(\phi_0)\ y_{\widehat{h}_0} : \{0,1\}^{64} \text{ such that } y_{\widehat{h}_0} = 1\\
&(\phi_1)\ y_{\widehat{h}_1} : \{0,1\}^{64} \to \{0,1\}^{64} \text{ such that } \forall x : \{0,1\}^{64}, y_{\widehat{h}_1}(x) = x\\
&\qquad y_{\llcorner\blacksquare\lrcorner_*} : \{0,1\}^3 \text{ such that } \top
\end{aligned}$$

which we solve with the following models:

$$y_{\widehat{h}_0} = (1)_{64} \text{ satisfies } \phi_0 \qquad y_{\widehat{h}_1}(x) = x \text{ satisfies } \phi_1 \qquad y_{\llcorner\blacksquare\lrcorner_*} = (0)_3 \text{ satisfies } \top$$

and we thus have, for any source value v of type t_{ABC}:

$$\begin{aligned}
&\textsc{Repr}_t^\bullet(\mathit{Some}(v)) =\\
&\mathrm{Ptr}_{64,3}(0) \rightarrowtriangle \{\!\{\mathrm{Word}_{64}(1), \mathrm{Word}_{64}\left(\textsc{Repr}_{t_{\mathrm{ABC}}}^\bullet(v)\right)\}\!\}
\end{aligned}$$

Notice that $\textsc{Repr}_{t_{\mathrm{ABC}}}^\bullet(v)$ corresponds to $x_{\mathit{Some}(\square)}$ in $\textsc{PreRepr}_t^\bullet(\mathit{Some})$. ♦

D.6 Pattern matching primitives

We finally synthesise pattern matching primitives $\textsc{Unwrap}^\bullet$ and $\textsc{InspectAndSplit}^\bullet$ for each sum type.

D.6.1 Composability primitives: $\textsc{Unwrap}^\bullet$

Since the composability ingredients for the only subterm of both sum types t_{ABC} and t are identical, their composability primitives – which are generated by converting a memory context ingredient to a target expression – are also identical.

From the memory context

$$\textsc{Position}_{t_{\mathrm{ABC}}}^\bullet(C(\square)) = \textsc{Position}_t^\bullet(\mathit{Some}(\square)) = \mathrm{Ptr}_{64,3}\{\!\{\_, \llcorner\blacksquare\lrcorner_*, \ldots\}\!\}$$

we obtain

$$\textsc{Unwrap}_{t_{\mathrm{ABC}}}^\bullet(C(\square)) = (*\Delta).1 \qquad \textsc{Unwrap}_t^\bullet(\mathit{Some}(\square)) = (*\Delta).1$$

With both types, the evaluation of this expression on the representation of a non-unit source value yields the representation of its subvalue under the constructor, as expected. Indeed, $*\left(\mathrm{Ptr}_{64,3}(0) \rightarrowtriangle \{\!\{\widehat{v}_0, \widehat{v}_1\}\!\}\right).1$ evaluates to $\widehat{v}_1$.

D.6.2 Distinguishability primitives: $\textsc{InspectAndSplit}^\bullet$

The different signatures of t_{ABC} and t result in distinct decision trees to distinguish between constructors. Indeed, $\textsc{InspectAndSplit}_{t_{\mathrm{ABC}}}^\bullet$ features an extra switch node to distinguish between unit constructors A and B.

Example 17 (t_{ABC})**.** Let $\mathit{branch} : \{A, B, C\} \to \mathit{Trees}$. The generation of $\textsc{InspectAndSplit}_{t_{\mathrm{ABC}}}^\bullet(\mathit{branch})$ begins with its toplevel decision tree $\mathcal{T}_{\{A,B,C\}}$.

The only minimal relevant discriminant for $\{A, B, C\}$ is $D_{AB,C}$, which inspects the pointer flag to distinguish between unit and non-unit constructors. We emit the following switch node, in which recursively defined trees are present:

$$\mathcal{T}_{\{A,B,C\}} = \mathsf{switch}\,(\Delta\ \&\ 1)\begin{Bmatrix} 1 \mapsto \mathcal{T}_{\{A,B\}} \\ 0 \mapsto \mathcal{T}_{\{C\}} \\ \top \mapsto \mathcal{T}_{\emptyset} \quad \text{(unreachable)} \end{Bmatrix}$$

We now compute $\mathcal{T}_{\{A,B\}}$, whose minimal relevant discriminant is $D_{A,B}$, and $\mathcal{T}_{\{C\}}$:

$$\mathcal{T}_{\{A,B\}} = \mathsf{switch}\,(\Delta)\begin{Bmatrix} 1 \mapsto \mathcal{T}_{\{A\}} \\ 3 \mapsto \mathcal{T}_{\{B\}} \\ \top \mapsto F(\emptyset) \quad \text{(unreachable)} \end{Bmatrix} \qquad \mathcal{T}_{\{C\}} = \mathit{branch}(C)$$

Finally, we compute the last subtrees $\mathcal{T}_{\{A\}} = \mathit{branch}(A)$ and $\mathcal{T}_{\{B\}} = \mathit{branch}(B)$. Combining all previously defined subtrees yields the full decision tree: $\textsc{InspectAndSplit}^{\bullet}_{t_{\mathrm{ABC}}}(\mathit{branch}) =$

$$\mathsf{switch}\,(\Delta\ \&\ 1)\begin{Bmatrix} 1 \mapsto \mathsf{switch}\,(\Delta)\begin{Bmatrix} 1 \mapsto \mathit{branch}(A) \\ 3 \mapsto \mathit{branch}(B) \end{Bmatrix} \\ 0 \mapsto \mathit{branch}(C) \end{Bmatrix}$$

♦

Example 18 (t)**.** Let $\mathit{branch} : \{\mathit{None}, \mathit{Some}\} \to \mathit{Trees}$. The generation of $\textsc{InspectAndSplit}^{\bullet}_{t}(\mathit{branch})$ begins with its toplevel decision tree $\mathcal{T}_{\{\mathit{None},\mathit{Some}\}}$. Its only minimal relevant discriminant is $D_{\mathit{None},\mathit{Some}}$, which inspects the pointer flag to distinguish between *None* and *Some*. This step is sufficient to fully distinguish all constructors: we have $\mathcal{T}_{\mathit{None}} = \mathit{branch}(\mathit{None})$, $\mathcal{T}_{\mathit{Some}} = \mathit{branch}(\mathit{Some})$ and obtain the following decision tree: $\textsc{InspectAndSplit}^{\bullet}_{t}(\mathit{branch}) =$

$$\mathsf{switch}\,(\Delta\ \&\ 1)\begin{Bmatrix} 1 \mapsto \mathit{branch}(\mathit{None}) \\ 0 \mapsto \mathit{branch}(\mathit{Some}) \end{Bmatrix}$$

♦

D.7 Pattern matching compilation

Recall the pattern matching problem from our running example:

$$m_0 = \mathrm{Match}\begin{Bmatrix} \mid (p_0)\ \mathit{None} \mid \mathit{Some}(A) \\ \mid (p_1)\ \mathit{Some}(B) \\ \mid (p_2)\ \mathit{Some}(C(n)) \end{Bmatrix}$$

We can now apply our templated compilation procedure $\textsc{Compile}_t$ to m_0 by instantiating $\textsc{InspectAndSplit}^{\bullet}$ and $\textsc{Unwrap}^{\bullet}$ calls.

Example 19 (OCaml pattern matching compilation of our running example)**.** Our compilation procedure generates the following parametrised decision tree for the input m_0:

$$\textsc{Compile}_t(m_0) = \textsc{InspectAndSplit}^{\bullet}_{t}\left(\begin{array}{l} \mathit{None} \mapsto \mathsf{success}(0) \\ \mathit{Some} \mapsto \textsc{InspectAndSplit}^{\bullet}_{t_{\mathrm{ABC}}}\left(\begin{array}{l} A \mapsto \mathsf{success}(0) \\ B \mapsto \mathsf{success}(1) \\ C \mapsto \mathsf{success}(2, \{n \mapsto \textsc{Unwrap}^{\bullet}_{t_{\mathrm{ABC}}}(C(\square))\}) \end{array}\right) \\ \quad [\Delta/\textsc{Unwrap}^{\bullet}_{t}(\mathit{Some}(\square))] \end{array}\right)$$

We first determine whether we are dealing with *None* or a *Some* value; in the latter case, we determine whether the value "under *Some*" is a unit constructor or C, in which case we bind the representation of

the integer value "under C" to the symbol n. We can now replace the symbolic calls to pattern matching primitives with their expressions to obtain the final decision tree:

$$\textsc{Compile}_t(m_0) = \mathsf{switch}\,(\Delta\ \&\ 1)\ \left\{\begin{array}{l} 1 \mapsto \mathsf{success}(0) \\ 0 \mapsto \mathsf{switch}\,(((*\Delta).1)\ \&\ 1)\ \left\{\begin{array}{l} 1 \mapsto \mathsf{switch}\,((*\Delta).1)\ \left\{\begin{array}{l} 1 \mapsto \mathsf{success}(0) \\ 3 \mapsto \mathsf{success}(1) \end{array}\right\} \\ 0 \mapsto \mathsf{success}(2, \{n \mapsto ((*\Delta).1).1\}) \end{array}\right\} \end{array}\right\}$$

Notice how $\textsc{Unwrap}^{\bullet}_t(\mathit{Some}(\square))$ calls are composed into the $\textsc{InspectAndSplit}^{\bullet}_{t_{\mathrm{ABC}}}$ decision tree so as to inspect the representation of the *subvalue* "under *Some*", which is $(*\Delta).1$. ♦

Patterns

$$
\begin{array}{rll}
p ::= & \_ & \text{(Wildcard)} \\
\mid & x & \text{(Variable)} \\
\mid & (p \mid p') & \text{(Disjunction)} \\
\mid & z \in \mathbb{Z} & \text{(Integer constant pattern)} \\
\mid & \&p & \text{(Reference pattern)} \\
\mid & \langle p_0, \ldots, p_{n-1} \rangle & \text{(Product pattern)} \\
\mid & K(p_0, \ldots, p_{n-1}) & \text{(Constructor pattern)}
\end{array}
$$

Matching

$$
m ::= \text{Match}\,\{p_0 \mid \cdots \mid p_{n-1}\} \qquad \text{(Matching)}
$$

Types

$$
\begin{array}{rll}
t ::= & T \text{ (e.g., i32, u16)} & \text{(Primitive type)} \\
\mid & \&t & \text{(Reference type)} \\
\mid & \prod_n t_i & \text{(Product type)} \\
\mid & \sum_n K_i(t_{i,0}, \ldots, t_{i,n_i-1}) & \text{(Sum type)} \\
\Gamma ::= & \left\{ \begin{array}{l} x_0 : t_0; \ldots; x_{n-1} : t_{n-1}; \\ t_0 \mapsto \tau_0; \ldots; t_{n'-1} \mapsto \tau_{n'-1} \end{array} \right\} & \text{(Type environment)}
\end{array}
$$

Values

$$
\begin{array}{rll}
v ::= & z \in \mathbb{Z} & \text{(Integer constant)} \\
\mid & \&v & \text{(Reference)} \\
\mid & \langle v_0, \ldots, v_{n-1} \rangle & \text{(Product)} \\
\mid & K(v_0, \ldots, v_{n-1}) & \text{(Constructor)} \\
\sigma ::= & \{x_0 \mapsto v_0; \ldots; x_{n-1} \mapsto v_{n-1}\} & \text{(Value env.)}
\end{array}
$$

Figure 11: Pat, our simplified language of patterns and types

$$
\textsc{Any}\ \ \_ \rhd v \to \emptyset
\qquad
\textsc{Var}\ \ x \rhd v \to \{x \mapsto v\}
\qquad
\textsc{Constant}\ \ z \rhd z \to \emptyset
\qquad
\textsc{Reference}\ \ \frac{p \rhd v \to \sigma}{\&p \rhd \&v \to \sigma}
$$

$$
\textsc{Tuple}\ \ \frac{\forall i, p_i \rhd v_i \to \sigma_i \qquad \forall j \neq i, \sigma_i \cap \sigma_j = \emptyset}{\langle \overline{p_i} \rangle \rhd \langle \overline{v_i} \rangle \to \bigcup \sigma_i}
\qquad
\textsc{Constructor}\ \ \frac{\forall i, p_i \rhd v_i, \sigma_i \qquad \forall j \neq i, \sigma_i \cap \sigma_j = \emptyset}{K(\overline{p_i}) \rhd K(\overline{v_i}) \to \bigcup \sigma_i}
\qquad
\textsc{AltL}\ \ \frac{p_1 \rhd v \to \sigma}{(p_1 \mid p_2) \rhd v \to \sigma}
$$

$$
\textsc{AltR}\ \ \frac{p_1 \not\rhd v \qquad p_2 \rhd v \to \sigma}{(p_1 \mid p_2) \rhd v \to \sigma}
\qquad
\textsc{Matching}\ \ \frac{p_i \rhd v \to \sigma \qquad \forall j < i, p_j \not\rhd v}{\text{Match}\,\{p_1 \mid \cdots \mid p_n\} \rhd v \to i, \sigma}
$$

Figure 12: Semantics of patterns: $p \rhd v \to \sigma$

$$\textsc{Any}\quad \Gamma \vdash \_ : t \to \emptyset \qquad \textsc{Var}\quad \Gamma \vdash x : t \to \{x : t\} \qquad \textsc{Reference}\quad \frac{\Gamma \vdash p : t \to \Gamma'}{\Gamma \vdash \&p : \&t \to \Gamma'} \qquad \textsc{Constant}\quad \frac{z \in T}{\Gamma \vdash z : T \to \emptyset}$$

$$\textsc{Tuple}\quad \frac{\forall i; \Gamma \vdash p_i : t_i \to \Gamma_i \qquad \forall i,j; \Gamma_i \cap \Gamma_j = \emptyset}{\Gamma \vdash \langle \overline{p_i} \rangle : \prod t_i \to \bigcup \Gamma_i} \qquad \textsc{Constructor}\quad \frac{\exists i_0, K = K_{i_0} \qquad \forall j; \Gamma \vdash p_j : t_{i,j} \to \Gamma_j \qquad \forall i,j; \Gamma_i \cap \Gamma_j = \emptyset}{\Gamma \vdash K(\overline{p_j}) : \sum K_i(\overline{t_{i,j}}) \to \bigcup \Gamma_j}$$

$$\textsc{Alt}\quad \frac{\forall i;\ \Gamma \vdash p_i : t \to \Gamma'}{\Gamma \vdash p_1 \mid p_2 : t \to \Gamma'} \qquad \textsc{Matching}\quad \frac{\forall i;\ \Gamma \vdash p_i : t \to \Gamma'}{\Gamma \vdash \mathrm{Match}\,\{p_1 \mid \cdots \mid p_n\} : t \to \Gamma'}$$

Figure 13: Typing of patterns: $\vdash p : t \to \Gamma$

$$\phi ::= \top \mid (\mathrm{ops} = x_h) \mid (\mathrm{ops} = z) \mid \phi \wedge \phi$$
$$\widehat{p} ::= t@h \mid \mathrm{Word}_\ell\,(\phi) \mid \mathrm{Ptr}_{\ell,a}(\phi) \twoheadrightarrow \widehat{p} \mid \left\{\!\left\{\overline{\widehat{p}?}\right\}\!\right\} \mid \bigsqcup \widehat{p}$$

Figure 14: Memory patterns values and types

$$(t@h) \cap (t@h) = t@h \qquad \frac{\textsc{PreRepr}_t^\bullet(\top) \cap \widehat{p} = \textsc{PreRepr}_t^\bullet(\top)}{(t@h) \cap \widehat{p} = t@h} \qquad \frac{\ell \leq \ell'}{\mathrm{Word}_\ell\,(\phi) \cap \mathrm{Word}_{\ell'}\,(\phi') = \mathrm{Word}_{\ell'}\,(\phi \wedge \phi')}$$

$$\frac{\ell' \leq a}{\mathrm{Ptr}_{\ell,a}(\phi) \twoheadrightarrow \widehat{p} \cap \mathrm{Word}_{\ell'}\,(\phi') = \mathrm{Ptr}_{\ell,a}(\phi \wedge \phi') \twoheadrightarrow \widehat{p}} \qquad \frac{\widehat{p} \cap \widehat{p}' = \widehat{p}''}{\mathrm{Ptr}_{\ell,a}(\phi) \twoheadrightarrow \widehat{p} \cap \mathrm{Ptr}_{\ell,a}(\phi') \twoheadrightarrow \widehat{p}' = \mathrm{Ptr}_{\ell,a}(\phi \wedge \phi') \twoheadrightarrow \widehat{p}''}$$

$$\frac{n \leq n' \qquad \forall i < n, \widehat{p}_i \cap \widehat{p}'_i = \widehat{p}''_i}{\left\{\!\left\{\overline{\widehat{p}_i}^{\,n}, \ldots\right\}\!\right\} \cap \left\{\!\left\{\overline{\widehat{p}'_i}^{\,n'}, \ldots\right\}\!\right\} = \left\{\!\left\{\overline{\widehat{p}''_i}^{\,n}, \overline{\widehat{p}'_{n+i}}^{\,n'-n}, \ldots\right\}\!\right\}} \qquad \frac{\forall i, \widehat{p} \cap \widehat{p}_i = \widehat{p}'_i}{\widehat{p} \cap \bigsqcup_n \widehat{p}_i = \bigsqcup_n \widehat{p}'_i}$$

Figure 15: Intersection

$$(t@h) \cup (t@h) = t@h \qquad \frac{\textsc{PreRepr}_t^\bullet(\top) \cup \widehat{p} = \textsc{PreRepr}_t^\bullet(\top)}{(t@h) \cup \widehat{p} = t@h} \qquad \mathrm{Word}_\ell(\phi) \cup \mathrm{Word}_\ell(\phi) = \mathrm{Word}_\ell(\phi)$$

$$\frac{\widehat{p} \cup \widehat{p}' = \widehat{p}''}{\mathrm{Ptr}_{\ell,a}\widehat{p}(\phi) \cup \mathrm{Ptr}_{\ell,a}\widehat{p}(\phi) = \mathrm{Ptr}_{\ell,a}\widehat{p}''(\phi)}$$

$$\frac{\widehat{p}_{i_0} \cup \widehat{p}'_{i_0} = \widehat{p}''_{i_0}}{\left\{\!\left\{\overline{\widehat{p}_i}^{\,n}, \ldots\right\}\!\right\} \cup \left\{\!\left\{\overline{\widehat{p}_i}^{\,i_0}, \widehat{p}'_{i_0}, \overline{\widehat{p}_i}^{\,n-i_0-1}, \ldots\right\}\!\right\} = \left\{\!\left\{\overline{\widehat{p}_i}^{\,i_0}, \widehat{p}''_{i_0}, \overline{\widehat{p}_i}^{\,n-i_0-1}, \ldots\right\}\!\right\}} \qquad \frac{\forall i, \widehat{p} \cup \widehat{p}_i = \widehat{p}'_i}{\widehat{p} \cup \bigsqcup_n \widehat{p}_i = \bigsqcup_n \widehat{p}'_i}$$

$$\frac{\text{no other rule applies}}{\widehat{p} \cup \widehat{p}' = \widehat{p} \sqcup \widehat{p}'}$$

Figure 16: Union

$$\llcorner\blacksquare\lrcorner_*[\widehat{p}] = \widehat{p} \qquad \frac{\mathrm{ops}[\phi] = \phi'}{\llcorner\underline{\mathrm{ops}}\lrcorner_*[\mathrm{Word}_\ell\,(\phi)] = \mathrm{Word}_\ell\,(\phi') \qquad \llcorner\underline{\mathrm{ops}}\lrcorner_\ell[\mathrm{Word}_\ell\,(\phi)] = \mathrm{Word}_\ell\,(\phi')}$$

$$\frac{\mathrm{ops}[\phi] = \phi'}{\llcorner\underline{\mathrm{ops}}\lrcorner_*[\mathrm{Ptr}_{\ell,a}(\phi) \twoheadrightarrow \widehat{p}] = \mathrm{Ptr}_{\ell,a}(\phi') \twoheadrightarrow \widehat{p} \qquad \llcorner\underline{\mathrm{ops}}\lrcorner_a[\mathrm{Ptr}_{\ell,a}(\phi) \twoheadrightarrow \widehat{p}] = \mathrm{Ptr}_{\ell,a}(\phi') \twoheadrightarrow \widehat{p}}$$

$$\frac{\widehat{h}[\widehat{p}] = \widehat{p}'}{\left(\mathrm{Ptr}_{\ell,a}\widehat{h}\right)[\widehat{p}] = \mathrm{Ptr}_{\ell,a}(\top) \twoheadrightarrow \widehat{p}'} \qquad \frac{\widehat{h}[\widehat{p}] = \widehat{p}'}{\left\{\!\left\{\_, \ldots, \underset{i}{\widehat{h}}, \ldots, \_\right\}\!\right\}[\widehat{p}] = \left\{\!\left\{\_, \ldots, \underset{i}{\widehat{p}'}, \ldots, \_\right\}\!\right\}}$$

Figure 17: Memory context instantiation – $\mathrm{ops}[\phi]$ substitutes each clause $\mathrm{ops}' = x$ in ϕ with $\mathrm{ops} \circ \mathrm{ops}' = x$

$$\llcorner\blacksquare\lrcorner_* \prec \llcorner\mathbf{op}(\mathrm{ops})\lrcorner_* \qquad \llcorner\blacksquare\lrcorner_\ell \prec \llcorner\mathbf{op}(\mathrm{ops})\lrcorner_\ell \qquad \frac{\llcorner\underline{\mathrm{ops}}\lrcorner_* \prec \llcorner\underline{\mathrm{ops}'}\lrcorner_*}{\llcorner\mathbf{op}(\mathrm{ops})\lrcorner_* \prec \llcorner\mathbf{op}(\mathrm{ops}')\lrcorner_*} \qquad \frac{\llcorner\underline{\mathrm{ops}}\lrcorner_\ell \prec \llcorner\underline{\mathrm{ops}'}\lrcorner_\ell}{\llcorner\mathbf{op}(\mathrm{ops})\lrcorner_\ell \prec \llcorner\mathbf{op}(\mathrm{ops}')\lrcorner_\ell}$$

$$\llcorner\underline{\mathrm{ops}}\lrcorner_* \prec \mathrm{Ptr}_{\ell,a}\widehat{h} \qquad \llcorner\blacksquare\lrcorner_\ell \prec \mathrm{Ptr}_{\ell,a}\widehat{h} \qquad \llcorner\underline{\mathrm{ops}}\lrcorner_a \prec \mathrm{Ptr}_{\ell,a}\widehat{h} \qquad \llcorner\blacksquare\lrcorner_* \prec \left\{\!\left\{\_, \ldots, \underset{i}{\widehat{h}}, \ldots, \_\right\}\!\right\}$$

$$\frac{\widehat{h} \prec \widehat{h}'}{\mathrm{Ptr}_{\ell,a}\widehat{h} \prec \mathrm{Ptr}_{\ell,a}\widehat{h}' \qquad \left\{\!\left\{\_, \ldots, \underset{i}{\widehat{h}}, \ldots, \_\right\}\!\right\} \prec \left\{\!\left\{\_, \ldots, \underset{i}{\widehat{h}'}, \ldots, \_\right\}\!\right\}}$$

Figure 18: Partial prefix order

$$\frac{\mathbf{op} \neq \mathbf{op}'}{\llcorner\mathbf{op}(\mathrm{ops})\lrcorner_* \bowtie \llcorner\mathbf{op}'(\mathrm{ops}')\lrcorner_* \qquad \llcorner\mathbf{op}(\mathrm{ops})\lrcorner_\ell \bowtie \llcorner\mathbf{op}'(\mathrm{ops}')\lrcorner_\ell} \qquad \frac{\llcorner\underline{\mathrm{ops}}\lrcorner_* \bowtie \llcorner\underline{\mathrm{ops}'}\lrcorner_*}{\llcorner\mathbf{op}(\mathrm{ops})\lrcorner_* \bowtie \llcorner\mathbf{op}(\mathrm{ops}')\lrcorner_*}$$

$$\frac{\llcorner\underline{\mathrm{ops}}\lrcorner_\ell \bowtie \llcorner\underline{\mathrm{ops}'}\lrcorner_\ell}{\llcorner\mathbf{op}(\mathrm{ops})\lrcorner_\ell \bowtie \llcorner\mathbf{op}(\mathrm{ops}')\lrcorner_\ell} \qquad \frac{\llcorner\underline{\mathrm{ops}}\lrcorner_* \not\prec \mathrm{Ptr}_{\ell,a}\widehat{h}}{\llcorner\underline{\mathrm{ops}}\lrcorner_* \bowtie \mathrm{Ptr}_{\ell,a}\widehat{h}} \qquad \frac{\llcorner\underline{\mathrm{ops}}\lrcorner_a \not\prec \mathrm{Ptr}_{\ell,a}\widehat{h}}{\llcorner\underline{\mathrm{ops}}\lrcorner_a \bowtie \mathrm{Ptr}_{\ell,a}\widehat{h}}$$

$$\frac{i \neq i'}{\left\{\!\left\{\_, \ldots, \underset{i}{\widehat{h}}, \ldots, \_\right\}\!\right\} \bowtie \left\{\!\left\{\_, \ldots, \underset{i'}{\widehat{h}'}, \ldots, \_\right\}\!\right\}} \qquad \frac{\widehat{h} \bowtie \widehat{h}'}{\mathrm{Ptr}_{\ell,a}\widehat{h} \bowtie \mathrm{Ptr}_{\ell,a}\widehat{h}' \qquad \left\{\!\left\{\_, \ldots, \underset{i}{\widehat{h}}, \ldots, \_\right\}\!\right\} \bowtie \left\{\!\left\{\_, \ldots, \underset{i'}{\widehat{h}'}, \ldots, \_\right\}\!\right\}}$$

Figure 19: Non-comparable compatibility

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399