Knit&Frog: Pattern matching compilation for custom memory representations

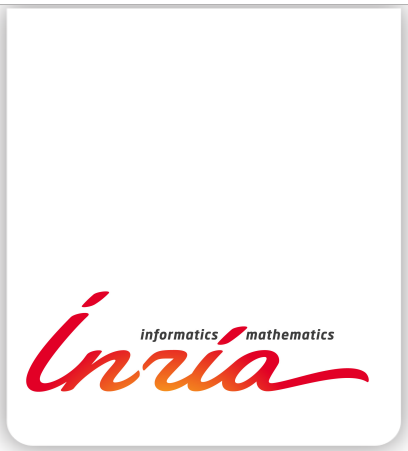
Thaïs Baudon, Laure Gonnord, Gabriel Radanne

Knit&Frog: Pattern matching compilation for custom memory representations

Thaïs BAUDON, Laure GONNORD, Gabriel RADANNE

RESEARCH REPORT
N° 9473
June 2022
Project-Team CASH

ISSN 0249-6399 ISRN INRIA/RR--9473--FR+ENG



Knit&Frog: Pattern matching compilation for custom memory representations

Thaïs BAUDON*, Laure GONNORD†, Gabriel RADANNE‡

Project-Team CASH

Research Report n° 9473 — June 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 *internal memory representation* which must accommodate garbage-collection and polymorphism.
ADTs now appear in languages more liberal in their memory representation such as Rust. Notably, Rust is now introducing more and more optimizations of the memory layout of Algebraic Data Types. As memory representation and compilation are interdependent, it raises the question of pattern matching compilation in the presence of non-regular, potentially customized, memory layouts.
In this report, we present Knit&Frog, a framework to compile pattern-matching for monomorphic ADTs, *parametrized* by an arbitrary memory representation. We propose a novel way to describe choices of memory representation along with a validity condition under which we prove the correctness of our compilation scheme. The approach is implemented in a prototype tool `ribbit`.

Key-words: Algebraic Data Types, 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

Knit&Frog: Pattern matching compilation for custom memory representations

Résumé : Ce rapport de recherche présente Knit&Frog, un cadre formel pour décrire des représentations mémoires de types algébriques, ainsi que son usage dans la formalisation d'un algorithme de filtrage (*pattern-matching*) paramétré par la représentation mémoire. La correction de l'algorithme est prouvé sous une hypothèse de *validité* de la représentation. L'algorithme est implémenté dans l'outil `ribbit`.

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

1 Introduction

Algebraic Data Types (ADTs) are an essential tool to model data and information. They allow to group together information in a consistent way through the use of records, also called product types, and to organize options through the use of variants, also called sum types. Product and sum types together allow to enforce invariants present in the domain under study and provide convenient tools to inspect these data-types through pattern matching.

Combined, these features offers numerous advantages:

- 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.
- Optimize 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 to language designers who want 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 optimizations 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 optimize the representation of terms. The simplest example is the `Option` type, which is either `Some value` or `None`. If the value in question is an integer from 0 to 10, `None` can easily be represented as 11. This optimization is regularly done by programmers manually, forgoing the guarantees provided by languages. More complex optimizations on nested and rich data-types are even more error-prone. These transformations could easily be done automatically by the compiler. This trove of optimization potential has been brushed on in recent versions of Rust, but remains largely unexplored.

Finally, on top of the previously mentioned difficulty of adapting pattern matching compilation schemes to irregular memory representations, the correctness of such compilation schemes is also delicate to establish when the terms can be arbitrarily shaped.

In this article, we lay the groundwork for highly optimized pattern matching in any language with arbitrary non-uniform memory representation. We present Knit&Frog, a compilation framework for rich pattern languages with arbitrary memory representation of monomorphic terms:

- We provide a characterization of memory representations for terms of Algebraic Data Types as two main operations: **Knit** which builds a term into its memory representation, and **Frog** which deconstructs the memory representation to identify the underlying term.
- We provide a compilation scheme parameterized by the memory representation. This scheme is adapted from Maranget (2008), the best implementation of pattern matching known thus far.
- We provide a sufficient condition for a memory representation to yield itself to pattern matching and prove end-to-end correctness of our compilation scheme in this case.

We start by giving some examples of memory representations and pattern-matching compilation as motivation in Section 2. We then provide a formal presentation of our pattern language (Section 3) and our target memory representations (Section 4). Finally, we present our compilation scheme (Section 5) along with end-to-end examples (Section 6) and a proof of correctness (Section 7).

```
enum Option<T> {
  None, // No value
  Some (T), // Some value
}
type Interval = Option<&(u64, u64)>
```

```
fn length(x: Interval) -> u64 {
    match x {
        None => 0,
        Some(&(i1, j1)) => j1 - i1
    }}
```

Figure 1: Example of program manipulating the `Option` type in Rust

2 Algebraic Data Types and their memory representation

Let us now explore the different memory representation choices for Algebraic Data Types (ADTs, for short), and how their values are manipulated. In this section, we present various examples of types and programs and discuss how they are currently represented in existing languages, and how they might be. For illustration, we will look at the output of two languages which implement 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 favors performance and absolute control over low-level details. These differences result in drastically different choices in memory representation. We thus start our exploration by a classic question: how to represent the option type?

Example 1 (Option type)**.** We consider the program in Fig. 1 in Rust syntax. We first restate the `Option<T>` type, whose goal is to model that some data of type `T` is *optionally* present. Either we have `Some` data, or `None`. This is modeled by the two constructors `None` and `Some`. The `None` constructor is argument-less, while the `Some` constructor has a parameter of type `T`.

The `Interval` type is an optional pair of integers. `u64` denotes an unsigned integer over 64 bits and `&(u64, u64)` denotes a *reference*[1] to a pair of such integers. Finally, `Option<&(u64, u64)>` denotes an optional pair. The value `Some(&(u1756,u1791))` is of type `Interval` and designates an interval from 1756 to 1791.

The `length` function takes an interval and returns its length 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, `None` yields a length of 0 and `Some(i,j)` yields a length of `j-i`.

We now look at how Rust and OCaml compile the `length` 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 representation. It uses the `if` primitive to discriminate between cases. Rust's MIR[2] uses slightly lower level operations such as dereferencing and field accesses, and a `switch` construct on memory words.

In both cases, `None` is represented as an unboxed integer. 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. This block contains a single field with a pointer to the inner tuple. This tuple is a block with a tag and two fields, containing the integers. This totals 6 memory words[3]. In Rust, `Some` is a pointer referencing directly a memory span of 2 memory words. Rust recognizes `Option<&T>` as something that can be represented compactly, by remarking that Rust pointers are non-null. It can thus use 0 as the `None` value, and directly use the pointer in the `Some` cases. The resulting type uses 3 memory words.

Our goal here is not to oppose the two representations: the OCaml representation is designed to support metadata for the GC and works smoothly with polymorphism and separate compilation, while Rust's representation focuses on efficiency and control. Both are excellent designs for their objectives. However, in both cases, we want efficient implementations of pattern matching to deconstruct values in memory. In fact, the choice of memory representation can have a great influence on the shape of the code generated to match values (in terms of number of switches, complexity of expressions...). The goal of this article is precisely to

[1] For simplicity of presentation, we omit details regarding lifetimes and differences between `&` and `Box`.

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

[3] Knowledgeable readers might remark that it is possibly to give up the common `Option` type and inline the definition of the tuple inside the sum, going down to 1 indirection and 4 memory words at the price of modularity.

```
fn length(_1:Interval) =
  switch(_1){
  | 0 -> 0
  | 1 -> let _5 = (*((_1 as Some).0)).0;
         let _6 = (*((_1 as Some).0)).1;
         _6 - _5
}
```

(a) Simplified Rust "MIR" output

```
length(param/85) =
  (if (== param/85 1)
    0
    (let (*match*/88 (field 1 param/85))
      (- (field 2 *match*/88) (field 1 *match*/88))
    ))
}
```

(b) Simplified OCaml "Lambda" output

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

```
//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,
}
```

```
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),
}
```

Figure 3: Example of Red-Black tree and their balancing operation in Rust

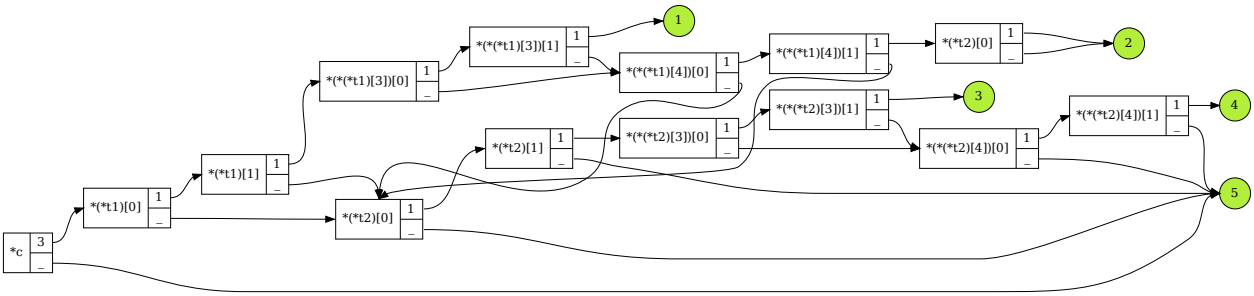


Figure 4: Output of our `ribbit` compiler for the code of Fig. 3 in the OCaml memory representation

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).

provide efficient compilation of complex patterns for any memory representation. As patterns become more complex, the compiler has to do more work, as we show in Example 2.

Example 2 (Search Trees)**.** To illustrate the need for a sophisticated pattern matching compiler, we now look at *Red-Black Trees*. A Rust version is depicted in Fig. 3. This type definition expresses trees as recursive data structures, and uses a tuple for the various fields in the `Node` case. For instance, `Node(Red, 1515, &Node(Black, &Empty,&Empty), &Empty)` is a tree of type `RBT<Int>`.

Red-Black trees famously rely on a fairly complex balancing step, which redistributes the colors depending on the internal invariant of the data structure. Thanks to nested pattern matching, this step can be expressed very compactly, as shown in Fig. 3. This pattern matching inspects four arguments at the same time: the current color `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 optimized decision trees Kosarev et al. (2020); Maranget (2008); Sestoft (1996). The resulting code is highly non-trivial, as can be seen in Fig. 4.

Such optimized decision trees should still respect, and even take advantage of, the memory representation. For instance, the type `RBT<u64>` is represented by the Rust compiler using only 4 memory words per `Node`. A node is then made of a word with a bit marking it as non-0 (to avoid confusion with `None`) and the color bit, followed by the 64-bit integer and the two pointers. It is however possible to be even more compact: the Linux kernel uses a hand-crafted representation of Red-Black trees that exploits bit-stealing to use one less word Wilke (2016): since pointers are word-aligned, colors can be stored in the lower bits of each pointer. These complex manually-tuned optimizations are common in low-level C programming, but come at a high cost: programmers forgo the use of modern safer constructs such as Algebraic Data Types and pattern

matching. Furthermore, they now need to design the decision tree themselves, and make sure their choice of representation contains enough information to distinguish, for instance, between the `Empty` and `Node` cases. Moreover, the emission of code for such mangled objects in memory is delicate (notably, the bit-stealing technique in the Linux kernel is undefined behavior in the C standard).

Ideally, we would like to still use high-level pattern languages, while the compilation process adapts and exploits the optimized representation. This article makes a first step towards this goal by presenting Knit&Frog, a pattern matching compilation technique **parameterized by the memory representation**. We formalize what it means to define a memory representation, present a validity criterion ensuring its suitability for pattern matching and prove the correctness of our compilation scheme. We also present several examples of both toy and realistic memory representations. Finally, we implemented our technique in a tool called `ribbit`, whose output using the OCaml representation is shown in Fig. 4.

3 Pattern language

We now present a formal version of our source language. For clarity and ease of presentation, we make some simplifying assumptions. First off, we only consider the pattern part of the language, which represents the core of our compilation procedure. Furthermore, we assume the following:

- Types are monomorphic. All code has been specialized previously, and all type variables have been replaced by concrete types.
- Patterns are always *exhaustive*: all possible cases are handled.
- Constant types fit in a machine word, such as machine integers, floats, chars, ... Strings are thus not a constant type (and could, for instance, be represented as arrays of chars).
- Types are non-recursive. This limitation is far less restrictive than it seems for compilation purposes since recursive types can be unfolded to the size of the given pattern.

We shall come back to these limitations later in the article.

Our language, described in Fig. 5, 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 constant, product, constructor and reference patterns as introduced previously, along with variables (x), wildcards (_) and "or"-patterns ($p_1 \mid p_2$). Types, denoted τ, are composed of constant types, product types ($\prod_k \tau_k$, also written $(\tau_1, \tau_2, \dots)$), sum types ($\sum_k K_k(\tau_{k,1}, \dots, \tau_{k,n_k})$ also written $K_1(\tau_{1,1}, \dots, \tau_{1,n_1}) + K_2(\tau_{2,1}, \dots, \tau_{2,n_2}) + \cdots$) and reference types ($\&\tau$). Typing environments, denoted Γ, associate variables to their types. Finally, values, denoted v, follow a subset of the pattern grammar. Value environments, denoted σ, associate variables to their values.

Example 3 (`Interval` example, cont')**.** The `Interval` type of Example 1 is denoted by

$$Interval = None() + Some(\&(u64, u64)),$$

The first constructor `None` has arity 0 and the second one `Some` has arity 1 with a reference type to a tuple (product of size 2). The matching problem for function `length` will be expressed as:

$$\text{Match} \left\{ \begin{array}{l} \mid None \\ \mid Some(\&(x_1, x_2)) \end{array} \right\}$$

3.1 Semantics

The semantics of patterns is presented in Fig. 6, which defines the judgment $p \rhd v \to \sigma$ meaning that pattern p matches value v and binds environment σ. The object being matched (here, v) is also called the *discriminant*. We also assume the existence of the negated judgment, $p \not\rhd v$ where pattern p doesn't match value v. Most of the rules are straightforward, with the following specificities:

- The bound value environment returned by the matching is populated by variables, through the VAR rule.

Patterns

$$
\begin{array}{lr}
p ::= c \in \mathcal{C} & \text{(Constant pattern)} \\
\mid \langle p_1, \ldots, p_n \rangle & \text{(Product pattern)} \\
\mid K(p_1, \ldots, p_n) & \text{(Constructor pattern)} \\
\mid \&p & \text{(Reference pattern)} \\
\mid \_ & \text{(Wildcard)} \\
\mid x & \text{(Variable)} \\
\mid (p_1 \mid p_2) & \text{(Disjunction)}
\end{array}
$$

Matching

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

Types

$$
\begin{array}{lr}
\tau ::= T & \text{(Built-in Type)} \\
\mid \prod_n \tau_i & \text{(Product Type)} \\
\mid \sum_n K_i(\tau_1, \ldots, \tau_{\ell_i}) & \text{(Sum Type)} \\
\mid \&\tau & \text{(Reference Type)} \\
\Gamma ::= \{x_1 : \tau_1; \ldots; x_n : \tau_n\} & \text{(Type Environment)}
\end{array}
$$

Values

$$
\begin{array}{lr}
v ::= c \in \mathcal{C} & \text{(Constant)} \\
\mid \langle v_1, \ldots, v_n \rangle & \text{(Product)} \\
\mid K(v_1, \ldots, v_n) & \text{(Constructor)} \\
\mid \&v & \text{(Reference)} \\
\sigma ::= \{x_1 \mapsto v_1; \ldots; x_n \mapsto v_n\} & \text{(Value Env.)}
\end{array}
$$

Figure 5: Our language of pattern and their types

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

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

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

Figure 6: Semantic of patterns – $p \rhd v \to \sigma$

- We enforce that there is no shadowing: variables must be bound only once, as asserted by the side-condition in the TUPLE and 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 judgment in rule MATCHING: $\text{Match}\,\{p_1 \mid \cdots \mid p_n\} \rhd v \to i, \sigma$ which behaves as the normal judgment, 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.

Example 4 (`Interval` example, cont')**.** For the following matching problem in which we provide pattern names:

$$
\text{Match} \left\{ \begin{array}{l} \mid (p_0)\ \mathit{None} \\ \mid (p_1)\ \mathit{Some}(\&(x_1, x_2)) \end{array} \right\}
$$

- The *pattern* p_0 will match the value `None`, which will be denoted by $p_0 \rhd \mathit{None} \to 0, \emptyset$;
- The second pattern, p_1, will match any `Interval` inhabitant of the form $\mathit{Some}(\&(v, v'))$, with v, v' two integers. For instance $p_1 \rhd \mathit{Some}(\&(1756, 1791)) \to 1, \{x_1 \mapsto 1756; x_2 \mapsto 1791\}$.

3.2 Typing

Fig. 7 defines the typing judgment $\vdash p : \tau \to \Gamma$ which means that pattern p has type τ and binds variables present in environment Γ. Typing follows the semantics closely:

$$\textsc{Any}\quad \vdash \_ : \tau \to \emptyset \qquad \textsc{Var}\quad \vdash x : \tau \to \{x : \tau\} \qquad \textsc{Constant}\ \frac{c \in T}{\vdash c : T \to \emptyset} \qquad \textsc{Reference}\ \frac{\vdash p : \tau \to \Gamma}{\vdash \&p : \&\tau \to \Gamma}$$

$$\textsc{Tuple}\ \frac{\forall i;\ \vdash p_i : \tau_i \to \Gamma_i \qquad \forall i, j; \Gamma_i \cap \Gamma_j = \emptyset}{\vdash \langle \overline{p_i} \rangle : \prod \tau_i \to \bigcup \Gamma_i} \qquad \textsc{Alt}\ \frac{\forall i;\ \vdash p_i : \tau \to \Gamma}{\vdash p_1 \mid p_2 : \tau \to \Gamma}$$

$$\textsc{Constructor}\ \frac{\exists i_0, K = K_{i_0} \qquad \forall j;\ \vdash p_j : \tau_{i,j} \to \Gamma_j \qquad \forall i, j; \Gamma_i \cap \Gamma_j = \emptyset}{\vdash K(\overline{p_j}) : \sum K_i(\overline{\tau_{i,j}}) \to \bigcup \Gamma_j} \qquad \textsc{Matching}\ \frac{\forall i;\ \vdash p_i : \tau \to \Gamma}{\vdash \mathrm{Match}\,\{p_1 \mid \cdots \mid p_n\} : \tau \to \Gamma}$$

Figure 7: Typing of patterns – $\vdash p : \tau \to \Gamma$

- 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 in the Alt rule.

Example 5 (Pattern types for `Interval` example)**.**

$$\vdash p_1 : \mathit{Interval} \to \emptyset \qquad \vdash p_2 : \mathit{Interval} \to \{x_1 : u64;\ x_2 : u64\}$$

4 Target Language and Memory Representations

We now describe the target environment of our compilation procedure. In the context of this article, part of the target environment is considered universal: what kind of structures can be represented in memory (such as machine words, pointers, ...) and how to manipulate and compute them. Another part of the target is specific and defines a so-called "memory representation": a choice of how language values are represented in memory. We formally state the notion of *valid* memory representations which is crucial for our compilation procedure, along with some examples.

4.1 Memory values

In order to reason about values in memory, we consider an abstraction of memory contents, whose grammar is presented in Fig. 8a. Memory values can be arbitrarily long machine words, denoted by w; n-bit wide pointers to another memory value, denoted by $\&_n r$; or contiguous arrays of memory values with alignment a, denoted by $[r_0, \ldots, r_n]_a$.

Example 6 (Memory value of *Some*(5) in OCaml)**.** In OCaml, *Some*(5) of type `int option` (an optional integer) is represented in memory by a pointer to a block (a contiguous array) containing a tag followed by the integer 5. The tag in this case is 0. The integer 5 is represented as $\overline{1011}^2$ due to the pointer flag Minsky and Madhavapeddy (2021). The resulting representation of *Some*(5) can thus be written as $\&_{64} \left[0, \overline{1011}^2\right]_{64}$.

4.2 Computing on memory values

4.2.1 Decision trees

From a matching problem, we will produce our target under the form of *decision trees*, defined in Fig. 8c. 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 – also called the *switch discriminant* – and of a list of cases $\mathcal{C}$. Leaves of a decision tree can either be $\mathsf{success}(j, \Theta)$, which successfully returns a branch index j and a set of binding

Memory values

$r ::= w$ (Word)
$\mid \&_n r$ (n-bit wide reference)
$\mid [\overline{r_i}]_a$ (a-aligned contiguous array)

(a) Memory values

Tags

$t ::= \top$ (Default tag)
$\mid c$ (Constant tag)
$\mid K$ (Constructor tag)

Contexts

$h ::= \square$ (Hole)
$\mid \langle \_, \ldots, \_, h, \_, \ldots, \_ \rangle$ (Product context)
$\mid K(\_, \ldots, \_, h, \_, \ldots, \_)$ (Constr. context)
$\mid \&h$ (Reference context)

(b) Tags and Contexts

Target expressions

$e ::= \Delta$ (Main Discriminant)
$\mid *e$ (Dereferencing)
$\mid e.\text{offset}$ (Array access)
$\mid \ldots$ (Other operations)

Binding expressions

$\Theta ::= \{\overline{x_i \mapsto e_i}\}$

Switch cases

$\mathcal{C} ::= w \mapsto \mathcal{T} \cdot \mathcal{C}$ (Regular case)
$\mid \top \mapsto \mathcal{T}$ (Default case)
$\mid \emptyset$ (No Default case)

Decision trees

$\mathcal{T} ::= \mathsf{switch}\,(e)\,\{\,\mathcal{C}\,\}$ (Decision node)
$\mid \mathsf{success}(j, \Theta)$ (Branch j with bindings Θ)
$\mid \mathsf{unreachable}$ (Unreachable leaf)

(c) Decision trees

Figure 8: The target languages

$$\textsc{Success}\ \frac{\forall (x \mapsto e) \in \Theta, \quad e[r] \hookrightarrow r_x}{\mathsf{success}(j, \Theta)[r] \hookrightarrow j, \{x \mapsto r_x\}_{x \in s}} \qquad \textsc{Default}\ \frac{\mathcal{T}[r] \hookrightarrow j, \sigma}{\mathsf{switch}\,(e)\,\{\top \mapsto \mathcal{T}\}[r] \hookrightarrow j, \sigma}$$

$$\textsc{CaseT}\ \frac{\mathcal{T}[r] \hookrightarrow j, \sigma \qquad e[r] \hookrightarrow w}{\mathsf{switch}\,(e)\,\{w \mapsto \mathcal{T} \cdot l\}[r] \hookrightarrow j, \sigma}$$

$$\textsc{CaseF}\ \frac{\mathsf{switch}\,(e)\,\{l\}[r] \hookrightarrow j, \sigma \qquad e[r] \hookrightarrow w' \neq w}{\mathsf{switch}\,(e)\,\{w \mapsto \mathcal{T} \cdot l\}[r] \hookrightarrow j, \sigma}$$

Figure 9: Decision tree evaluation

expressions Θ, or unreachable. Θ can be understood as a set of let-binders which are necessary for the later evaluation of the successful branch. At toplevel, decision trees take as input the *main discriminant* of the whole matching, designated by Δ. The discriminant of a (sub)-switch is an expression manipulating memory values mentioning this main discriminant. In addition, expressions are target-dependent and can include dereferencing denoted by $*e$, word accesses denoted by e.offset (where the offset is statically known), arithmetic and logical operations, etc.

A decision tree $\mathcal{T}$ applied to an input memory value r reduces to a result j, σ where j is a branch index and σ is a binding environment. This judgment is written $\mathcal{T}[r] \hookrightarrow j, \sigma$ and is defined in Fig. 9. We assume that a similar judgment reduces expressions to memory values, and is denoted by $e[r] \hookrightarrow r'$. The rule Success evaluates a successful leaf by evaluating each expression contained in the binders Θ. The three other rules Default, CaseT and CaseF evaluate a switch node by finding the first case matching the discriminant: either the first branch whose left-hand side is equal to the discriminant, or the wildcard (if present). There are no rules for evaluating an empty branch or an "unreachable" leaf: as mentioned previously, we consider all source-level language patterns to be exhaustive. We will prove that such cases are indeed unreachable for all emitted trees.

Example 7 (Decision tree for `Interval`)**.** For illustration purposes, we translate the decision tree shown

in Fig. 2, for the Rust version:

$$\mathsf{switch}\,(\Delta)\left\{\begin{array}{l} 0 \to \mathsf{success}(0,\emptyset) \\ 1 \to \mathsf{success}(1,\sigma) \end{array}\right\}$$

In this decision tree we identify input `Interval`s of the form $Some(\&(x_1, x_2))$. At toplevel, the decision tree checks whether the considered memory value Δ is 0 or 1, which is enough to decide whether the Interval is `None` or not. The *binding environment* σ for "success at branch 1" maps, for instance, x_1 to the left value of the interval.

4.2.2 Tags and Contexts

To fully describe memory representations and later our compilation scheme, we need two additional tools: tags and contexts. Their grammar is given in Fig. 8b.

Tags are the simplest description of the possible variants of a type. They can be either a constant of a primitive type (integer, float, char...), a constructor of a sum type or a wildcard $\top$. Other types such as references and product types do not have associated tags, as they can't give rise to alternatives in patterns by themselves.

Contexts are used to deconstruct values and types. Contexts are a subclass of patterns where all sub-patterns are wildcards, except one, which is a hole denoted by $\Box$. We will often use contexts to precisely describe the structure around subterms bound by a variable in a pattern.

Focusing is a set of operations related to contexts. $\mathbf{focus}\,(h, v)$ returns the sub-value of v at the position of $\Box$ in h. $\mathbf{focus}\,(h, \tau)$ returns the type of the subterm at the position of $\Box$ for values of type τ. Both judgments are fully defined in Appendix B. For now, we only give the intuition through Example 8.

Example 8 (Contexts for `Interval`s)**.** Let us consider the pattern $p = Some(\&(x_1, x_2))$. The context corresponding to x_2 in p is $h = Some(\&(\_, \Box))$ We thus have:

$$\mathbf{focus}\,(h, Interval) = u64$$
$$\mathbf{focus}\,(h, Some(\&(u1756, u1791))) = 1791$$

4.3 Memory representation

The compilation process depicted in Section 5 will be *parametrized* by a memory representation. The objective of this section is to define the required ingredients for this parametrization, and also define what would be a *valid* representation.

A *memory representation* $\mathfrak{R}$ is a triple $\mathfrak{R} = (\textsc{Repr}^\bullet, \textsc{Frog}^\bullet, \textsc{Knit}^\bullet)$. Indeed, giving a function $\textsc{Repr}^\bullet : \textsc{Val} \to \textsc{MemVal}$ computing memory values from values (Section 4.1) is not sufficient: decision trees should be produced with their target-dependent expressions inside switches. For instance, in Example 7, the second switch should come with a representation-dependent way to compute $(*\Delta).2$. Moreover, we also need a way to generate alternatives and their associated selection code. We propose two additional methods for which an example is given in Section 4.4. For ease of reading, operations specific to a given representation are denoted with a full dot.

- $\textsc{Frog}^\bullet_{\tau_\Delta} : \mathcal{H} \to \left(\overline{t \mapsto \mathcal{T}}\right) \to \mathcal{T}$ creates representation-specific code to choose between several alternatives in a given type[4]. It takes a context h indicating a subterm of the current pattern (the subterm of the main discriminant Δ under scrutiny) and a list of tags and their associated decision trees $\overline{(t_k, \mathcal{T}_k)}$. It returns a complete decision tree $\mathcal{T} = \mathsf{switch}\,(e)\begin{cases} w_1 \mapsto \mathcal{T}_1 \\ \vdots \\ w_n \mapsto \mathcal{T}_n \end{cases}$ where e is a representation-dependent expression that characterizes the subterm and each w_k is one of its possible runtime values. The semantics of this tree is "expression e computes a value that exactly characterizes the object inside the hole h; we choose a subtree through which to continue according to this value".
  Each tag t_k is either a possible variant for the destructed type $\mathbf{focus}\,(h, \tau_\Delta)$ that encompasses all values of this variant (see Section 4.2.2) or the special default tag $\top$ (if the image of the $\mathbf{focus}\,(h, \tau)$ function does not cover all possible values of the type τ).

[4] For crochet enthusiasts, *Frogging* is the action of undoing the stitches: rip it, ripit, ribbit, ribbit, ...

- $\textsc{Knit}^\bullet_{\tau_\Delta} : \mathcal{H} \to e$. $\textsc{Knit}^\bullet$ creates representation-specific code to build the memory value representing a subterm of the main discriminant. It takes a context that indicates the subterm under scrutiny and returns an expression computing the memory value representing this subterm. As shown in Example 7, these memory values might not be directly accessible from the parent memory value and might need to be reconstructed based on various pieces present in the parent memory value, hence the need for such a function.

Naturally, there are constraints over what determines a reasonable choice of $\textsc{Repr}^\bullet$, $\textsc{Knit}^\bullet$ and $\textsc{Frog}^\bullet$. The main idea is that $\textsc{Knit}^\bullet$ and $\textsc{Frog}^\bullet$ should be compatible with $\textsc{Repr}^\bullet$: the expression produced by $\textsc{Knit}^\bullet$ should evaluate to a final memory value which is the same as the one built by $\textsc{Repr}^\bullet$, and the decision tree produced by $\textsc{Frog}^\bullet$ should properly distinguish between different variants based on the memory values produced by $\textsc{Repr}^\bullet$.

Definition 1 (Valid Representation)**.** *We say that a triple* $\mathfrak{R} = (\textsc{Repr}^\bullet, \textsc{Frog}^\bullet, \textsc{Knit}^\bullet)$ *is a* valid representation *if, and only if, the following holds.*

Let $(t_1, \mathcal{T}_1) \dots (t_n, \mathcal{T}_n)$ *be the branches under consideration,* h *a context,* τ *the type associated with the tags and* v *a value of type* τ*.*

We define the auxiliary function $\text{Tag2Pat} : \mathit{Tags} \mapsto \textsc{Pat}$ *s.t.*

$$\text{Tag2Pat}(c) = c \qquad \text{Tag2Pat}(K) = K(\_, \dots, \_) \qquad \text{Tag2Pat}(\top) = \_$$

Then we should have:

$$\textsc{Knit}^\bullet_\tau(h)[\textsc{Repr}^\bullet_\tau(v)] \hookrightarrow \textsc{Repr}^\bullet_{\boldsymbol{focus}(h,\tau)}(\boldsymbol{focus}(h, v))$$

and:

$$\textsc{Frog}^\bullet_\tau\left(h, \overline{(t_k, \mathcal{T}_k)}\right)[\textsc{Repr}^\bullet_\tau(v)] \hookrightarrow j, \sigma$$

$$\Updownarrow$$

$$\exists k.\quad \text{Tag2Pat}(t_1) \mid \dots \mid \text{Tag2Pat}(t_n) \rhd v \to k \ \wedge\ \mathcal{T}_k[\textsc{Repr}^\bullet_\tau(v)] \hookrightarrow j, \sigma$$

4.4 A first example: the boxed representation

4.4.1 Memory values

The boxed representation maps values of all types to uniform, word-wide memory values. The complete definition is given in Fig. 10.

Constant and reference values can always fit into a single memory word and are thus stored *unboxed.* (Numeric) constants are stored as words in a type-appropriate representation/encoding. The memory value of a reference is a pointer to the memory value of the referenced value. Access is done by dereferencing the pointer. Tuple and constructor values may not fit into a single word, hence the need for *blocks*, which are word-aligned contiguous arrays. More precisely, tuples are represented by pointers to blocks in which the k-th word is a memory value representing the k-th field of the tuple. The representation of constructor values is similar, with the index of the constructor among all possible constructors for the considered type stored in the first word of the block and the k-th field in the $k+1$-th word. Accessing the k-th field of such a value consists in dereferencing the pointer and accessing the k-th or $k+1$-th word of the block.

Example 9 (Boxed representation, memory values)**.** Let us consider the type $\tau_0 = \mathit{None} + \mathit{Some}(A + B + C(\mathit{Int}_{32}))$, which represent an optional value, itself containing three possible cases. Here are a few examples of values according to the boxed representation:

$$\textsc{Repr}^\bullet_{\tau_0}(\mathit{None}) = \&_{64}[0]_{64} \qquad \textsc{Repr}^\bullet_{\tau_0}(\mathit{Some}(A)) = \&_{64}[1,\ \&_{64}[0]_{64}]_{64}$$
$$\textsc{Repr}^\bullet_{\tau_0}(\mathit{Some}(B)) = \&_{64}[1,\ \&_{64}[1]_{64}]_{64} \qquad \textsc{Repr}^\bullet_{\tau_0}(\mathit{Some}(C(n))) = \&_{64}[1,\ \&_{64}[2,\ n]_{64}]_{64}$$

$$\textsc{Repr}^\bullet_T : c \mapsto c$$
$$\textsc{Repr}^\bullet_{\&\tau} : \&v \mapsto \&_{64}(\textsc{Repr}^\bullet_\tau(v))$$
$$\textsc{Repr}^\bullet_{\prod \tau_i} : \langle \overline{v_i} \rangle \mapsto \&_{64}\left[\overline{\textsc{Repr}^\bullet_{\tau_i}(v_i)}\right]_{64}$$
$$\textsc{Repr}^\bullet_{\sum K_i(\overline{\tau_{i,j}})} : K_i(\overline{v_j}) \mapsto \&_{64}\left[i,\ \overline{\textsc{Repr}^\bullet_{\tau_{i,j}}(v_j)}\right]_{64}$$

(a) Memory values

$$f(\square, e) = e$$
$$f(\&h, e) = f(h, *e)$$
$$f(\langle h_k \rangle, e) = f(h, *e.k)$$
$$f(K(h_k), e) = f(h, *e.k + 1)$$
$$\textsc{Knit}^\bullet(h) = f(h, \Delta)$$

(b) Knitting

Frogging, constant type – focus $(h, \tau) = T$

$$\textsc{Frog}^\bullet_\tau(h)\left(\overline{(c_k, \mathcal{T}_k)} \cdot (\top, \mathcal{T}_*)\right) = \mathsf{switch}\,(\textsc{Knit}^\bullet(h)) \begin{cases} \overline{c_k \mapsto \mathcal{T}_k} \\ \top \mapsto \mathcal{T}_* \end{cases}$$

Frogging, sum type – focus $(h, \tau) = \sum K_i(\overline{\tau})$

$$\textsc{Frog}^\bullet_\tau(h)\left(\overline{(K_k, \mathcal{T}_k)}\right) = \mathsf{switch}\,((*\textsc{Knit}^\bullet(h)).0) \left\{ \overline{k \mapsto \mathcal{T}_k} \qquad \text{with } \{\overline{K_k}\} \text{ a complete signature} \right.$$
$$\textsc{Frog}^\bullet_\tau(h)\left(\overline{(K_{i_k}, \mathcal{T}_k)} \cdot (\top, \mathcal{T}_*)\right) = \mathsf{switch}\,((*\textsc{Knit}^\bullet(h)).0) \begin{cases} \overline{i_k \mapsto \mathcal{T}_k} \\ \top \mapsto \mathcal{T}_* \end{cases}$$

(c) Frogging

Figure 10: Boxed representation with 64-bit words

4.4.2 Representation (Knit&Frog)

To distinguish between constructors, $\textsc{Frog}^\bullet$ emits code which dereferences the pointer and accesses the first word of the block, and $\mathsf{switch}\,()$ on it.

The boxed representation has the remarkable property of being *composable*: memory value representing a value always contains the memory values of its subterms. Concretely, almost all functions manipulating the representation can be defined by induction without much inspection of the sub-types. For instance, $\textsc{Repr}^\bullet$, defined in Fig. 10a, is a direct induction over the structure of values. Similarly, knitting is defined through an auxiliary function f shown in Fig. 10b which takes a context, and an expression computing the parent value of that context. It then inductively accumulates the expression by deconstructing the context. $\textsc{Knit}^\bullet$ is a restriction of f to the main discriminant Δ. Furthermore, accesses to constructors of values are uniform across types and variants. $\textsc{Frog}^\bullet$, defined in Fig. 10c, directly uses $\textsc{Knit}^\bullet$ to reach the constructor of a memory value and always yields a single switch to distinguish between all variants of a given type.

Example 10 (Boxed representation, cont')**.** Examples of $\textsc{Knit}^\bullet$ and $\textsc{Frog}^\bullet$ on values of type τ_0 of Example 9 are given in Fig. 11. For $\textsc{Knit}^\bullet$, to compute the subterm induced by a context, we simply access the piece of memory contained at its position with simple dereferencing and indexed access. The decision tree $\textsc{Frog}^\bullet_{\tau_0}(\mathit{Some}(\square))\,((B, \mathcal{T}_1) \cdot (\top, \mathcal{T}_*))$ should distinguish *whether the hole* is a B or anything else. It is thus sufficient to look at the discriminant $(*(*\Delta).1).0$ (Example 9). In the general case $\textsc{Frog}^\bullet$ accesses the relevant tag and distinguish cases in a uniform manner.

The boxed memory values and its $\textsc{Knit}^\bullet$ and $\textsc{Frog}^\bullet$ operations closely mimic pattern matching on values in the surface language, we thus state the following result:

Lemma 1. The boxed representation is a valid representation.

As we have seen, thanks to its very uniform definition the boxed representation is easy to state and manipulate. $\textsc{Knit}^\bullet$ and $\textsc{Frog}^\bullet$ can be defined in term of each other, and a call to $\textsc{Frog}^\bullet$ always yields a single switch. This is rarely the case in more complex representations. Naturally, this simplicity induces

Knitting

$$\textsc{Knit}^{\bullet}_{\tau_0}(\mathit{Some}(\square)) = (*\Delta).1 \qquad \textsc{Knit}^{\bullet}_{\tau_0}(\mathit{Some}(C(\square))) = *\left((*\Delta).1\right).1$$

Frogging

$$\textsc{Frog}^{\bullet}_{\tau_0}(\square)\left((\mathit{None}, \mathcal{T}_0) \cdot (\mathit{Some}, \mathcal{T}_1)\right) = \mathsf{switch}\left((*\Delta).0\right) \begin{cases} 0 \mapsto \mathcal{T}_0 \\ 1 \mapsto \mathcal{T}_1 \end{cases}$$

$$\textsc{Frog}^{\bullet}_{\tau_0}(\mathit{Some}(\square))\left((B, \mathcal{T}_1) \cdot (\top, \mathcal{T}_*)\right) = \mathsf{switch}\left((*(*\Delta).1).0\right) \begin{cases} 1 \mapsto \mathcal{T}_1 \\ \top \mapsto \mathcal{T}_* \end{cases}$$

$$\textsc{Frog}^{\bullet}_{\tau_0}(\mathit{Some}(C(\square)))\left((42, \mathcal{T}_{42}) \cdot (\top, \mathcal{T}_*)\right) = \mathsf{switch}\left((*(*\Delta).1).1\right) \begin{cases} 42 \mapsto \mathcal{T}_{42} \\ \top \mapsto \mathcal{T}_* \end{cases}$$

Figure 11: Memory values, knitting and frogging for values of type τ_0

numerous inefficiencies: Every constructor is represented as a pointer to a block, including argument-less variants such as *None* which could easily be represented as unboxed integers. Frogging a value with the context $\mathit{Some}(\square)$ thus requires dereferencing two pointer, while only one might be necessary for a more packed representation. Mainstream languages with ADTs such as OCaml, Haskell, or Rust apply such optimization to their representation with various means of differentiating between constructors and values. Section 6 showcases more optimized representation in order to demonstrate our compilation process in a more realistic setting.

5 Representation-dependent compilation

We now define our compilation scheme. In the previous sections, we have defined pieces of the target decision trees we want to emit. In isolation, those pieces are easy to deduce from simple patterns. The difficulty, however, comes with composition: how to handle nested patterns and how to expose back the values bound through variables in a way that is compatible with the chosen memory representation. This composition is precisely the object of our compilation procedure.

Our procedure is inspired by Maranget (2008). The novelty here is that this procedure is now parameterized by a *valid representation* and creates decision trees manipulating memory values directly. Our contribution is the careful split between the representation-dependent elements, which are parameterized over and delegated to the $\textsc{Knit}^{\bullet}$ and $\textsc{Frog}^{\bullet}$ functions as described in the previous sections, and the representation-independent aspects which are described here. Additionally, we add handling of the bound environment, necessary for the rest of the compilation process.

Let us start by stating our goal: we aim to compile a matching m to a *decision tree* $\mathcal{T}$ that takes a memory value r as input and evaluates to an output (denoted by $\mathcal{T}[\overline{r_i}] \hookrightarrow j, \sigma$, see Section 4.2.1) consistent with the matching judgment on the source values.

More formally, given a (source-level) value v of type τ, we want :

$$m \rhd v \to j, \sigma \iff \mathcal{T}[\textsc{Repr}^{\bullet}_{\tau}(v)] \hookrightarrow j, \left\{x \mapsto \textsc{Repr}^{\bullet}_{\tau_x}(y) \mid (x \mapsto y) \in \sigma\right\}$$

In the rest of this section, we consider as an implicit ambient parameter a *valid representation* $\mathfrak{R} = (\textsc{Repr}^{\bullet}, \textsc{Frog}^{\bullet}, \textsc{Knit}^{\bullet})$. As stated before, operations specific to a given representation are denoted with a full dot: $\textsc{Repr}^{\bullet}$. Operations which are *parameterized* by the representation are denoted with an empty dot: $\textsc{Compile}^{\circ}_{\tau_\Delta}$. Operations without any dot are representation-agnostic.

5.1 Algorithm

$\textsc{Compile}^{\circ}_{\tau_\Delta}(\mathcal{P})$ compiles the given patterns to a decision tree. It takes two parameters: the type of main discriminant τ_Δ, and the *pattern matrix* $\mathcal{P}$, which gathers all the necessary ingredients. In its most general form, the pattern matrix is composed as so:

$$\mathcal{P} = \left(\begin{array}{ccccc|c} h_1 & \dots & h_i & \dots & h_n & \\ \hline p_1^1 & & p_i^1 & & p_n^1 & j^1, s^1 \\ \vdots & & \vdots & & \vdots & \vdots \\ p_1^\ell & \dots & p_i^\ell & \dots & p_n^\ell & j^\ell, s^\ell \\ \vdots & & \vdots & & \vdots & \vdots \\ p_1^m & & p_i^m & & p_n^m & j^m, s^m \end{array}\right)$$

where:

- The input matching problem is encoded in rows: row k represents a case of the match $(\mid (p_1^k, \dots, p_n^k))$. These rows end with a main branch index (j^k) and a binding environment (s^k). Importantly, we consider a *vector* of discriminants instead of a single one as done previously, allowing us to split product patterns and choose which one to inspect first.
- Each column represents the different parts of the discriminant under consideration. Their headers h_i are contexts that indicate which parts of the main discriminant this column is matching against, as a subterm of the main discriminant Δ . For instance, on inspecting an optional integer, we might discover that it is a $\mathit{Some}(\dots)$ and want to explore the subterm in context $\mathit{Some}(\square)$ of the discriminant Δ.

$\textsc{Compile}^{\circ}$ proceeds by recursively emitting Switch nodes, and reducing the pattern matrix until exhaustion. It has five cases, depicted in Fig. 12.

- If $\mathcal{P}$ is empty, the branch is unreachable as there are no patterns to inspect the discriminant.
- If the first row consists only of wildcards, the matching can only succeed.
- If there exists a variable pattern p_i^ℓ and the previous rows contain no variable patterns, we introduce p_i^ℓ in the binding environment.
- If there exists an "or"-pattern $p_i^\ell = p_1 \mid p_2$ and the previous rows contain no "or"-patterns, split it.
- Otherwise, we must build an actual switch node. This case is more involved and detailed in the rest of this section.

The aim of the switch node is to inspect the head constructor of the value contained in one of the discriminants against the patterns of its column in $\mathcal{P}$, and to branch to the sub-trees that perform the remaining computations (on other columns and nested values). Note that we assume $\mathcal{P}$ does not contain any variable or "or"-patterns, as those are removed by the two previous cases. The construction of a switch node (last case of Fig. 12) is done in four steps, detailed as follows.

1. **Pick a column to inspect**
   We first pick an arbitrary column i such that $p_i^1 \neq \_$ to generate a node switching on that column. This pick is done by the function $\textsc{PickColumn}$, which takes the pattern matrix $\mathcal{P}$, and returns a column index. The choice of column can be done through heuristics as described by Maranget (2008); Scott and Ramsey (2000). It does not affect our safety proof.

2. **Identify all the necessary branches for the chosen column**
   Now that we have chosen a column, we need to identify the different branches of our switch node. Let us consider the i-th column: it is associated with a discriminant designated by a context h_i. We can obtain its type thanks to focusing: $\tau_i = \mathbf{focus}\,(h_i, \tau_\Delta)$. The set of necessary branches depends directly on the type of the column τ_i: if the discriminant represents a reference or tuple value, then only one possible branch is needed. Otherwise, if τ_i is a primitive or sum type, several branches are needed. More precisely, the generated switch node should include a branch for each tag that appears as the head constructor of a pattern in column i, with an additional default branch if those constructors do not encompass all possible values of this type. The auxiliary representation-independent function $\textsc{GetTags}_{\tau_i}$ defined below

$$\textsc{Compile}^{\circ}_{\tau_\Delta}(\emptyset) = \mathsf{unreachable} \qquad \text{(No pattern case)}$$

$$\textsc{Compile}^{\circ}_{\tau_\Delta}\left(\begin{array}{ccc|c} h_1 & \dots & h_n & \\ \hline \_ & \dots & \_ & j^1, s^1 \\ \vdots & \ddots & \vdots & \vdots \\ p_1^m & \dots & p_n^m & j^m, s^m \end{array}\right) = \mathsf{success}(j^1, s^1) \qquad \text{(Wildcard case)}$$

$$\textsc{Compile}^{\circ}_{\tau_\Delta}\left(\begin{array}{ccc|c} h_1 & h_i & h_n & \\ \hline p_1^1 & p_i^1 & p_n^1 & j^1, s^1 \\ \vdots & \vdots & \vdots & \vdots \\ p_1^\ell & x & p_n^\ell & j^\ell, s^\ell \\ \vdots & \vdots & \vdots & \vdots \\ p_1^m & p_i^m & p_n^m & j^m, s^m \end{array}\right) = \textsc{Compile}^{\circ}_{\tau_\Delta}\left(\begin{array}{ccc|c} h_1 & h_i & h_n & \\ \hline p_1^1 & \dots & p_n^1 & j^1, s^1 \\ \vdots & \vdots & \vdots & \vdots \\ p_1^\ell & \_ & p_n^\ell & j^\ell, s^\ell \cup s_x \\ \vdots & \vdots & \vdots & \vdots \\ p_1^m & \dots & p_n^m & j^m, s^m \end{array}\right)$$

$$\text{where } s_x = \left\{x \to \textsc{Knit}^{\bullet}_{\tau_\Delta}(h_i)\right\} \qquad \text{(Variable case)}$$

$$\textsc{Compile}^{\circ}_{\tau_\Delta}\left(\begin{array}{ccc|c} h_1 & h_i & h_n & \\ \hline p_1^1 & \dots & p_n^1 & j^1, s^1 \\ \vdots & \vdots & \vdots & \vdots \\ p_1^\ell & (p \mid q) & p_n^\ell & j^\ell, s^\ell \\ \vdots & \vdots & \vdots & \vdots \\ p_1^m & \dots & p_n^m & j^m, s^m \end{array}\right) = \textsc{Compile}^{\circ}_{\tau_\Delta}\left(\begin{array}{ccc|c} h_1 & h_i & h_n & \\ \hline p_1^1 & \dots & p_n^1 & j^1, s^1 \\ \vdots & \vdots & \vdots & \vdots \\ p_1^\ell & p & p_n^\ell & j^\ell, s^\ell \\ p_1^\ell & q & p_n^\ell & j^\ell, s^\ell \\ \vdots & \vdots & \vdots & \vdots \\ p_1^m & \dots & p_n^m & j^m, s^m \end{array}\right) \qquad \text{(Or case)}$$

$$\textsc{Compile}^{\circ}_{\tau_\Delta}(\mathcal{P}) = \begin{cases} i \leftarrow \textsc{PickColumn}(\mathcal{P}) & (1) \\ \text{Let } h_i \text{ the } i\text{-th context in } \mathcal{P} \text{ and } \tau_i = \mathbf{focus}\,(h_i, \tau_\Delta) & \\ \mathit{Tags} \leftarrow \textsc{GetTags}_{\tau_i}(i, \mathcal{P}) & (2) \\ \mathcal{C} \leftarrow \bigsqcup_{tag \in Tags} \left(t, \textsc{CompileBranch}^{\circ}_{\tau_\Delta}(\mathcal{P})\,(i, tag)\right) & (3) \\ \textsc{Frog}^{\bullet}_{\tau_\Delta}(h_i)\,(\mathcal{C}) & (4) \end{cases} \qquad \text{(Switch case)}$$

Figure 12: The compilation procedure – $\textsc{Compile}^{\circ}_{\tau_\Delta}(\mathcal{P})$

enumerates the *tags* (as defined in Section 4.2.2) associated with the necessary sub-trees. It works by collecting all the tags present in the pattern matrix at the column i. For constant types, we always add a "catch-all" case $\top$. For sum types, we add it only if the collected tags do not cover all the constructors in the type τ_i. Finally, for all other cases, as described before, there is only one case corresponding to the generic $\top$ tag.

$$\textsc{GetTags}_T(i, \mathcal{P}) = \{c \mid \exists j. p_i^j = c\} +\!\!+ \top \qquad \text{where } +\!\!+ \text{ is concatenation}$$

$$\textsc{GetTags}_\Sigma(i, \mathcal{P}) = \{K \mid \exists j. p_i^j = K(\dots)\} +\!\!+ \begin{cases} \{\top\} & \text{if } \exists K. \forall j, p_i^j = K'(\dots) \wedge K' \neq K \\ \{\} & \text{otherwise} \end{cases}$$

$$\textsc{GetTags}_\_(i, \mathcal{P}) = \top$$

3. **Compile the decision sub-trees corresponding to each branch**
   Now that we have identified the tag corresponding to each branch, we can compute the corresponding subtree for each tag. Each subtree is the result of the compilation of a new pattern matrix, as done by

Inputs		**Outputs**	
τ	*tag*	$\mathcal{H}$	f
T	c	$()$	$c \mapsto ()$ $c' \mapsto \emptyset \quad c' \neq c$ $\_ \mapsto ()$
T	$\top$	$()$	$c \mapsto \emptyset$ $\_ \mapsto ()$
$\&\tau$	$\top$	$(h[\&\square])$	$\&q \mapsto (q)$ $\_ \mapsto (\_)$
$\prod_{1\leq k\leq \ell} \tau_k$	$\top$	$\big(h\,[h_0] \quad \cdots \quad h\,[h_\ell]\big)$ with $h_k = \langle \ldots, \square_k, \ldots \rangle$ in the k^{th} position	$\langle q_0, \ldots, q_\ell \rangle \mapsto \big(q_0 \quad \cdots \quad q_\ell\big)$ $\_ \mapsto \big(\_ \quad \cdots \quad \_\big)$
$\sum_{1\leq i\leq \ell} K_i(\tau_{i,0}, \ldots, \tau_{i,n_i})$	K_{i_0}	$\big(h\,[h_0] \quad \cdots \quad h\,[h_{n_{i_0}}]\big)$ with $h_k = K_{i_0}(\ldots, \square_k, \ldots)$ in the k^{th} position	$K_{i_0}(q_0, \ldots, q_{n_{i_0}}) \mapsto \big(q_0 \quad \cdots \quad q_{n_{i_0}}\big)$ $K_i(\ldots) \mapsto \emptyset$ $\_ \mapsto \big(\_ \quad \ldots \quad \_\big)$
$\sum_{1\leq i\leq \ell} K_i(\tau_{i,0}, \ldots, \tau_{i,n_i})$	$\top$	$()$	$K_i(\ldots) \mapsto \emptyset$ $\_ \mapsto ()$

Table 1: Input expansion and pattern specialization – $\textsc{Expand}^\circ(\tau, tag, h)$

the $\textsc{CompileBranch}^\circ$ function.

$$\textsc{CompileBranch}^\circ_{\tau_\Delta}(\mathcal{P})(i, tag) = \textsc{Compile}^\circ_{\tau_\Delta}(\mathcal{P}')$$
$$\text{where} \left| \begin{array}{l} \text{Let } h_i \text{ the } i\text{-th context in } \mathcal{P} \text{ and } \tau_i = \mathbf{focus}(h_i, \tau_\Delta) \\ \mathcal{H}, f = \textsc{Expand}^\circ(\tau_i, tag, h_i) \\ \mathcal{P}' = \textsc{FilterMapAtColumn}(\mathcal{H}, f, \mathcal{P}, i) \end{array} \right.$$

The semantics is given by the function $\textsc{Expand}^\circ$ defined in Table 1. $\textsc{Expand}^\circ(\tau, tag, h)$ takes as parameters a tag, the context h of the considered column and its type τ. It returns an expansion $\mathcal{H}, f$: $\mathcal{H}$ defines the new columns and f defines how to expand the cells of the matrix. f maps patterns to either vectors or the empty set. Concretely, $\textsc{Expand}^\circ$ carries out two tasks:

Specialization Only keep cases (i.e., rows) in the matrix whose i-th pattern is compatible with the given tag. For instance, if we consider the branch with tag *Some*, rows with a *None* pattern in position i are removed. This is done in f by returning $\emptyset$ for incompatible cases.

Expansion We replace the i-th column in each input with new columns for the nested patterns. For instance, when considering the tag *Node*, we expand the pattern $Node(p_v, p_1, p_2)$, thus replacing the column i with three new columns, one for each sub-pattern. This is done by adding new columns in $\mathcal{H}$ and replacing the column in f.

The actual new pattern matrix $\mathcal{P}'$ that serves as input for the recursive call is computed by the operation $\textsc{FilterMapAtColumn}(\mathcal{H}, f, \mathcal{P}, i)$. It first extends the contexts contained in the column headers by replacing h with $\mathcal{H}$. It then applies f to each cell p_i^j of the column i in matrix $\mathcal{P}$. If $f(p_i^j)$ is the empty set, the whole row at j is deleted. Otherwise, p_i^j is replaced with the cells returned by f. For each cell, f must return vectors of the length of $\mathcal{H}$ so that the resulting matrix has a consistent size.

4. **Assemble all these sub-trees into a single tree of Switch nodes**
We have now gathered all the tags that we need to branch from, along with all the sub-trees corresponding to these branches. We can finally assemble them together. This assembly is representation-specific: indeed, the way to distinguish, for instance, between *Some* and *None* constructors depends on the type under consideration and the choice of representation. We thus delegate the final assembly to the $\textsc{Frog}^\bullet$ function. $\textsc{Frog}^\bullet_{\tau_\Delta}(h, \mathcal{C})$ takes as parameters the type of the main discriminant τ_Δ, the context h of the column under consideration, and the list of all cases $\mathcal{C}$ (i.e., pairs of tags and sub-trees). It returns a tree as defined in Section 4.3.

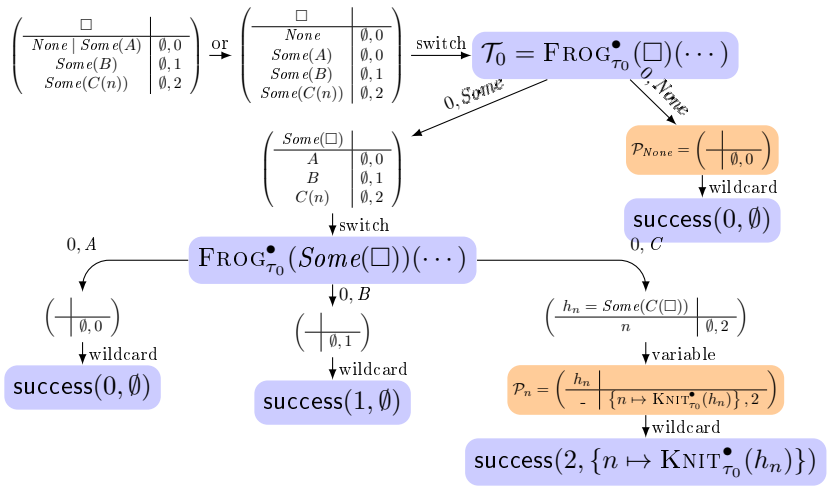


Figure 13: Pattern compilation for $m_0 = (None \mid Some(A)) \mid Some(B) \mid Some(C(n))$.

6 End-to-end examples

We now define examples of memory representations and how our compilation procedure behaves on them. We first illustrate our compilation procedure on concrete pattern matches. We then showcase a real-world implementation based on the OCaml one. We then define the "packed" representation, which attempts to compact memory values as much as possible.

6.1 Compilation procedure

Following our illustration of the boxed representation in Section 4.4, we consider the type $\tau_0 = None + Some\,(A + B + C(Int_{32}))$. The match m_0 and the associated pattern matrix $\mathcal{P}_0$ are shown below:

$$m_0 = \text{Match}\left\{\begin{array}{ll} \mid None \mid Some(A) & \to 0 \\ \mid Some(B) & \to 1 \\ \mid Some(C(n)) & \to n+1 \end{array}\right\} \qquad \mathcal{P}_0 = \left(\begin{array}{c|c} \square & \\ \hline None \mid Some(A) & \emptyset, 0 \\ Some(B) & \emptyset, 1 \\ Some(C(n)) & \emptyset, 2 \end{array}\right)$$

The derivation of the compilation is presented in Fig. 13. Nodes that result in decision trees are highlighted in blue. Some matrices of particular interest are highlighted in orange and named. The calls to $\textsc{Frog}^\bullet$ and $\textsc{Knit}^\bullet$ are left as-is, since we have not specified any memory representation. The algorithm starts with an `or` rule to split the nested patterns into lines, followed by a `switch` rule along the only column of context $\square$, thus producing the $\mathcal{T}_0$ node. The context on which the switch is split is shown in the call to $\textsc{Frog}^\bullet$. The type at this position is of the form $None + Some(\ldots)$. Furthermore, the *None* and *Some* tags are both present in the column. GetTags thus gives two branches: *None* and *Some*. The $\textsc{CompileBranch}^\circ$ operation applies input expansion and pattern specialization to obtain the pattern matrices in each of these branches. In the *None* branch, the pattern matrix $\mathcal{P}_{None}$ has one line (the *None* case in the original code) and no column. Indeed, pattern specialization mandates that only the lines that could match *None* are still present, and there is only one. Input expansion mandates only columns for "interesting" sub-patterns should remain, and there are none. Since all (zero) potential columns are wildcards, we use the `wildcard` rule to produce a success() leaf. In the *Some* branch, however, pattern specialization gives us three potential patterns to inspect but removes the outer *Some*, which is now unneeded. The rest of the compilation proceeds by successive application of the `switch` and `wildcard` rules. Finally, in the matrix $\mathcal{P}_n$, which corresponds to the context $h_n = Some(C(\square))$, we use the `variable` rule to add n to the binding environment, using $\textsc{Knit}^\bullet$, before producing the final success() leaf.

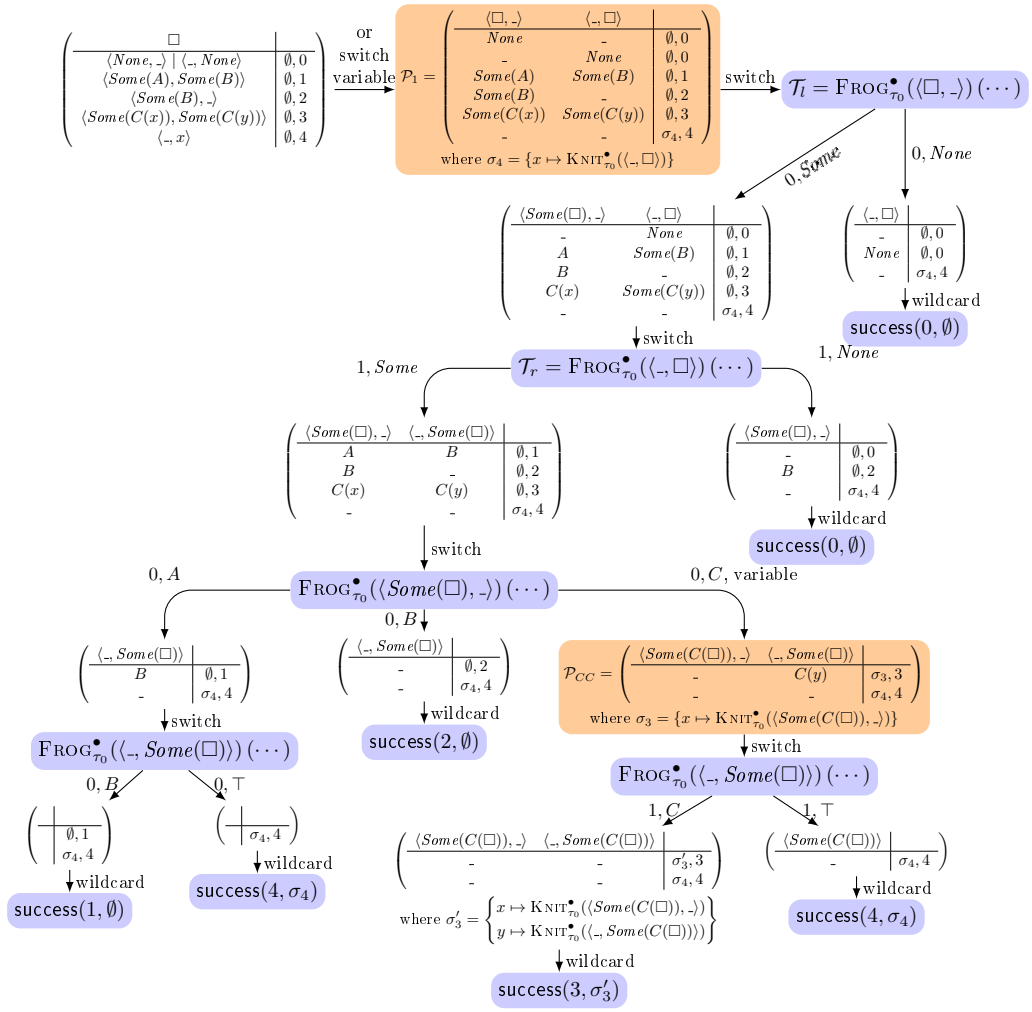


Figure 14: Pattern compilation for m_1.

We now consider a more complex example in Fig. 14 corresponding to the following matching:

$$m_1 = \text{Match} \left\{ \begin{array}{ll} \mid \langle None, \_ \rangle \mid \langle \_, None \rangle & \to None \\ \mid \langle Some(A), Some(B) \rangle & \to Some(A) \\ \mid \langle Some(B), \_ \rangle & \to Some(B) \\ \mid \langle Some(C(x)), Some(C(y)) \rangle & \to Some(C(x+y)) \\ \mid \langle \_, x \rangle & \to x \end{array} \right\}$$

This example matches on a *tuple* of elements of τ_0. In this example, some rule applications were fused for ease of reading. In the first steps, the tuples are translated, thanks to a `switch` case, into multiple columns in the matrix. This is visible in the $\mathcal{P}_1$ matrix. Each creation of a `switch` node will then pick a given column. For instance, the first $\mathcal{T}_l$ node picks the first column (context $\langle \square, \_ \rangle$) and the second $\mathcal{T}_r$ node picks the second column (context $\langle \_, \square \rangle$).

The $\mathcal{P}_{CC}$ matrix presents a particularity: the first column is composed entirely of wildcards. This particular column can thus not be picked (indeed, the algorithm would loop). We must switch on the second column ($\langle \_, Some(\square) \rangle$). The type at this context is of the form $A + B + C(\ldots)$; however, only C is present in the pattern. GETTAGS thus only produces two branches: C and $\top$.

Many of the matrices where the `wildcard` rule is applied are composed of several lines. This is expected: when looking at the original code, while no case is fully redundant, there are numerous overlaps in which case the first pattern wins. It is thus crucial that the `wildcard` rule only applies to the *first* line of a pattern

matrix.

The rest of the compilation is done by applying the `switch`, `variable` and `wildcard` rules.

6.2 The OCaml representation

We now look at a real-world example of memory representation: the OCaml one, depicted in Fig. 15. We picked this representation since it is uniform and well documented Minsky and Madhavapeddy (2021), allowing us to describe it almost completely in our formalism. Indeed, the OCaml representation is similar to the boxed representation described in Section 4.4, in that memory values are either unboxed immediate objects or pointers to blocks, each one-word wide. As this representation is rather uniform and composable, both knitting and frogging remain relatively straightforward.

6.2.1 Memory values

As in the boxed representation, OCaml blocks are contiguous word-aligned arrays. Every block starts with a *header* word, whose last byte (called *tag*) provides information about the variant and type of the underlying value. The rest of the header word is composed of information such as block length or GC markers. Since this information is irrelevant for our purposes here (curious readers can enjoy the full details in Minsky and Madhavapeddy (2021)) we will omit them and consider that the header is an integer indicating the variant under consideration. A major difference from the boxed representation is that argument-less variants of sum types are represented by an unboxed integer, rather than empty blocks. The integer is the index of the variant amongst argument-less variants. For instance, *None* will be mapped to the same representation as 0, without any boxing. As a consequence, the type associated with a memory value is no longer sufficient to determine whether it is an unboxed constant or a pointer. In order 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 block pointer. Blocks are always aligned on 8 bytes, guaranteeing their last bit is always 0. A further consequence is the loss of one bit of precision for constants. This is not problematic for most primitive types[5]. Finally, even though OCaml does offer a concept of "reference", they are defined as a record (i.e., product types) with a mutable field, and follow the associated representation rather than a separate, pointer-like structure. Reference values are therefore omitted here.

Example 11 (OCaml representation, memory values)**.** As in Section 4.4, we demonstrate the representation on the type $\tau_0 = \mathit{None} + \mathit{Some}(A + B + C(\mathit{Int}_{32}))$. Here are a few examples of values:

$$\textsc{Repr}^{\bullet}_{\tau_0}(\mathit{None}) = \texttt{0x1} \qquad \textsc{Repr}^{\bullet}_{\tau_0}(\mathit{Some}(C(n))) = \&_{64}\,[\texttt{0x0}; \&_{64}\,[\texttt{0x0}; ((n << 1) \mid \texttt{0x1})]_{64}]_{64}$$

$$\textsc{Repr}^{\bullet}_{\tau_0}(\mathit{Some}(A)) = \&_{64}\,[\texttt{0x0}; \texttt{0x01}]_{64} \qquad \textsc{Repr}^{\bullet}_{\tau_0}(\mathit{Some}(B)) = \&_{64}\,[\texttt{0x0}; \texttt{0x11}]_{64}$$

6.2.2 Representation (Knit&Frog)

As with the boxed representation, we first define an auxiliary function f that inductively deconstructs a context, then use f to define $\textsc{Knit}^{\bullet}$ and the discriminant portion of $\textsc{Frog}^{\bullet}$. f is identical to its boxed representation analog. The small optimization on argument-less variants has no effect on knitting, as no context can ever involve such variants (they are, indeed, argument-less).

Frogging undergoes more significant changes. First, numeric constants are transformed to make them match their corresponding memory values with an appropriate shift. Second, since the memory values representing constructor values are no longer uniform across variants, frogging for a sum type potentially yields a two-tiered switch structure. We first consider the case where the considered variants are either all without arguments or all with arguments. The no-arguments case is similar to frogging for primitive types. The case with argument variants involves accessing the lowest byte of the block header after dereferencing the block pointer, then comparing it to the considered constructor indexes. In the general case where both variants with or without arguments are present, the generated code first inspects the lowest bit to determine

[5] Floating-point numbers take up a whole word and are thus boxed in a block whose second word contains the raw, full-width constant.

whether the frogged memory value represents a constructor value with or without arguments; it then branches to a second switch that specifically discriminates between different constructors. The default tag, if present, is propagated to both sub-trees since it accepts either type of variants. Note that this might leave the inner switches with only a single branch (consider, for instance, the option type). For ease of reading, our definition does not remove such single-branch switches, but it is trivial to do so.

Example 12 (OCaml representation, cont')**.** As said before, the $\textsc{Knit}^\bullet$ function behaves similarly to the one from the boxed representation. For instance, at the context $\mathit{Some}(\square)$, it dereferences the pointer to access the block, and accesses its first field: $\textsc{Knit}^\bullet_{\tau_0}(\mathit{Some}(\square)) = *\Delta.2$

$\textsc{Frog}^\bullet$, on the other hand, must take into account the particular representation of argument-less variants. For instance, still at position $\mathit{Some}(\square)$ with the type τ_0, we consider the pattern where the branches B, C and $\top$ are present. In that case, two switches are required: first to know if the value under consideration is an argument less variant or not by switching on the expression $*\Delta.2$ & `0x1`. If it is not argument-less, it must necessarily be the C branch. Otherwise, we must test if the variant under consideration is B or not by switching on $*\Delta.2$.

$$\textsc{Frog}^\bullet_{\tau_0}(\mathit{Some}(\square))\,((B,\mathcal{T}_B)\cdot(C,\mathcal{T}_C)\cdot(\top,\mathcal{T}_*)) = \mathsf{switch}\,(*\Delta.2\ \&\ \texttt{0x1})\begin{cases}0\mapsto\mathcal{T}_C\\ 1\mapsto \mathsf{switch}\,(*\Delta.2)\begin{cases}\texttt{0x11}\mapsto\mathcal{T}_B\\ \top\mapsto\mathcal{T}_*\end{cases}\end{cases}$$

Since the OCaml representation is fairly close to the boxed one, we can state:

Lemma 2. The OCaml representation is a valid representation.

6.2.3 Compilation

Finally, we present the full compilation of the second example of Section 6.1. The compilation procedure itself was illustrated in Fig. 14 through a graph of the recursive calls of $\textsc{Compile}^\circ$ where the calls to $\textsc{Frog}^\bullet$ and $\textsc{Knit}^\bullet$ were left unresolved. Fig. 16 shows the final decision tree output by our prototype tool, `ribbit`. Rectangle white nodes are switch () nodes, with the cases in squares on the right. The green nodes represent success() leafs, with the left part being the label, and the right part the binding environment. The choice of order in which to split the columns is identical to the one in Fig. 16. We first inspect each part of the tuple to distinguish between *None* and *Some* (recall that a tuple is a block starting with a tag, hence the members of a couple are at index 1 and 2). In the original derivation, we now need to distinguish between A, B and $C(\dots)$. This is done by two switches: first "$(*(*\Delta).1).1$ & `0x1`" to distinguish between argument-less variants $A+B$ on one hand, and variants with arguments C on the other; then "$(*(*\Delta).1).1$" to distinguish between A and B. The rest of the tree proceeds similarly. As before, our procedure doesn't do any sort of common sub-expression elimination, hence a certain redundancy in the result. We assume the follow-up compilation process, which will also choose how to effectively compile the switches, is a better place to apply such optimizations.

6.3 The Packed Representation

The Packed Representation is a memory representation which attempts to maximally pack each type, regardless of how it is used: for each type, we define its necessary bit-word length and a function from values to bit-words of that length.

6.3.1 Memory values

More formally, let τ a type. Let $\mathcal{V}_\tau = \{v \mid \vdash v : \tau\}$ its values. Since all our base types are finite and there are no recursive types, $\mathcal{V}_\tau$ is finite: let $|\mathcal{V}_\tau|$ its cardinal. By definition, there exists an injection, denoted $\textsc{Repr}^\bullet_\tau$, from $\mathcal{V}_\tau$ to words of size up to $\lceil log_2(|\mathcal{V}_\tau|)\rceil$.

Note that this function is certainly not defined inductively over τ! Additionally, such representation admits no sharing in general, and might have inefficient building and matching of values.

Example 13 (Packed representation, memory values)**.** For instance, given type $\tau_0 = None + Some(A + B + C(Int_{32}))$, the cardinal is $2^{32} + 3$, leading to values in 34-bit words, and the following definition:

$$\textsc{Repr}^\bullet_{\tau_0}(None) = 0 \qquad \textsc{Repr}^\bullet_{\tau_0}(Some(A)) = 1$$
$$\textsc{Repr}^\bullet_{\tau_0}(Some(B)) = 2 \qquad \textsc{Repr}^\bullet_{\tau_0}(Some(C(i))) = i + 3$$

6.3.2 Representation (Knit&Frog)

Let us first remark that given a type τ, a context h generates a subset of the image of $\textsc{Repr}^\bullet_\tau$. Let us denote $Img_\tau(h)$ this subset.

$\textsc{Knit}^\bullet$ can then be defined by using the function $m \mapsto \textsc{Repr}^\bullet_{\mathbf{focus}(h,\tau)}\left(\mathbf{focus}\left(h, \textsc{Repr}^{\bullet\,-1}_\tau(m)\right)\right)$, which forms a total mapping from $Img_\tau(h)$ to $Img_{\mathbf{focus}(h,\tau)}(\square)$. This mapping can be compiled to target expressions using arithmetic and binary operations.

Example 14 (Packed representation, $\textsc{Knit}^\bullet$)**.** Following the previous example, the context $Some(C(\square))$ corresponds to the subset $[3 \dots 2^{34} - 1]$. To map this interval to $[0 \dots 2^{32} - 1]$, we can use arithmetic and shifting: $\textsc{Knit}^\bullet_{\tau_0}(h, x) = (x - 3) >> 2$

To build $\textsc{Frog}^\bullet$, let us consider a context h, a list of branches $l = (t_1, \mathcal{T}_1) \dots (t_n, \mathcal{T}_n)$, and the patterns $p_i = \text{Tag2Pat}(t_i)$ built using the function from Definition 1. Since $\textsc{Repr}^\bullet_\tau$ is injective we can use the p_i patterns to build a corresponding covering partition of $Img_\tau(h)$ which can then be translated to a decision tree using arithmetic or bit-word tests, whose leaves are the $\mathcal{T}_i$.

Example 15 (Packed representation, cont')**.** Concretely, here are two examples of patterns following the preceding example:

$$\textsc{Frog}^\bullet_{\tau_0}(\square, x)((None, \mathcal{T}_{None}), (Some, \mathcal{T}_{Some})) = \mathsf{switch}\,(x) \begin{cases} 0 & \to & \mathcal{T}_{None} \\ \top & \to & \mathcal{T}_{Some} \end{cases}$$

$$\textsc{Frog}^\bullet_{\tau_0}(Some(\square), x)((A, \mathcal{T}_A), (B, \mathcal{T}_B), (C, \mathcal{T}_C)) = \mathsf{switch}\,(x) \begin{cases} 0 & \to & \mathsf{unreachable} \\ 1 & \to & \mathcal{T}_A \\ 2 & \to & \mathcal{T}_B \\ \top & \to & \mathcal{T}_C \end{cases}$$

Note how the second switch handles the case 0: indeed, 0, which represents $None$, is not in $Img_{\tau_0}(Some(\square))$ and can thus never be reached.

Lemma 3. The packed representation is a valid representation.

As said before, the packed representation defined here is an extreme example, forgoing many optimizations available in more reasonable representations such as sharing or easy subterm extraction. Nevertheless, it shows that our framework is amenable to realistic representation with *some* packing and, given an appropriate definition of $\textsc{Frog}^\bullet$, will yield efficient pattern matching code.

7 Correctness

We now state the end-to-end correctness of our compilation scheme. The complete proofs are given in Appendix C. Let us first present our hypothesis.

Exhaustivity As stated in Section 3, we only consider patterns which are *exhaustive*: i.e. patterns handle all their possible input values.

More formally, let p a pattern, τ a type, v a value and Γ a typing environment.

$$\vdash p : \tau \to \Gamma \wedge \vdash v : \tau \implies \exists \sigma.\ p \rhd v \to \sigma \qquad \text{(Exhaustivity)}$$

Sound analysis for exhaustiveness have already been proposed Maranget (2007), even in the presence of powerful pattern languages Karachalias et al. (2015); Garrigue and Normand (2015). We assume such analysis was used to complete non-exhaustive patterns with wildcards as necessary.

7.1 Typing preservation for pattern matching

To ensure that the semantic of pattern is sound with respect to its typing, we state two properties, corresponding to preservation in expression languages.

In order to do so, we first define a typing judgment on values, denoted $\vdash v : \tau$, which is defined by a straightforward restriction of the typing on patterns to values. We also equip ourselves with a typing judgment on environments, denoted $\Gamma \vdash \sigma$. Both judgments are defined fully in Appendix C.

Theorem 1. *Typing preservation*

When a value matches, the bound value environment is well typed with respect to the bound typing environment. More formally, let p a pattern, τ a type, v a value. Let Γ and σ such that

$$\vdash p : \tau \to \Gamma \qquad \vdash v : \tau \qquad p \rhd v \to \sigma$$

Then, the dynamic and static bound variables coincide: $\Gamma \vdash \sigma$.

Proof. By induction. See Appendix C.2.

7.2 Soundness

Theorem 2. *Equivalence between pattern matching and compilation*

Let $p^1, \dots, p^m$ be m patterns and τ_Δ, γ such that $\forall j, \vdash p^j : \tau_\Delta \to \gamma$. Let v be a value of type τ_Δ. Let $\mathcal{T} = \textsc{Compile}^{\circ}_{\tau_\Delta}\left(\begin{array}{c|c} \square & \\ \hline p^1 & 1, \emptyset \\ \vdots & \vdots \\ p^m & m, \emptyset \end{array}\right)$. *Then for any case j and store σ:*

$$p^1 \mid \dots \mid p^m \rhd v \to j, \sigma \iff \mathcal{T}\left[\textsc{Repr}^{\bullet}_{\tau_\Delta}(v)\right] \hookrightarrow j, \{x \mapsto \textsc{Repr}^{\bullet}(y)\}_{(x \mapsto y) \in \sigma}$$

Proof. Proof sketch. See Appendix C.3 for the full proof.

- We define a *sequence* of pattern matrices from which $\textsc{Compile}^{\circ}_{\tau_\Delta}$ generates the nodes that are traversed during execution on input $\textsc{Repr}^{\bullet}_{\tau_\Delta}(v)$.
- $p^j \rhd v$ holds iff. any matrix in the sequence contains at least one row of $\mathcal{P}_k$ that outputs j and accepts the values obtained by focusing v on the contexts in $\mathcal{P}_k$.
- If the algorithm ends with $\mathsf{success}(j, s)$ and $p^j \rhd v \to \sigma$, then s and σ are compatible.
- The algorithm terminates (i.e., the sequence is finite).
- By induction from the last matrix in the sequence, and using the previous results, the theorem holds for the whole generated decision tree.

□

8 Related Works

8.1 Memory representation and Algebraic Data Types

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 optimized 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 behavior and parallelism. Our work would augment these approaches with efficient decision tree generation.

8.2 Pattern languages

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.

Pattern matching is also used pervasively in dependently typed languages Cockx et al. (2016); Tuerk et al. (2015). Since our definition of KNIT$^\bullet$ and FROG$^\bullet$ takes the type into account, we can't simply delegate the typing to a previous phase and operate on an untyped representation. One difficulty in dependent pattern matching (and in GADTs Garrigue and Normand (2015); Karachalias et al. (2015)) is that matching on a column can reveal the type of other columns. To adapt our framework to this setting, we could keep track of columns whose type is not revealed yet, and prioritize the decomposition of other columns. Columns would still have a coherent type, thanks to the split of the pattern matrix. OCaml implements a simplified version of this (it always splits on the first column, and type dependency can only go towards previous columns).

Active patterns Syme et al. (2007) from F# allow users to abstract over patterns by exposing "constructors" which do not directly reflect the underlying definition of the algebraic data type. The original formulation allows active patterns to execute arbitrary code during matching, making optimization difficult. Pattern synonyms Pickering et al. (2016) restrict patterns to be closer to the actual implementation, making it more amenable to optimization procedures such as ours.

8.3 Pattern Matching Optimization

Optimization of pattern matching for algebraic data types was first proposed by Augustsson (1985) in the context of the LML language. It first introduced the notion of pattern matrices and produced a backtracking automaton. Optimization of pattern matching is significantly more delicate in *lazy* languages (such as LML), since matching should avoid forcing unnecessary terms. Several semantics and associated optimizations have been proposed (Puel and Suárez, 1993; Maranget, 1992; Wadler, 1987), the last of which is used the current implementation of GHC (Marlow et al., 2004).

For *strict* languages however, Baudinet and MacQueen (1985) remark that the different parts of the discriminant can be evaluated in any order, allowing for more optimization opportunities. Fessant and Maranget (2001) first proposed optimizations 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. Sestoft (1996) emits a rough tree of `if` nodes, and relies on a global supercompilation pass to optimize the resulting tree. It is not clear how to make it parametric in term of the memory representation. Kosarev et al. (2020) explore a different optimization technique by encoding the choice of optimal decision tree into a relational synthesis problem, and solving through miniKanren. Their idea is very promising, but fails to scale to big matches. Solodkyy et al. (2013) propose patterns-as-library for C++ based on objects and template meta-programming. In all these cases, DAGs with maximal sharing can be created from trees by using Hash-consing Filliâtre and Conchon (2006).

Most approaches rely on heuristics for the choice of column to split (albeit with a limited choice for lazy languages). Maranget (2008) introduces the "Necessity" heuristic. 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 is a much bigger factor.

9 Conclusion

We have presented Knit&Frog, a technique to compile pattern matching parameterized by an arbitrary memory representation. On our way, we have given a novel way to define memory representations, along

with a validity criterion that ensures such a representation is appropriate for pattern matching. We used this criterion to prove the correctness of our compilation procedure. Finally, we illustrated our approach on several examples, including a realistic description of the OCaml memory representation. To our knowledge, this is the first generic description of memory representation, and the relation with Algebraic Data Types. 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 formalization and description of new optimization techniques for memory representation. In recent versions, Rust has been introducing more and more complex memory representation optimizations, which are so far unspecified. Furthermore, we believe there is a trove of untapped optimization yet to be explored when it comes to the memory representation of Algebraic Data Types. Some promising leads would be to apply super-optimization to individual performance-sensitive data structures, or to allow programmers to specify whether types should be optimized for space, cache behavior, or even best sharing. We hope this work serves as a stepping stone for these further optimizations.

$$\text{Tag2Word}_\tau(c) = (c << 1 \mid \texttt{0x1}) \qquad \text{when } \tau = T \in \mathcal{C}$$

$$\text{Tag2Word}_{\sum K_i(\overline{\tau_{i,j}}^{J(i)})}(K_{i_0}) = \begin{cases} \text{Tag2Word}_{Int}(nth(K, \{K_i \mid J(i) = 0\})) & \text{when } J(i_0) = 0 \\ nth(K, \{K_i \mid J(i) > 0\}) & \text{when } J(i_0) > 0 \end{cases}$$

(a) The Tag2Word utility function

$$\textsc{Repr}^\bullet_T(c) = \text{Tag2Word}_T(c) \qquad \textsc{Repr}^\bullet_{\prod \overline{\tau_i}}(\langle \overline{v_i} \rangle) = \&_{64}\left[0;\ \overline{\textsc{Repr}^\bullet_{\tau_i}(v_i)}\right]_{64}$$

$$\textsc{Repr}^\bullet_\tau(K_i) = \text{Tag2Word}_\tau(K_i) \qquad \textsc{Repr}^\bullet_{\tau=\sum K_i(\overline{\tau_{i,j}})}(K_i(\overline{v_j})) = \&_{64}\left[\text{Tag2Word}_\tau(K_i);\ \overline{\textsc{Repr}^\bullet_{\tau_{i,j}}(v_j)}\right]_{64}$$

(b) Memory values

$$f(\square, e) = e \qquad f(\langle \_, \ldots, h_k, \ldots, \_ \rangle, e) = f(K(\_, \ldots, h_k, \ldots, \_), e) = f(h_k, k + 1. * e)$$

$$\textsc{Knit}^\bullet_\tau(h) = f(h, \Delta)$$

(c) Knitting

$$\textsc{Frog}^\bullet_\tau(h)\left(\overline{(t_k, \mathcal{T}_k)}\right) = \mathsf{switch}\,(f(h,\Delta))\left\{\overline{\text{Tag2Word}_\tau(t_k) \mapsto \mathcal{T}_k}\right.$$
$$\text{where } \textbf{focus}\,(h,\tau) = T \in \mathcal{C}$$

$$\textsc{Frog}^\bullet_\tau(h)\left(\overline{(K_{i_k}, \mathcal{T}_{i_k})} \cdot (\top, \mathcal{T}_*)\right) = \mathsf{switch}\,(f(h,\Delta))\begin{cases}\overline{\text{Tag2Word}_\tau(K_{i_k}) \mapsto \mathcal{T}_{i_k}} \\ \top \mapsto \mathcal{T}_*\end{cases}$$
$$\text{where } \textbf{focus}\,(h,\tau) = \sum K_i(\overline{\tau_{i,j}}^{J(i)}) \text{ and } \forall k, J(i_k) = 0$$

$$\textsc{Frog}^\bullet_\tau(h)\left(\overline{(K_{i_k}, \mathcal{T}_{i_k})} \cdot (\top, \mathcal{T}_*)\right) = \mathsf{switch}\,(f(h,\Delta)\ \&\ \texttt{0x1})\begin{cases} 1 \mapsto \textsc{Frog}^\bullet_\tau(h)\left(\overline{(K_{i_k}, \mathcal{T}_{i_k})}_{J(i_k)=0} \cdot (\top, \mathcal{T}_*)\right) \\ 0 \mapsto \mathsf{switch}\,(*(f(h,\Delta)).1) \\ \qquad \begin{cases}\overline{\text{Tag2Word}_\tau(K_{i_k}) \mapsto \mathcal{T}_{i_k}} \\ \top \mapsto \mathcal{T}_*\end{cases}\end{cases}$$
$$\text{where } \textbf{focus}\,(h,\tau) = \sum K_i(\overline{\tau_{i,j}}^{J(i)})$$

(d) Frogging

Figure 15: The OCaml representation

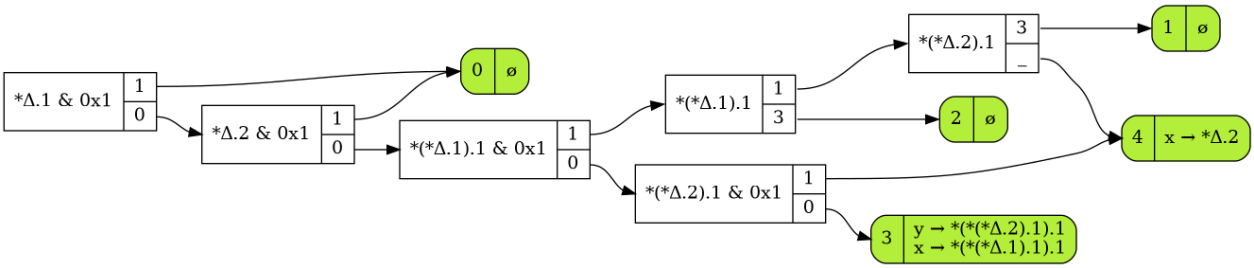


Figure 16: The decision tree for Fig. 14 with the OCaml representation, using our prototype tool `ribbit`.

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

Marianne Baudinet and David MacQueen. 1985. Tree pattern matching for ML. (1985).

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

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

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

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

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

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

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

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

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

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

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

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

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

Simon Marlow, Simon Peyton Jones, et al. 2004. The glasgow haskell compiler.

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

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

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

Laurence Puel and Ascánder Suárez. 1993. Compiling Pattern Matching by Term Decomposition. *J. Symb. Comput.* 15, 1 (1993), 1–26. https://doi.org/10.1006/jsco.1993.1001

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

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

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

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

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

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

Pierre Wilke. 2016. *Formally verified compilation of low-level C code. (Compilation formellement vérifiée de code C de bas-niveau).* Ph. D. Dissertation. University of Rennes 1, France. `https://tel.archives-ouvertes.fr/tel-01483676`

A Typing of values

In this section, we define the typing judgment $\vdash v : \tau$ for values and $\vdash \sigma : \Gamma$ for environments in Fig. 17. The rules are a direct subset of the rules for patterns presented in Section 3.

$$\textsc{Constant}\ \frac{c \in T}{\vdash c : T} \qquad \textsc{Reference}\ \frac{\vdash v : \tau}{\vdash \&v : \&\tau} \qquad \textsc{Tuple}\ \frac{\forall i, \vdash v_i : \tau_i}{\vdash \langle \overline{v_i} \rangle : \prod \tau_i} \qquad \textsc{Constructor}\ \frac{\exists i, K = K_i \qquad \forall j, \vdash v_j : \tau_{i,j}}{\vdash K(\overline{v_j}) : \sum K_i(\overline{\tau i, j})}$$

$$\textsc{Env}\ \frac{\forall i, \vdash v_i : \tau_i}{\vdash \{x_i \mapsto v_i\} : \{x_i : \tau_i\}}$$

Figure 17: Typing of values and environments – $\vdash v : \tau$

B Contexts and focusing

We initially introduced the focus judgment in Section 4.2.2 as a way to manipulate values and types with respect to a context h. We defined two operations: $\mathbf{focus}\,(h, v)$ which focuses a value and returns a value, and $\mathbf{focus}\,(h, \tau)$ which focuses a type and returns a type. For the purpose of the proof, we introduce an additional one: $\mathbf{focus}\,(h, p)$, which focuses a pattern and returns a pattern, by collecting all the alternatives. The complete definitions are given in Fig. 18.

C Proof of correctness

We now give the proof of correctness of the theorems stated in Section 7: the typing preservation, and the soundness of our compilation procedure. We also define a few convenience results on the way.

C.1 Coherence of focusing

We use focusing pervasively in our proof. Let us state a convenience result which shows that focus and typing of values are coherent:

Lemma 4. Let h a context, v a value, τ a type.

$$\tau \vdash v \wedge \mathbf{focus}\,(h, v) = v' \wedge \mathbf{focus}\,(h, \tau) = \tau' \implies \tau' \vdash v'$$

Proof. By induction over the typing judgment. □

C.2 Typing preservation

Proof of Theorem 1. Let p a pattern, τ a type, v a value. Let Γ and σ such that

$$\vdash p : \tau \to \Gamma \qquad \vdash v : \tau \qquad p \rhd v \to \sigma$$

By induction on p, we show that $x \in \Gamma \iff x \in \sigma$ and $\forall x \in \Gamma, \vdash \sigma(x) : \Gamma(x)$.

Wildcard or constant pattern no bound variables: $\Gamma = \sigma = \emptyset$.
Variable pattern $(p = x)$ x is the only bound variable: $\Gamma = \{x : \tau\}$; $\sigma = \{x \mapsto v\}$. Since $\vdash v : \tau$, we have $\vdash \sigma(x) : \tau$.

$$\mathbf{focus}\,(\square, v) = v \qquad \mathbf{focus}\,(\&h, \&v) = \mathbf{focus}\,(h, v)$$

$$\mathbf{focus}\,(\langle \_, \ldots, \_, h, \_, \ldots, \_\rangle, \langle \_, \ldots, \_, v, \_, \ldots, \_\rangle) = \mathbf{focus}\,(h, v)$$

$$\mathbf{focus}\,(K(\_, \ldots, \_, h, \_, \ldots, \_), K(\_, \ldots, \_, v, \_, \ldots, \_)) = \mathbf{focus}\,(h, v)$$

(a) Focus on values – $\mathbf{focus}\,(h, v)$

$$\mathbf{focus}\,(\square, p) = p \qquad \mathbf{focus}\,(\&h, \&p) = \mathbf{focus}\,(h, p)$$

$$\mathbf{focus}\,(\langle \_, \ldots, \_, h, \_, \ldots, \_\rangle, \langle \_, \ldots, \_, p, \_, \ldots, \_\rangle) = \mathbf{focus}\,(h, p)$$

$$\mathbf{focus}\,(K(\_, \ldots, \_, h, \_, \ldots, \_), K(\_, \ldots, \_, p, \_, \ldots, \_)) = \mathbf{focus}\,(h, p)$$

$$\mathbf{focus}\,(h, p_1 \mid p_2) = \mathbf{focus}\,(h, p_1) \mid \mathbf{focus}\,(h, p_2)$$

(b) Focus on patterns – $\mathbf{focus}\,(h, p)$

$$\mathbf{focus}\,(\square, \tau) = \tau \qquad \mathbf{focus}\,(\&h, \&\tau) = \mathbf{focus}\,(h, \tau) \qquad \mathbf{focus}\left(\left\langle \_, \ldots, \_, \underset{k}{h}, \_, \ldots, \_ \right\rangle, \prod \tau_i\right) = \mathbf{focus}\,(h, \tau_k)$$

$$\mathbf{focus}\left(K_{i_0}(\_, \ldots, \_, \underset{k}{h}, \_, \ldots, \_), \sum_{0 \le i < n} K_i(\overline{\tau_{i,j}})\right) = \mathbf{focus}\,(h, \tau_{i_0,k}) \text{ with } 0 \le i_0 < n$$

(c) Focus on types – $\mathbf{focus}\,(h, \tau)$

Figure 18: Focusing judgments

Reference ($p = \&p'$) let v' and τ' such that $\vdash p' : \tau', \Gamma$, $v = \&v'$ and $p' \rhd v', \sigma$. From the Lemma 4, we have $\vdash v' : \tau'$. Any variable bound in p is bound in p'; from the induction hypothesis, we have $x \in \Gamma \iff x \in \sigma$ and $\forall x \in \Gamma, \vdash \sigma(x) : \Gamma(x)$ for any bound variable x.
Tuple ($p = \langle \overline{p_k} \rangle$) let $\overline{\tau_k}$, $\overline{v_k}$, $\overline{\Gamma_k}$ and $\overline{\sigma_k}$ such that:

$$\forall k, \vdash p_k : \tau_k, \Gamma_k \qquad \forall k, p_k \rhd v_k, \sigma_k \qquad v = \langle \overline{v_k} \rangle \qquad \forall k, \vdash v_k : \tau_k \text{ (Lemma 4)} \qquad \tau = \prod \tau_k \qquad \Gamma = \bigsqcup \Gamma_k$$

$$\sigma = \bigsqcup \sigma_k$$

For any bound variable $x \in \Gamma$, there is exactly one k such that $x \in \Gamma_k$, $\Gamma(x) = \Gamma_k(x)$ and $\sigma(x) = \sigma_k(x)$. From the induction hypothesis, we have $x \in \sigma_k$ and $\vdash \sigma_k(x) : \Gamma_k(x)$.
Conversely, for any $x \in \sigma$, there is exactly one k such that $x \in \sigma_k$; from the induction hypothesis, we have $x \in \Gamma_k \subset \Gamma$.
Constructor pattern ($p = K(\overline{p_k})$) let $\tau = \sum K_i(\overline{\tau_{i,k}})$ and i such that $K = K_i$ and $\forall k, \vdash p_k : \tau_k$. Let $\overline{v_k}$, $\overline{\Gamma_k}$ and $\overline{\sigma_k}$ such that:

$$\forall k, v = K(\overline{v_k}) \qquad \forall k, \vdash v_k : \tau_k \text{ (Lemma 4)} \qquad \forall k, \vdash p_k : \tau_k, \Gamma_k \qquad \forall k, p_k \rhd v_k, \sigma_k \qquad \Gamma = \bigcup \Gamma_k \qquad \sigma = \bigcup \sigma_k$$

We show that the properties hold as in the previous case.
Disjunction ($p = p_1 | p_2$) we have $\vdash p_1 : \tau, \Gamma$, $\vdash p_2 : \tau, \Gamma$ and either $p_1 \rhd v, \sigma$ or $p_2 \rhd v, \sigma$. The variables bound in p are bound in p_1 or p_2. In both cases, from the induction hypothesis, we have $x \in \Gamma \iff x \in \sigma$ and $\forall x \in \Gamma, \vdash \sigma(x) : \Gamma(x)$.

□

C.3 Soundness

Proof of Theorem 2. Let $p_0^0, \dots, p_0^{m-1}$ be m (initial) patterns and τ_Δ, γ such that $\forall j, \vdash p_0^j : \tau_\Delta \to \gamma$. Let v be a value such that $\vdash v : \tau_\Delta$ and $r = \textsc{Repr}^\bullet_{\tau_\Delta}(v)$. Finally, let $\mathcal{P}_0 = \left(\begin{array}{c|c} \square & \\ \hline p_0^0 & 0, \emptyset \\ \vdots & \vdots \\ p_0^{m-1} & m-1, \emptyset \end{array}\right)$.

We show that for any j, σ, $p_0^0 | \dots | p_0^{m-1} \rhd v \to j, \sigma$ iff. $\textsc{Compile}^\circ_{\tau_\Delta}(\mathcal{P}_0)[r] \hookrightarrow j, \{x \mapsto \textsc{Repr}^\bullet(y)\}_{x \in \sigma}$.

C.3.1 Compilation steps

We define a sequence of pattern matrices corresponding to the $\textsc{Compile}^\circ_{\tau_\Delta}$ calls that define those nodes that are part of the execution path on input r.

For any pattern matrix $\mathcal{P}_k$ in the sequence, we write $\mathcal{P}_k = \left(\begin{array}{ccc|c} h_{k,0} & \dots & h_{k,n_k-1} & \\ \hline p_{k,0}^0 & \dots & p_{k,n_k-1}^0 & j_k^0, s_k^0 \\ \vdots & \ddots & \vdots & \vdots \\ p_{k,0}^{m_k-1} & \dots & p_{k,n_k-1}^{m_k-1} & j_k^{m_k-1}, s_k^{m_k-1} \end{array}\right)$.

We define the sequence of relevant pattern matrices inductively:

- The first pattern matrix is $\mathcal{P}_0$, as defined above.
- If $\mathcal{P}_k$ belongs to the sequence, we follow the definition of $\textsc{Compile}^\circ_{\tau_\Delta}$ to determine the remaining steps:
  - If $\mathcal{P}_k = \emptyset / m_k = 0$, the sequence ends (no pattern case).
  - If $\mathbf{Pos}_k^0 = \emptyset$, the sequence ends (wildcard case).
  - If the variable (resp. disjunction) case applies and we lift (resp. split) the pattern in column i and row ℓ, we define the next matrix $\mathcal{P}_{k+1}$ according to $\textsc{Compile}^\circ_{\tau_\Delta}$ and write $\mathcal{P}_k \to_{\text{var},i,\ell} \mathcal{P}_{k+1}$ (resp. $\mathcal{P}_k \to_{\text{or},i,\ell} \mathcal{P}_{k+1}$).
  - Otherwise, the switch case applies. Let i the column chosen by $\textsc{PickColumn}$. We have $h_{k,i} \in \mathbf{Pos}_k^1$. Let $\tau_i = \mathbf{focus}(h_{k,i}, \tau_\Delta)$ and t be the first tag generated by $\textsc{GetTags}[\tau_i]$ that matches $\mathbf{focus}(h_{k,i}, v)$. $\textsc{Frog}^\bullet$ (used in the last step of the switch case of $\textsc{Compile}^\circ_{\tau_\Delta}$) guarantees that at this point, the execution path on input r will branch to the subtree associated with t. This subtree is generated by $\textsc{CompileBranch}^\circ[\tau_\Delta](\mathcal{P}_k)(i,t)$, which performs one recursive call to $\textsc{Compile}^\circ_{\tau_\Delta}$. Let $\mathcal{P}_{k+1}$ be the pattern matrix built by $\textsc{CompileBranch}^\circ$ and passed to $\textsc{Compile}^\circ_{\tau_\Delta}$ for this recursive call. Then $\mathcal{P}_{k+1}$ is the next matrix in the sequence, and we write $\mathcal{P}_k \to_{\text{switch},i,t} \mathcal{P}k+1$.

Finally, for any matrix $\mathcal{P}_k$ in the sequence and initial case j, let $\mathcal{P}_k^j$ be the (possibly empty) sub-matrix of $\mathcal{P}_k$ that contains all rows such that $j_k^\ell = j$. We trivially show by induction on k that these sub-matrices never overlap: $\mathcal{P}_k = \begin{pmatrix} \mathcal{P}_k^0 \\ \vdots \\ \mathcal{P}_k^{m-1} \end{pmatrix}$.

C.3.2 Case matching equivalence

Proposition 1. *Let j an initial case and $\mathcal{P}_k$ a pattern matrix in the sequence. We have $p_0^j \rhd v$ iff. $\exists \ell, j_k^\ell = j \wedge \forall i, p_{k,i}^\ell \rhd \boldsymbol{focus}(h_{k,i}, v)$.*

Proof. By induction on k. The base case is trivial ($n_0 = 1$, $p_{0,0}^j = p_0^j$ and $h_{0,0} = \square$). We now consider the case where $k > 0$ and suppose we have

$$p_0^j \rhd v \iff \exists \ell, j_{k-1}^\ell = j \wedge \forall i, p_{k-1,i}^\ell \rhd \mathbf{focus}(h_{k-1,i}, v)$$

We show that the property carries over to the next pattern matrix $\mathcal{P}_k$ by induction on the transition step from $\mathcal{P}_{k-1}$ to $\mathcal{P}_k$:

Variable case: $\mathcal{P}_{k-1} \to_{\mathbf{var},i,\ell} \mathcal{P}_k$**.** We have $p^{\ell'}_{k,i'} = p^{\ell'}_{k-1,i'}$ if $i' \neq i$ or $\ell' \neq \ell$, $p^{\ell}_{k,i} = \_$ and $p^{\ell}_{k-1,i}$ is a variable pattern. In both cases, we have

$$p^{\ell}_{k,i} \rhd \mathbf{focus}\,(h_{k,i}, v) \iff p^{\ell}_{k-1,i} \rhd \mathbf{focus}\,(h_{k-1,i}, v)$$

therefore, the property holds for $\mathcal{P}_k$.
Disjunction case: $\mathcal{P}_{k-1} \to_{\mathbf{or},i,\ell} \mathcal{P}_k$**.** We have $p^{\ell}_{k-1,i} \rhd \mathbf{focus}\,(h_{k-1,i}, v) \iff p^{\ell}_{k,i} \rhd \mathbf{focus}\,(h_{k,i}, v) \vee p^{\ell+1}_{k,i} \rhd \mathbf{focus}\,(h_{k,i}, v)$, and since the other patterns of ℓ are unchanged,

$$(\forall i, p^{\ell}_{k-1,i} \rhd \mathbf{focus}\,(h_{k-1,i}, v)) \iff (\forall i, p^{\ell}_{k,i} \rhd \mathbf{focus}\,(h_{k,i}, v)) \vee (\forall i, p^{\ell+1}_{k,i} \rhd \mathbf{focus}\,(h_{k,i}, v))$$

As the other rows are unchanged, the property holds for $\mathcal{P}_k$.
Switch case $\mathcal{P}_{k-1} \to_{\mathbf{switch},i,t} \mathcal{P}_k$ Let $\mathcal{H}, f = \textsc{Expand}^{\circ}(\tau_i, t, h_{k,i})$. Since the patterns outside of column i are unchanged, it is sufficient to prove that

$$p^{\ell}_{k-1,i} \rhd \mathbf{focus}\,(h_{k-1,i}, v) \iff f(p^{\ell}_{k-1,i}) = \left(\overline{p^{\ell'}_{k,i'}}\right) \wedge \forall i' \in \mathcal{H}, p^{\ell'}_{k,i'} \rhd \mathbf{focus}\,(h_{k,i'}, v)$$

We have $p^{\ell}_{k-1,i} \rhd \mathbf{focus}\,(h_{k-1,i}, v)$ iff.

1. their head constructors (if any) match and
2. $\forall h_{k-1,i}[h] \in \mathcal{H}, \mathbf{focus}\left(h, p^{\ell}_{k-1,i}\right) \rhd \mathbf{focus}\,(h_{k-1,i}[h], v)$.

From the definitions of $\textsc{Expand}^{\circ}$ and the sequence, we have

1. If $f(p^{\ell}_{k-1,i}) = \emptyset$, then:

   $$p^{\ell}_{k-1,i} = c \wedge t \neq c \vee p^{\ell}_{k-1,i} = K(\dots) \wedge t \neq K \iff$$
   $$p^{\ell}_{k-1,i} = c \wedge \mathbf{focus}\,(h_{k-1,i}, v) = c' \wedge c' \neq c \vee p^{\ell}_{k-1,i} = K(\dots) \wedge \mathbf{focus}\,(h_{k-1,i}, v) = K'(\dots) \wedge K' \neq K$$

2. If $f(p^{\ell}_{k-1,i}) \neq \emptyset$, let ℓ' be the row of $\mathcal{P}_k$ yielded by $f(p^{\ell}_{k-1,i})$.
   Then $\forall h_{k,i'} = h_{k-1,i}[h] \in \mathcal{H}, p^{\ell'}_{k,i'} = \mathbf{focus}\left(h, p^{\ell}_{k-1,i}\right)$ and thus:

   $$p^{\ell'}_{k,i'} \rhd \mathbf{focus}\,(h_{k,i'}, v) \iff \mathbf{focus}\left(h, p^{\ell}_{k-1,i}\right) \rhd \mathbf{focus}\,(h_{k-1,i}[h], v)$$

And thus the property holds.

□

C.3.3 Bindings compatibility

For any pattern matrix of the sequence $\mathcal{P}_k$ and any row ℓ of this matrix, we define the set of contexts that designate variable subterms of the patterns of this row:

$$\mathbf{Var}^{\ell}_k = \left\{h \mid \exists h_{k,i}, h'.h = h_{k,i}[h'] \wedge V(\mathbf{focus}\left(h', p^{\ell}_{k,i}\right)) \neq \emptyset\right\}$$
$$\text{where } V(p) = \begin{cases} \{x\} & \text{if } p = x \text{ is a variable pattern} \\ V(q^1) & \text{if } p = q^1|q^2 \text{ and } V(q^1) = V(q^2) \\ \emptyset & \text{otherwise} \end{cases}$$

Lemma 5. Let j an initial case. If $p^j_0 \rhd v \to \sigma$, then

$$\sigma = \left\{x \mapsto \mathbf{focus}\,(h, v) \mid V(\mathbf{focus}\left(h, p^j_0\right)) = \{x\}\right\}$$

Proof. Trivial by induction on $\rhd$.

Lemma 6. Let $\mathcal{P}_k$ a pattern matrix in the sequence and ℓ a row of $\mathcal{P}_k$. Let $j = j^{\ell}_k$. If $p^{\ell}_{k,i} \neq \_$, then $V(p^{\ell}_{k,i}) = V(\mathbf{focus}\left(h_{k,i}, p^j_0\right))$.

Proof. By induction on k

- if $k = 0$, we have $V(p_0^j) = V(\mathbf{focus}\left(\Box, p_0^j\right))$.
- if $k > 0$, suppose the lemma holds for $\mathcal{P}_{k-1}$.
  - If $\mathcal{P}_{k-1} \to_{\mathrm{var},i,\ell} \mathcal{P}_k$, then $p_{k,i}^\ell = \_$ and since the other patterns are unchanged, for every i', ℓ' such that $p_{k,i'}^{\ell'} \neq \_$, we have:
$$V(p_{k,i'}^{\ell'}) = V(p_{k-1,i'}^{\ell'}) = V(\mathbf{focus}\left(h_{k,i}, p_0^{j_k^{\ell'}}\right))$$
  - If $\mathcal{P}_{k-1} \to_{\mathrm{or}} \mathcal{P}_k$ or $\mathcal{P}_{k-1} \to_{\mathrm{switch}} \mathcal{P}_k$, then $V(p_i^\ell)$ is unchanged for every i and ℓ and the lemma still holds.

□

Lemma 7. Let $\mathcal{P}_k$ a pattern matrix in the sequence and ℓ a row of $\mathcal{P}_k$. Let $j = j_k^\ell$. We have

$$s_k^\ell = \left\{x \mapsto \textsc{Knit}^\bullet_{\tau_\Delta}(h) \mid h \in \mathbf{Var}_0^j \setminus \mathbf{Var}_k^\ell \wedge V(\mathbf{focus}\left(h, p_0^j\right)) = \{x\}\right\}$$

.

Proof. By induction on k

- if $k = 0$, ℓ is itself an initial case j and $s_0^j = \emptyset = \mathbf{Var}_0^j \setminus \mathbf{Var}_0^j$.
- if $k > 0$, suppose the property holds for every row of $\mathcal{P}_{k-1}$. If $\mathcal{P}_{k-1} \to_{\mathrm{or}} \mathcal{P}_k$ or $\mathcal{P}_{k-1} \to_{\mathrm{switch}} \mathcal{P}_k$, the property still holds for every row as j^ℓ, s^ℓ and $\mathbf{Var}^\ell$ are unchanged.
  If $\mathcal{P}_{k-1} \to_{\mathrm{var},\ell,i} \mathcal{P}_k$, let $x = p_{k-1,i}^\ell$. We have $h_{k-1,i} \in \mathbf{Var}_{k-1}^\ell$, $\mathbf{Var}_k^\ell = \mathbf{Var}_{k-1}^\ell \setminus \{h_{k-1,i}\}$ and $s_k^\ell = s_{k-1}^\ell \cup \{x \mapsto \textsc{Knit}^\bullet_{\tau_\Delta}(h_{k-1,i})\}$. From Lemma 6, we have $V(\mathbf{focus}\left(h_{k-1,i}, p_0^j\right)) = \{x\}$ and thus the property holds for the row ℓ. It also holds for other rows, which are unchanged.

□

Proposition 2. *Let j an initial case. If $p_0^j \rhd v \to \sigma$ and the last pattern matrix of the sequence is $\mathcal{P}_k$ with $\textsc{Compile}^\circ_{\tau_\Delta}(\mathcal{P}_k) = \mathsf{success}(j, s_k^0)$, then we have*

$$x \in s_k^0 \iff x \in \sigma \text{ and } \forall x \in \sigma, s_k^0(x)[r] \hookrightarrow \textsc{Repr}^\bullet(\sigma(x))$$

Proof. Since the sequence ends with $\mathcal{P}_k$ (wildcard case), we have $\mathbf{Var}_k^0 = \emptyset$. The first part of the proposition then follows from Lemmas 5 and 7:

$$x \in s_k^0 \iff \exists h, V(\mathbf{focus}\left(h, p_0^j\right)) = \{x\} \iff (x \mapsto \mathbf{focus}\,(h, v)) \in \sigma$$

Let x be a variable bound in s_k^0. We have $s_k^0(x) = \textsc{Knit}^\bullet_{\tau_\Delta}(h)$ and $\sigma(x) = \mathbf{focus}\,(h, v)$ for some context h. $\textsc{Knit}^\bullet$ guarantees that $\textsc{Knit}^\bullet_{\tau_\Delta}(h)[r] \hookrightarrow \textsc{Repr}^\bullet_{\tau_\Delta}(\mathbf{focus}\,(h, v))$, thereby proving the property. □

C.3.4 Termination

We define, for every matrix in the sequence $\mathcal{P}_k$ and every row ℓ of $\mathcal{P}_k$, the set of contexts such that the corresponding pattern in row ℓ is not a wildcard: $\mathbf{Pos}_k^\ell = \{h_{k,i} \mid p_{k,i}^\ell \neq \_\}$.

We also define a function similar to V: $O(p) = \begin{cases} O(q^1) + O(q^2) + 1 & \text{if } p = q^1|q^2 \\ 0 & \text{otherwise} \end{cases}$.

Let $\mathcal{O}_k = \sum \{O(p) \mid \exists h_{k,i}, h, \ell.p = \mathbf{focus}\left(h, p_{k,i}^\ell\right)\}$.

Proposition 3. *Termination There exists a k_0 such that $\mathcal{P}_{k_0}$ is the last pattern matrix of the sequence, i.e., $\textsc{Compile}^\circ_{\tau_\Delta}(\mathcal{P}_{k_0}) = \mathsf{unreachable}$ (no pattern case) or $\textsc{Compile}^\circ_{\tau_\Delta}(\mathcal{P}_{k_0}) = \mathsf{success}(j, s)$ (wildcard case).*

Proof. We show that there is a finite number of each kind of transition between pattern matrices of the sequence.

Variable case We trivially show that if $\mathcal{P}_{k-1} \to_{\text{var}} \mathcal{P}_k$, then $\sum_{0\leq\ell<m_k} |\mathbf{Var}_k^\ell| < \sum_{0\leq\ell<m_{k-1}} |\mathbf{Var}_{k-1}^\ell|$. Therefore the sequence contains at most $\sum_{0\leq j<m} |\mathbf{Var}_0^j| \to_{\text{var}}$-steps.
Or case We trivially show that if $\mathcal{P}_{k-1} \to_{\text{or}} \mathcal{P}_k$, then $\mathcal{O}_k < \mathcal{O}_{k-1}$. Therefore the sequence contains at most $\mathcal{O}_0 \to_{\text{or}}$-steps.
Switch case We trivially show that if $\mathcal{P}_{k-1} \to_{\text{switch}} \mathcal{P}_k$, then $\mathbf{Pos}_k^1 \subsetneq \mathbf{Pos}_{k-1}^1$ or $m_k < m_{k-1}$ and therefore $\sum_{0\leq\ell<m_k} |\mathbf{Pos}_k^\ell| < \sum_{0\leq\ell<m_{k-1}} |\mathbf{Pos}_{k-1}^\ell|$. The sequence thus contains at most $\sum_{0\leq j<m} |\mathbf{Pos}_0^j|$.

□

C.3.5 Conclusion

Let $\mathcal{P}_k$ be the last pattern matrix of the sequence.

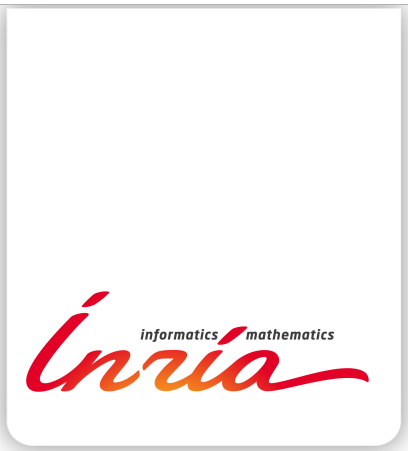
- If $\mathcal{P}_k = \emptyset$, then $\textsc{Compile}^\circ_{\tau_\Delta}(\mathcal{P}_k) =$ unreachable and from Proposition 1, we have $p_0^j \not\rhd v$ for every case j, that is $p_0^0 \mid \ldots \mid p_0^{m-1} \not\rhd v$.
- Otherwise, let j, s such that $\mathcal{P}_k = \left(\begin{array}{ccc|c} h_{k,0} & \cdots & h_{k,n_k-1} & \\ \hline _ & \cdots & _ & j, s \\ \vdots & \ddots & \vdots & \vdots \\ p_{k,0}^{m_k-1} & \cdots & p_{k,n_k-1}^{m_k-1} & j_k^{m_k-1}, s_k^{m_k-1} \end{array}\right)$.
  From Proposition 1, we have $p_0^j \rhd v$ and for all $j' < j$, $p_0^{j'} \not\rhd v$ (since $\mathcal{P}^{j'} = \emptyset$), that is $p_0^0 \mid \ldots \mid p_0^{m-1} \rhd v \to j$.
  Let σ such that $p_0^0 \mid \ldots \mid p_0^{m-1} \rhd v \to j, \sigma$. From Proposition 2, we have

$$s[r] \hookrightarrow \{x \mapsto \textsc{Repr}^\bullet(y)\}_{(x\mapsto y)\in\sigma}$$

  which concludes.

□

Contents



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